Extensibility and Efficiency of Top-Down Query Optimizers

Klaus Julisch

University of Stuttgart
Breitwiesenstr. 20-22
70565 Stuttgart, Germany

9th November 1999

Preface

The present thesis describes my work as a computer science masters student at the University of Stuttgart. My work as a masters student was integrated into the project *Parallelisation of Database Systems*, which is guided by Professor Bayer and Professor Mitschang. This project is part of the Sonderforschungsbereich 342 (SFB 342/B2), and is supported by the Deutsche Forschungsgesellschaft (DFG).

The goal of my research was to scrutinize the extensibility and efficiency of top-down query optimizers. As a concrete workbench for my studies, I used the optimizer of the MIDAS database prototype. This optimizer ranks as a top-down optimizer, and is thus well suited for my goal. While working with the MIDAS-optimizer, I also discovered and fixed several bugs in its code. These bug fixes are documented right away in the source code, so no further mention will be given to them here.

Throughout the thesis, I repeatedly use pseudo-code to describe algorithms. In general, this pseudo-code is very close to the actual source code. Sometimes, however, the source code is very complex or poorly structured. In these cases, I felt free to deviate from the source code and give pseudo-code which is semantically equivalent, but easier to understand. This should be kept in mind when comparing the pseudo-code of this thesis to the actual source code.

This thesis owes a debt to many individuals. Dr. Jaedicke, my adviser, deserves special mention for his role in guiding and reviewing my work. Professor Mitschang, the department head, deserves credit for giving me plenty of freedom in my work. Furthermore, Professor Mitschang also supported my research stay at the Portland State University (PSU). Finally, I am very grateful to Professor Shapiro from PSU for the many hours of stimulating discussions he had with me. Many thanks to all of them.

Klaus Julisch

Contents

Abstract **iii**

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Terminology . . . 2

1.3 Query Optimization Basics . . . 3

2 Cascades **8**

2.1 Cascades' Architecture . . . 8

2.2 Cascades' Data Structures . . . 12

2.2.1 Memo, Groups and Multi-Expressions . . . 13

2.2.2 Context and Winner's Circle . . . 15

2.3 Algorithmic Aspects of Cascades . . . 16

2.3.1 The Binding Algorithm . . . 16

2.3.2 Inserting into the Memo . . . 17

2.3.3 The Optimization Algorithm . . . 21

2.4 An Example . . . 24

3 Extensibility of Cascades **28**

3.1 Commuting JOIN and GROUP_BY . . . 28

3.2 Search Space Complexity . . . 30

3.2.1 Defining the Problem . . . 30

3.2.2 An Introductory Example . . . 33

3.2.3 The General Counting Argument . . . 35

3.2.4 Discussion . . . 36

3.3 Duplicate-Freeness of the Extended Rule-Set . . . 37

3.3.1 Duplicates in General . . . 38

3.3.2 Some Preliminaries . . . 38

3.3.3 Proving Duplicate-Freeness . . . 40

3.4 Generalization . . . 44

4 Improving the Code of Cascades **46**

4.1 Dynamic Arrays 47

4.2 A Better Binding Algorithm 48

4.3 A New Winner's Circle 49

4.4 Results 51

5 Group Pruning **54**

5.1 Definitions 54

5.2 The Group Pruning Strategy of Cascades 55

5.3 A Lower Bound for Groups 57

5.4 Results 62

6 Conclusion and Future Work **63**

6.1 Global Epsilon Pruning 63

6.2 Memory Management 66

6.3 Memory Requirements 66

6.4 Conclusion 67

A Failures in Tuning Cascades **69**

A.1 Searching the Memo 69

A.2 A More Succinct Representation for Multi-Expressions 71

Bibliography **73**

Index **76**

Abstract

Extensibility and Efficiency
of
Top-Down Query Optimizers

Klaus Julisch

Extensibility and efficiency are two key properties that every modern query optimizer must have. Top-down optimizers have traditionally claimed to score high with respect to extensibility. Since recently, they have also claimed to achieve a degree of efficiency which was formerly only attained by bottom-up optimizers. These claims were the impetus for the present thesis, which examines the extensibility and efficiency of top-down optimizers.

This thesis attributes the extensibility of top-down optimizers to their architecture. This architecture is based on sound software engineering principles and therefore facilitates the *implementation* of extensions. However, the major barrier to extending top-down optimizers is that they provide no guidance on *finding* practically usable extensions. In defense of this claim, the present thesis provides never before published treatment of *duplicates* and *search space complexity*. Both of these topics are key issues in finding practical extensions for top-down optimizers.

With respect to the efficiency of top-down optimizers, it will be shown that clever changes in the source code can lead to substantial efficiency gains. The presented efficiency increasing measures have been developed as part of the Columbia project, but this thesis is the first text to systematically assess their effectiveness.

Chapter 1

Introduction

1.1 Motivation

Even after 20 years of research in the area of SQL query optimization, there are still open problems to be solved. These problems emerged because user demand has constantly changed over the years. SQL, for example, had to be extended by object features which need to be supported by the query optimizer[1]. Paul Brown gives in [4] an overview of the most common of these extensions and argues that there's still a far way to go in order to completely satisfy user demand. Furthermore, new index types such as materialized views [2], join indexes [2], and bit indexes [27] have been used to speed up data warehouses. These indexes, in order to fully develop their potential, have to be supported by the query optimizer. Another clear development is that SQL queries have become more and more complex. This is illustrated by Table 1.1 which measures query complexity by the number of joins involved. Therefore, optimizer efficiency is a matter of concern these days. To summarize, it has become very clear that an up-to-date optimizer must meet the following two requirements:

(i) It must be extensible in order to keep up with new developments such as object extensions to SQL and new indexes.

(ii) It must be efficient in order to handle the ever increasing complexity of SQL queries.

The goal of this thesis is to scrutinize the extensibility and efficiency of top-down optimizers. As a concrete framework for the study, I'll use a top-down optimizer that is based on *Cascades* (see Chapter 2 for details on Cascades). This is no limitation since optimizers based on Cascades are very typical for modern top-down optimizers such as Microsoft SQL Server [15] or Tandem DBMS [5]. Therefore the results derived within the framework of Cascades also apply to top-down optimizers in general. To make this point very clear, special sections will be dedicated to generalizing the results. Now here is the claim of this thesis:

[1] I use the terms SQL query optimizer, query optimizer and optimizer interchangeably.

> Top-down optimizers are less extensible than commonly claimed, but they can achieve a considerable degree of efficiency.

The defense of this claim takes the following shape: In the rest of this chapter, I introduce the terminology and review optimization basics. Chapter 2 introduces Cascades. Chapter 3 shades light on extensibility issues of Cascades. Chapter 4, finally, shows how the data structures of Cascades can be tuned for efficiency. Chapter 5 examines the benefits of pruning for Cascades. Efficiency gains in these two chapters will be measured by means of the TPC-D benchmark queries [32]. The thesis is concluded by an overview of current research issues involved in tuning Cascades for efficiency.

Year	1980	1985	1990	1995	2000
Max. no. of joins used in commercial applications	4	6	10	16	23

Note: The given numbers were obtained by averaging the estimates of David Maier (OGI) and Leonard Shapiro (PSU).

Table 1.1: Increasing query complexity over time

1.2 Terminology

This section introduces only the most basic terminology that is indispensable for the understanding of this thesis. More precisely, the following keywords are covered: property (logical and physical), operator (logical and physical), operator tree (logical and physical), plan, expression, logical equivalence.

First of all, let's introduce logical and physical operators: For that purpose, please note that a *relation* in its mathematical sense is a multi-set (sometimes called a "bag") of tuples. Internal to a computer, however, it is represented as a *stream of tuples*. A stream is the natural representation for a relation since the step by step processing of a computer puts the tuples of a relation into some order. This distinction between representation (i.e. stream) and represented object (i.e. relation) motivates the definition of logical and physical properties:

(i) The *logical properties of a stream* are the properties of the relation that is represented by the stream. Typical logical properties are the schema and cardinality of the represented relation.

(ii) The *physical properties of a stream* are all its properties that are not logical. These properties are stream-specific and do not apply to the represented relation. The most typical physical property of a stream is the order of its tuples.

Next, let's introduce logical and physical operators. The *logical operators* are the operators of the relational algebra. They are defined by a formal description of how they map input relations onto output relations. SELECT, PROJECT, and JOIN are examples of logical operators.

A *physical operator*, by contrast, is the (concrete) implementation of an operator. This term was introduced to clearly distinguish between the *mathematical concept* of a logical operator and the *computer related concept* of its implementation. Since physical operators come out of the world of bits and bytes, they map streams onto streams, not relations onto relations. Therefore, they might require the input streams to have certain physical properties. The MERGE_JOIN operator, for instance, requires its inputs to be sorted on the join attributes. Analogously, some physical operators guarantee certain physical properties for their output stream. *Sort enforcers* are an example of this species: They take attributes of the input stream as parameter, and guarantee the output stream to be sorted on these attributes. Other common physical operators are HASH_JOIN, RELATION_SCAN, and INDEXED_RELATION_SCAN [12, 21, 25].

I conclude this section with some further definitions: An *operator tree* is a tree of operators in which the children of an operator produce the operator's inputs. An operator tree is *logical* or *physical* if its operators are exclusively logical or physical [2], respectively. Physical operator trees are also called *plans*. The term *operator tree* without any addition is used to designate a tree that might contain both, *logical* and *physical* operators. An *expression* is the symbolic representation of an operator tree. Thus an expression also can be logical, physical, or "mixed". Finally, I say that two operator trees (or expressions) are *logically equivalent* if they produce identical output relations over any legal database. They need not, however, produce identical output streams! Only the relations represented by the streams must be identical.

1.3 Query Optimization Basics

This section gives an introduction to the field of query optimization. First, the reason why query optimizers are needed is reviewed. After that, a taxonomy of query optimizers is given. Two optimizer types in that taxonomy are particularly interesting to us: The bottom-up optimizers and the top-down optimizers. Therefore a description of these optimizer types will conclude the section.

To understand the need for query optimization, let's consider what happens when the user submits the following SQL-query to its database system:

SELECT *
FROM A, B, C
WHERE ...

[2] **Note:** Some authors define an operator tree to be logical or physical if its *top operator* is logical or physical, respectively. I don't follow their linguistic usage!

After parsing the query, a relational database system will represent it internally as the join of the tables A, B, and C. (Please note that the Cartesian product can be considered as a join with no join condition.) But there's a problem: Because the join operator ($\bowtie$) is commutative and associative, the join of A, B and C might be represented as

$$(A \bowtie B) \bowtie C, \tag{1.1}$$

or as

$$(C \bowtie A) \bowtie B, \tag{1.2}$$

or indeed in any of a number of other ways. If we make the simplifying assumption that there is only one physical join operator, then there are still 12 different expressions that represent $A \bowtie B \bowtie C$. Each of these expressions can be used to calculate the result of the query, but the computational costs might vary quite a bit from one expression to another. Thus, choosing the right expression as the basis for query evaluation is of utmost importance to the performance of a database system. The module that makes this choice is known as the query optimizer.

The query optimizer's job is highly non-trivial! Indeed, for the query $R_1 \bowtie \cdots \bowtie R_n$ – the R_i being relations – there are $(2n-2)!/(n-1)!$ different expressions that can be used to evaluate it. This quantity grows at an explosive, faster-than-exponential rate. To see where this quantity comes from, please note that the relations $R_1, \cdots, R_n$ can be permuted in $n!$ different ways. Furthermore, for each of these permutations, the number of possible ways to parenthesize it is known as the n-th Catalan Number [18], which is equal to

$$\frac{1}{n} \times \binom{2(n-1)}{(n-1)}. \tag{1.3}$$

Multiplying $n!$ and formula (1.3) yields the given result!

OK, now that it has been said why query optimizers are needed, let's look at how they work. Figure 1.1 shows my attempt to classify the different optimizers that have been built over the years.

Purely heuristic optimizers [10, 12] were built until about 1985. They use heuristics in a greedy way to transform the initial operator tree into a supposedly cheaper one. Common heuristics are to avoid Cartesian products and to push SELECT and PROJECT nodes down towards the leaves. Even though these optimizers generally work well, there are cases where their effectiveness is poor. This spurred research into *cost-based optimizers*. Optimizers of this type search the space of all possible operator trees, and estimate their computational costs. The cheapest operator tree is returned as the optimization result. Cost-based optimizers might use heuristics in order to prune the search space, but they never use heuristics exclusively! This optimizer type has got two important subclasses: The *bottom-up optimizers* and the *top-down optimizers*.

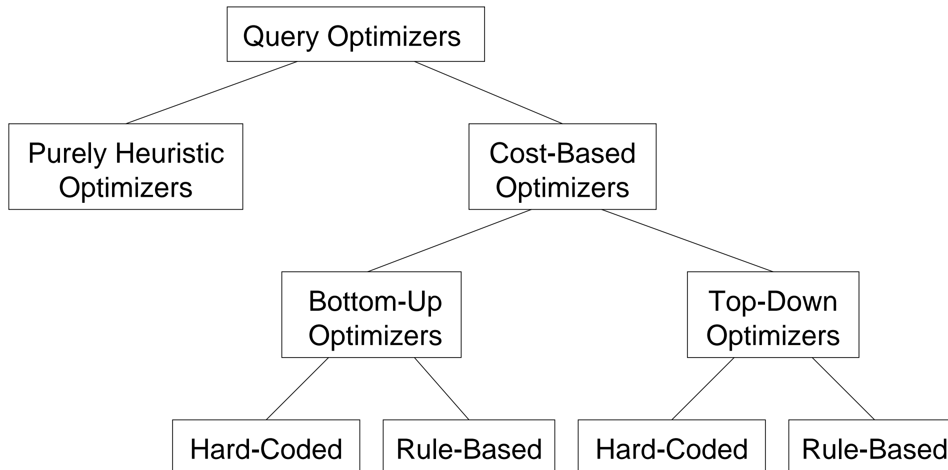


Figure 1.1: A taxonomy of query optimizers

The bottom-up optimizers [7, 31] are older than the top-down optimizers. They are based on *Bellman's principle of optimality* and *dynamic programming*. Bellman's principle of optimality is a very general approach to solving optimization problems. Applied to the field of query optimization, Bellman's principle of optimality says that – in order to find an optimal operator tree – it is sufficient to consider only trees whose subtrees are optimal. Dynamic programming, finally, is a programming technique which is commonly used to implement Bellman's principle of optimality. Figure 1.2 shows the bottom-up algorithm for join-order optimization. To see how the algorithm works, let's apply it on the query $A \bowtie B \bowtie C$: First the (trivial) problems A, B, and C are optimized, yielding $opt(\{A\})$, $opt(\{B\})$, and $opt(\{C\})$. Next these optima are used to optimize $A \bowtie B$, $A \bowtie C$, and $B \bowtie C$. That yields $opt(\{A, B\})$, $opt(\{A, C\})$, and $opt(\{B, C\})$, which in turn is used to calculate $opt(\{A, B, C\})$, the final result. Obviously, bottom-up optimizers perform a bottom-up, breadth-first search!

Bottom-up optimizers can be further subdivided into *hard-coded* and *rule-based* optimizers . To understand the difference, please notice that there has to be some rule-set (implicit or explicit) that dictates how the optimal solutions of small problems are fit together to yield the optimal solutions of bigger problems. If this rule-set is implicit, i.e. if it is smeared all over the optimizer code, then the optimizer is said to be hard-coded. If, in contrast, the rule-set is explicit, i.e. if it has been separated into a module of its own, then the optimizer is said to be rule-based. Obviously rule-based bottom-up optimizers are easier to extend and modify. System R [31] is an example of a hard-coded bottom-up optimizer; Starburst [19], on the other hand, is rule-based.

Bottom-up optimizers have one important problem: They are very difficult to extend because, in the general case, it is not clear how a query can be split into meaningful subproblems such that *(1)* the subproblems can be optimized independently, and *(2)* their solutions can be reassembled into the optimal solution of the initial query.

Input: $R_1 \bowtie \cdots \bowtie R_n$, the R_i being relations;
Output: $opt\,(\{R_1, \ldots, R_n\})$, the optimal operator tree for calculating $R_1 \bowtie \cdots \bowtie R_n$;
Algorithm: *// proceeds in bottom-up, breadth-first order*

```
for i := 1 to n do
    for each set S ⊆ {R1, ..., Rn}, |S| = i do
        // find and save the optimal operator tree
        // for joining the relations in S
        opt(S) := not defined;
        for each non-empty P ⊂ S do
            consider tree t := opt(P) ⋈ opt(S − P);
            if t is cheaper than opt(S) then
                opt(S) := t;
            endif;
        endfor;
    endfor;
endfor;
```

Figure 1.2: Bottom-up join-order optimization

Top-down optimizers were meant to improve upon bottom-up optimizers in this respect. Their basic idea is to use *rewriting rules* in a *depth-first search* manner in order to explore the search space. The emphasized terms are central, so let's have a closer look at them:

Rewriting rules: These are rules of the general form *before–pattern* $\longrightarrow$ *after–pattern*. Their meaning is that, whenever the pattern *before–pattern* is found in an operator tree, then it can be replaced by *after–pattern* to get a new, logically equivalent operator tree. Well-known rewriting rules are join-commutativity, i.e. $A \bowtie B \longrightarrow B \bowtie A$, and join-associativity, i.e. $(A \bowtie B) \bowtie C \longrightarrow A \bowtie (B \bowtie C)$.

Depth-first search: Depth-first search is the strategy that controls how the rewriting rules are applied in order to generate new operator trees out of old ones. Depth-first search traverses a given operator tree in post-order, first applying rewriting rules to the children of a node, then to the node itself.

Of course, all generated operator trees will be costed and the cheapest one will be returned as the optimization result. I don't give a pseudo-code description of top-down optimization here. That will be done in Chapter 2. One final point worth mentioning is that top-down optimizers, just like their bottom-up counterparts, can be *hard-coded* or *rule-based*. The distinctive feature is the same as for bottom-up optimizers, with the only difference that the *rule-set* of

a top-down optimizer is defined to be the set of all its rewriting rules. Volcano [17] is a hard-coded top-down optimizer. Modern top-down optimizers like Cascades [14] or Columbia [33, 37] are rule-based.

Chapter 2

Cascades

Cascades has been described at many different places [1, 13, 14, 21, 25, 34]. Still, I feel there's no thorough description of the underlying principles, data structures, and algorithms. The present chapter tries to fill this gab. To do so, it proceeds in four steps:

- Section 2.1 explains Cascades' philosophy to split an optimizer into two parts: The *model* that specifies the SQL-dialect of the optimizer, and the *search engine*, which constitutes a generic optimizer kernel that can be linked to any given model to build an optimizer for the specified SQL-dialect.
- Section 2.2 covers the data structures of Cascades.
- Section 2.3 describes the algorithm that Cascades uses to find the optimal operator tree, i.e. the operator tree that should be used as the basis of query evaluation.
- Section 2.4 contains an example that extensively uses the material covered in the preceding three sections.

2.1 Cascades' Architecture

In Section 1.3, Cascades has been classified as rule-based. To recap, "rule-based" means that the code which implements the rewriting rules is strictly separated from the code that uses them. This separation is achieved by using the object-paradigm [1] to implement the architecture shown in Figure 2.1. Next, starting at the top, the depicted architecture will be discussed.

[1] **Notice:** I'll use the terms

class – object – method – attribute

to discuss object-oriented concepts [23]. The term *class element* is used as a generic name for methods and attributes.

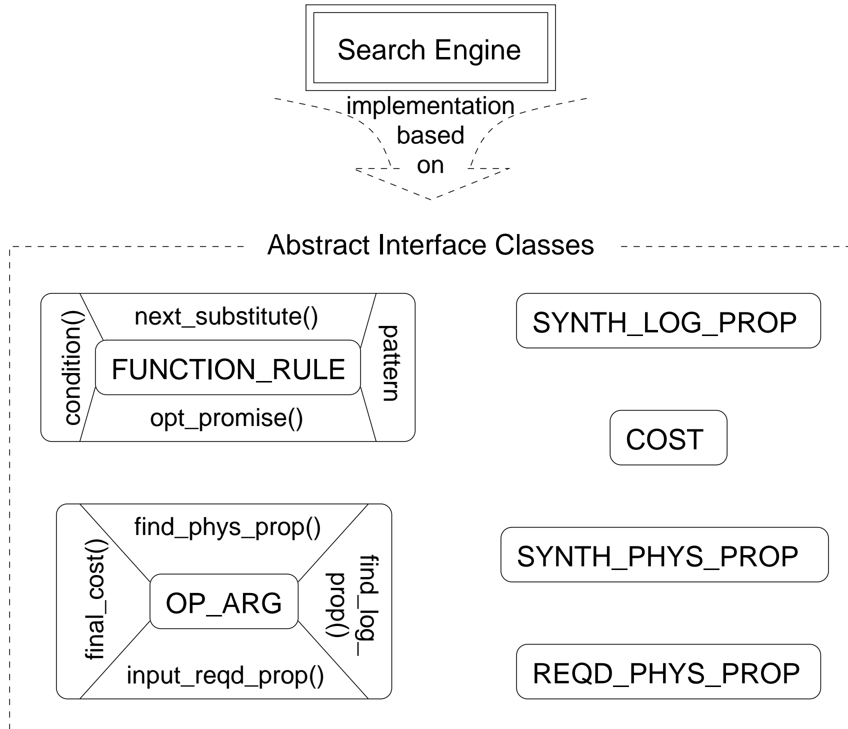


Figure 2.1: The architecture of Cascades

The *search engine* is a very complex piece of code. It's job is to generate, for a given query, all logically equivalent operator trees, cost them, and return the cheapest one. In order to fulfill this task, the search engine must know the operators it manipulates, as well as the rewriting rules that apply to them. This knowledge is provided through the *abstract interface classes*: By implementing (concrete) subclasses to these interface classes, all the necessary information is made available to the search engine. The set of these subclasses is called the *model*.

Here's an alternative way of looking at Cascades' architecture: The search engine is an "optimizer kernel" that can be used to optimize any SQL-dialect for any query execution system[2]. To build a full optimizer, the search engine must be supplemented with the description of a concrete SQL-dialect and its corresponding query execution system. This description is called the model. The model consists of a set of subclasses to Cascades' abstract interface classes. These subclasses must abide by the specification of the interface classes, so that the search engine can understand and use them. The search engine then exploits objects of the subclasses to learn all it needs to know about the given SQL-dialect and its query execution system. (Note: This way of using abstract interface classes is known as *polymorphism* [23].)

[2]The *query execution system* of a DBMS is the module that executes the output of the optimizer. Therefore, it determines the available physical operators, as well as their computational cost.

Class element	Overr.	Semantics
pattern	No	An attribute, used to store the before-pattern of a rule.
opt_promise()	Yes	A method returning an estimate for the usefulness of a rule. Used to apply the most promising rules first.
condition()	Yes	To apply a rule to an operator tree t, the rule's pattern must match a subtree of t, *and* the method condition() must evaluate to true.
next_substitute()	Yes	This method calculates the after-pattern of a rule. This is more flexible than fixing a priori an after-pattern.

Table 2.1: Description of the class FUNCTION_RULE

The tricky part of this architecture is that, whenever the SQL-dialect or the query execution system changes, all that has to be modified is the model; the search engine stays the same. But the model is easy to modify since the abstract interface classes are very simple. Therefore optimizers whose implementation is based on Cascades (also called *Cascades-based optimizers*) have been claimed to be extensible: Just modify the model to reflect the new requirements and you're done! Chapter 3 will examine how true this claim is.

Now let's have a closer look at the interface classes shown in Figure 2.1. The first thing that needs to be said is that the depicted interface is a simplification in two respects: Firstly, Cascades' interface comprises more classes. Secondly, for those classes in the figure, not all the class elements are shown. Nevertheless, the figure contains the most relevant information about Cascades' interface classes. The remainder of this section will be dedicated to describing each of the interface classes in Figure 2.1.

Each rewriting rule r is an object of a class R. R, for its part, is a subclass of the class FUNCTION_RULE. Table 2.1 shows for each class element of FUNCTION_RULE whether it must be overridden in R or not. The semantics column of Table 2.1 describes the usage of the class elements. For class elements that are overridden by R, *it is the model programmer's responsibility to abide by these semantic specifications!* Here's an example to illustrate the new concepts:

Example 2.1 *Consider the join-associativity rule. It's* pattern *is the expression* $(A \bowtie_P B) \bowtie_Q C$, *$P$, and Q being predicates and A, B, and C being so-called* leaves. *Leaves are variables that can be bound to any arbitrary operator tree. The rule's after-pattern is* not $A \bowtie_P (B \bowtie_Q C)$ *since when moving the parentheses, the predicates P and Q have to be revised, too. This is done by the method* next_substitute() *which calculates a tailor-made after-pattern for each binding of the before-pattern* pattern. *The method* opt_promise() *can be*

implemented as the model programmer sees fit. The method condition()*, finally, always returns true. An example of a rule with a non-trivial* condition() *is the rule that commutes* PROJECT *and* JOIN*:* $A \bowtie_P \pi(B) \longrightarrow \pi(A \bowtie_P B)$*. This rule must not be applied if the project node calculates new, additional attributes that are used in the join condition* P*. Thus,* condition() *only returns true if the schema of* B *contains all the attributes needed by the join condition* P*.* □

Next, let's consider the classes SYNTH_LOG_PROP, COST, SYNTH_PHYS_PROP and REQD_PHYS_PROP. To this end, the difference between *synthesized properties* and *required properties* must be defined. The *synthesized properties* of a stream are all its properties that Cascades can derive ("synthesize") from the operator tree and the database that produced the stream. Thus, synthesized properties are derived from what Cascades knows. Required properties, in contrast, describe what Cascades is heading for: A set of required properties tells Cascades to look for an operator tree whose output stream possesses these properties. Logical properties are always synthesized. Physical properties might be synthesized or required.

Now let's go back to the interface classes SYNTH_LOG_PROP, COST, SYNTH_PHYS_PROP and REQD_PHYS_PROP. Cascades uses these classes (almost) without making any use of their internal structure (i.e. their class elements). Therefore the model programmer has a lot of freedom when subclassing these interface classes. The only restriction is that each subclass must define what is indicated by the name of its parent class, i.e. it must define either (synthesized) logical properties, computational cost, synthesized physical properties or required physical properties.

Example 2.2 *Required physical properties arise, for example, in the case of the* MERGE_JOIN *operator. This operator requires its input streams to be sorted on the join attributes. To establish this sorting, the sorting requirement – which is a required physical property – is passed to the input trees of the* MERGE_JOIN *operator. Later, when Cascades optimizes the input trees, the sorting requirement conveys the following message to Cascades: The input trees must be optimized so as to minimize their computational cost,* and *to guarantee their output streams satisfy the sorting requirement.* □

Obviously, required physical properties propagate top-down (from the MERGE_JOIN operator to its inputs). We'll see later that synthesized properties (logical or physical) propagate bottom-up (from the leaves of an operator tree to its root).

The last interface class to be discussed is the class OP_ARG. Each operator is an object of some subclass of the class OP_ARG. The class OP_ARG possesses two types of methods: The methods final_cost(), find_phys_prop(), and input_reqd_prop(), which only apply to physical operators, and the method find_log_prop(), which only applies to logical operators. Let's have a closer look at each of these methods.

– Be o an operator. Feed the logical properties of o's input streams into the method find_log_prop() and the logical properties of its output stream are returned as a result. For a whole operator tree, the logical properties of its output stream can be derived by using the method find_log_prop() in a bottom-up fashion: Starting at the leaves, calculate the logical properties of each operator's output stream and pass this information upwards in the tree. Cascades has implemented this technique. (Now it becomes clear why Cascades needs to know nothing about the internal structure of the class that implements logical properties: Cascades just moves the objects of this class to and from the find_log_prop()-methods. All the real work that requires an understanding of what logical properties really are is done inside these methods.)

– The method final_cost() calculates the computational cost of a physical operator tree. Please notice that no cost is defined for an operator tree that contains logical operators. This is because logical operators are mathematical entities. There is no notion of executing logical operators; thus no cost can be associated with them.

– Be o a physical operator. Feed the logical and physical properties of o's input streams into the method find_phys_prop() and you get the synthesized physical properties of o's output stream as a result. For a whole operator tree, the physical properties of its output stream can be derived by the same bottom-up procedure that has been described for find_log_prop().

– Be o a physical operator, for example the MERGE_JOIN operator of example 2.2. Somehow Cascades must learn about the physical properties that the operator o requires from its input streams. This is done by invoking the method input_reqd_prop() on the operator o. The method's return value are just the required physical properties.

As might be expected, objects of the classes SYNTH_LOG_PROP, COST, SYNTH_PHYS_PROP and REQD_PHYS_PROP are used to pass logical properties, costs and physical properties to and from the above methods.

2.2 Cascades' Data Structures

This section covers the five data structures that Cascades' search engine uses: The *memo* (class MEMO), the *group* (class GROUP), the *multi-expression* (class EXPR_LIST), the *context* (class CONTEXT) and the *winner's circle* (class WINNER). The class names of these data structures are given for completeness only. No further implementation related aspects will be covered in this section.

The ER-diagram [12] in Figure 2.2 outlines the relationship between the data structures. Subsection 2.2.1 describes groups, multi-expressions, and the memo; Subsection 2.2.2 is dedicated to the context, and the winner's circle.

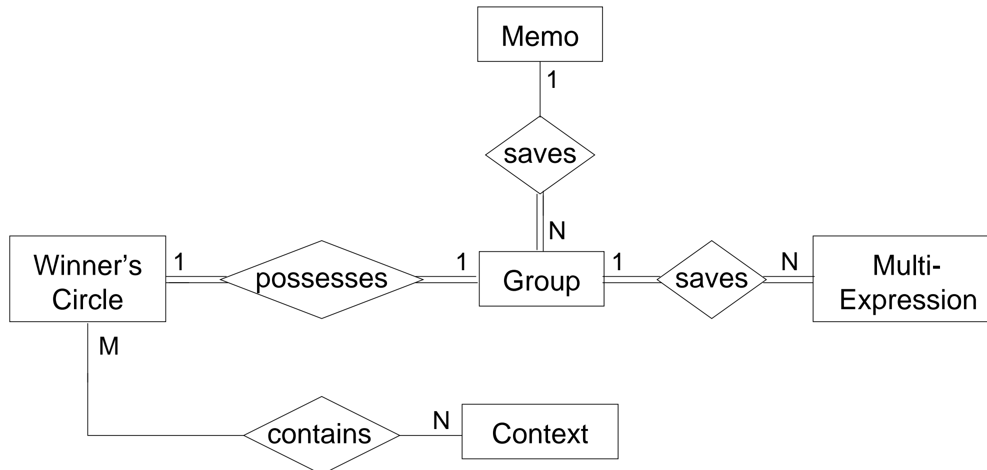


Figure 2.2: The interaction between Cascades' data structures

2.2.1 Memo, Groups and Multi-Expressions

The task Cascades performs can be described as follows: Given an operator tree, find a logically equivalent operator tree that is computationally cheaper to evaluate. From this description follows that operator trees are of central importance to Cascades. Therefore Cascades must have a way to represent them internally. The naive approach would be to store whole operator trees. This is demonstrated in Figure 2.3. As shown, a completely new operator tree is created to store the result of each rule application.

The problem with the approach of Figure 2.3 is that it is incredibly wasteful. Indeed, the two operator trees of the post rule-application memory only differ in the shaded sub-trees. The structure ($\star \bowtie C$), however, is identical for both operator trees – but still it is saved twice! Fortunately, it is possible to eliminate this inefficiency. All that has to be done is to build groups of logically equivalent (sub-)trees and to make these groups the input of operators. Figure 2.4 shows what the post rule-application memory looks like now. Formalizing this approach yields the following mutually recursive definition of *groups* and *multi-expressions*:

- A *multi-expression* is an operator having groups as inputs.
- A *group* is a set of logically equivalent multi-expressions.

The *memo*, finally, is the data structure that saves all the groups that Cascades creates in the course of its calculation. Whenever a new group is created, it is inserted into the memo.

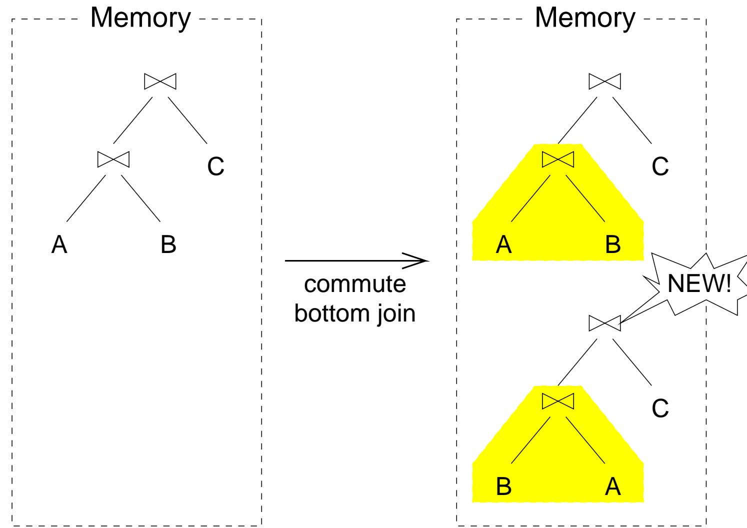


Figure 2.3: The naive approach to storing operator trees

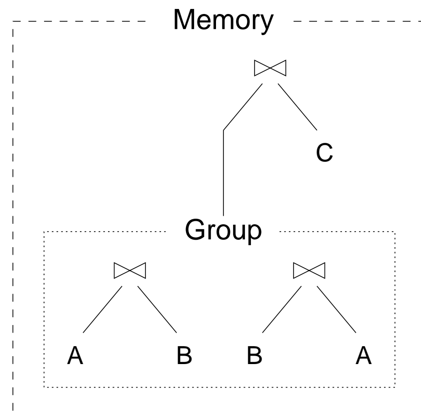


Figure 2.4: A more efficient way of storing operator trees

The following notation has become established for groups:

– The group $[R_1, \ldots, R_n]$ – the R_i being relations – is defined inductively:

  (i) $[R_i]$ designates the group that just contains the multi-expression GET_{R_i}. GET_{R_i} is the logical operator that returns the relation R_i. Its arity is zero.

  (ii) $[R_{i_1}, \cdots, R_{i_k}]$, $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$, designates the group that contains (inter alia) the multi-expression $[R_{i_1}] \bowtie [R_{i_2}, \cdots, R_{i_k}]$.

– Boxes are used to graphically represent groups. The name of the group is indicated in the upper-left corner of the box. The main part of the box contains the multi-expressions of the group.

Example 2.3 *To internally represent the expression* $((A \bowtie B) \bowtie (C \bowtie D))$, *the groups* $[A]$, $[B]$, $[C]$, $[D]$, $[A,B]$, $[C,D]$, *and* $[A,B,C,D]$ *must be created. These groups are initialized in the following way:*

- $[A]$, $[B]$, $[C]$, *and* $[D]$ *are initialized to contain* GET_A, GET_B, GET_C, *and* GET_D, *respectively;*
- $[A,B]$ *is initialized to contain* $[A] \bowtie [B]$;
- $[C,D]$ *is initialized to contain* $[C] \bowtie [D]$;
- $[A,B,C,D]$ *is initialized to contain* $[A,B] \bowtie [C,D]$.

Figure 2.5 shows the memo when the initialization is done. □

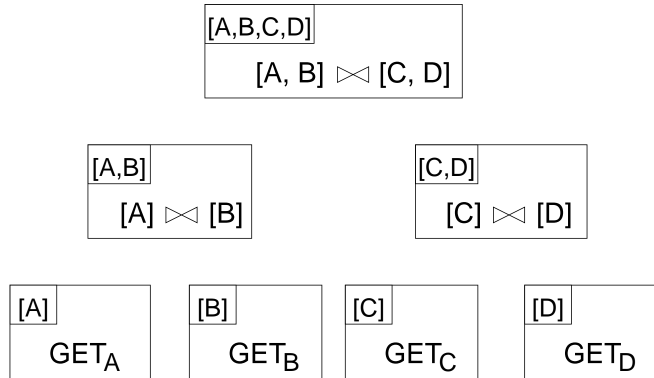


Figure 2.5: The memo saving the expression $((A \bowtie B) \bowtie (C \bowtie D))$

It is important **never** to forget that groups and multi-expressions just **represent** expressions (or operator trees, as you like). For a group G, the set $expr(G)$ of expressions represented by (or "contained in") G can be found as follows:

$$expr(G) = \{\, op(\, e_1, \ldots, e_n\,) \mid \text{It exists an operator } op \in \{\bowtie, \sigma, \Pi, \ldots\} \text{ and groups } G_1, \ldots, G_n, \text{ such that the multi-expression } op(\, G_1, \ldots, G_n\,) \text{ is in } G \text{ and } \forall i : e_i \in expr(G_i)\}.$$

The expressions represented by (or "contained in") a multi-expression M can be found in a similar way.

2.2.2 Context and Winner's Circle

Example 2.2 demonstrated that there are circumstances when Cascades not only searches any optimal operator tree, but an operator tree which – aside from being optimal – also has particular properties. Under these circumstances, the optimization process is subject to certain side conditions such as producing

a result that is cheaper than 100 cost units. The *context* is the data structure that has been devised to save and handle such side conditions. Its purpose is to specify an optimization goal consisting of two components:

- the maximum cost the optimization result must not exceed, and
- the physical properties the output stream of the optimization result is required to have.

The first of the above two components is widely known as the *upper cost bound of the context.* This component plays a central role in group pruning, which will be discussed in Chapter 5.

It occasionally happens that a group is repeatedly optimized for the same context. In this case, it is beneficial to save the optimization result the first time it is calculated; every other time, it will be looked up. Cascades uses the *winner's circle* to implement this idea: The winner's circle of a group is a list of (context, optimization result)-pairs, each containing the optimization result for the indicated context. The optimization results stored in the winner's circle are called *winners.*

2.3 Algorithmic Aspects of Cascades

In this section I'll explain how Cascades' search engine works. The Subsections 2.3.1 and 2.3.2 discuss the algorithmic problems that arise in manipulating the memo. Subsection 2.3.3 describes the optimization algorithm itself. But first, here is a little terminological wrap-up:

- A *multi-expression* is *logical* (*physical*) if its operator is logical (physical).

- Rewriting rules are commonly subdivided into *transformation rules* and *implementation rules.* The distinction is based on whether the top operator of the rule's after-pattern is logical or physical: If it is logical, then the rule is a transformation rule, otherwise it's an implementation rule. (Note: Aside from the top operator of the after-pattern, Cascades requires all other operators in the before- and after-pattern to be logical.)
- Two *multi-expressions* M_1 and M_2 are *logically equivalent* iff the expressions represented by M_1 are logically equivalent to the expressions represented by M_2.

2.3.1 The Binding Algorithm

Groups and multi-expressions have been devised because they allow to represent expressions in a very compact and memory efficient way. The inconvenience, however, is that groups and multi-expressions do not lend themselves very well

to being manipulated by rewriting rules. The problem is that rewriting rules transform expressions, not groups or multi-expressions! Thus, in order to apply a rewriting rule, it is necessary to *(1) extract* an expression from the memo, to *(2) transform* it according to the rule, and to *(3) reinsert* the result into the memo. In this subsection, I'll describe how the extracting is done; the reinserting is described in Subsection 2.3.2. Please note, that the terms "pattern" and "before-pattern of some rule" are used synonymously throughout this subsection.

Let's start off with the extracting issue. Firstly, it should be stressed that the above picture of extracting whole expressions is an over-simplification. Indeed, Cascades' algorithm implements *lazy extraction*: Just as much of an expression is extracted from the memo as to match the rule's before-pattern. The extracted fragment could be called a *generalized multi-expression*, since it is a multi-expression whose operator is the concatenation of several relational operators. For example, let G_1, G_2, and G_3 stand for groups. Then $(G_1 \bowtie G_2) \bowtie G_3$ is a generalized multi-expression, and $(\star \bowtie \star) \bowtie \star$ is its operator. The association of a pattern to a matching generalized multi-expression is called a *binding of the pattern*. "To find a binding" and "to find a matching generalized multi-expression" are therefore synonyms.

Cascades uses *bindery objects* (class BINDERY) to find all possible bindings of a pattern. Each bindery object has got two important attributes: The pattern and the domain. The first attribute specifies the pattern which the object tries to find bindings for. The second one specifies the domain that is searched for bindings. The domain can be a single multi-expression or a whole group. In both cases, the bindery object only considers those generalized multi-expressions that can be extracted from the given domain.

Bindery objects dispose of two important methods: The method advance() and the method extract_expr(). The method advance() looks for the next binding between the pattern and a generalized multi-expression of the domain. It returns *true* or *false* to indicate success or failure, respectively. In the case of success, the method extract_expr() can be used to extract the generalized multi-expression that pattern has been bound to.

Let's see how bindery objects work. For that purpose consider a bindery object O whose pattern is p. Roughly speaking, O creates and supervises other bindery objects, one for each input of pattern p. (This process of creating new bindery objects continues recursively.) When the method advance() is invoked on O, O first tries to create a new binding by systematically invoking advance() on the bindery objects it supervises. If that should fail then O tries to find a binding with the next multi-expression of its domain. The Figures 2.6 and 2.7 show the algorithm.

2.3.2 Inserting into the Memo

Now let's put the binding issue into context: To apply a rewriting rule, it's before-pattern is first bound to some generalized multi-expression A. Then, A

```
BINDERY::BINDERY(domain, pattern)
// The constructor: Initializes the object's attributes.
{   this→domain := domain;   this→pattern := pattern;
    this→state := "start";       this→b[] := empty-array;
    this→current := the first multi-expression in domain;
}; // end BINDERY::BINDERY

boolean BINDERY::advance()
{   loop
        switch(state) {
        case "start":
            if pattern = leave then
                state := "finished";    return(true);
            endif;
            // Be 'pattern' = op(p_1, ..., p_n) and 'current' = \overline{op}(G_1,
            // ..., G_m), with op, \overline{op} ∈ {⋈, σ, Π, ...}, and
            // the p_i (G_j) being patterns (groups), respectively.
            if \overline{op} ≠ op then
                state := "next_in_domain";    break;
            endif;
            Try to create bindings for all the sub-patterns p_i by
            allocating new bindery objects b[i] := BINDERY(G_i, p_i),
            i = 1, ..., n, and invoking advance() on each of them;
            if all invocations of advance() returned true then
                // All bindings have been successfully created!
                state := "valid_binding";    return(true);
            endif;
            state := "next_in_domain";    break;
        case "valid_binding":
            // Advance the input bindings b[i] in right-to-left order:
            i := n;
            while (i > 0) ∧ ¬(b[i]→advance()) do  dec(i);  endwhile;
            if i > 0 then
                for j := i + 1 to n do
                    b[j] := BINDERY(G_j, p_j);    b[j]→advance();
                endfor;
                return(true);
            endif;
            state := "next_in_domain";    break;
        case "next_in_domain":
            if current is the last multi-expression in domain then
                state := "finished";
            else
                current := next multi-expression in domain;
```

Figure 2.6: The class BINDERY

```
                    state := "start";
                endif;
                break;
            case "finished":
                return(false);
            }   // End Switch
        endloop;
    }; // end BINDERY::advance
```

Figure 2.7: The class BINDERY (continued)

is extracted from the memo and the rewriting rule is applied on A. This yields another generalized multi-expression, say B. Next, B is inserted into the memo. For that purpose, B is viewed as a tree t whose leaves are groups and whose inner nodes are relational operators. Inserting B into the memo creates a multi-expression for each relational operator in t, see Figure 2.8. The multi-expression created for the root of t is called the *primary multi-expression*; all the other multi-expressions are called *secondary multi-expressions*. The primary multi-expression is inserted into the same group G, that A has been extracted from. Secondary multi-expressions are either discarded right away (if they already exist) or a new group is set up to store them. Please notice, that the primary multi-expression might also happen to exist in some group $\overline{G}$ of the memo. If $G \neq \overline{G}$, then the multi-expressions in G are logically equivalent to those in $\overline{G}$. Therefore the two groups are combined into one group. That's what the procedure merge() does.

It is very important to be precise about what it means for two multi-expressions to be *identical* (see function search_memo()): Two multi-expressions are *identical*, iff their operators satisfy the following three conditions: *(1)* the operators must be of the same type (e.g. $\bowtie$ or σ or $\ldots$), and *(2)* they must agree on their parameters (e.g. join condition, select condition, etc.), and *(3)* they must take the very same groups as inputs.

Example 2.4 *(Continuation of Example 2.3.) Consider what happens when the associativity rule* $(X \bowtie Y) \bowtie Z \longrightarrow X \bowtie (Y \bowtie Z)$ *is applied to the memo of Figure 2.5. The only binding of the rule's before-pattern is to the generalized multi-expression* $([A] \bowtie [B]) \bowtie [C, D]$*. The result of the rule application is* $[A] \bowtie ([B] \bowtie [C, D])$*; to insert the result into the memo, the following steps are taken:*

- *Create the new group* $[B, C, D]$ *and store the secondary multi-expression* $[B] \bowtie [C, D]$ *in it;*
- *Create the primary multi-expression* $[A] \bowtie [B, C, D]$ *and add it to the group* $[A, B, C, D]$*.*

□

```
// Note: As in C++, the &-symbol is used below to declare
// a reference to an object.

GROUP &search_memo(mult_ex)
{   for each multi-expression M stored in the memo do
        if M and mult_ex are identical (see text above)
            then Return a reference to the group M belongs to;  endif;
    endfor;
    return(NULL);
};  // search_memo

GROUP &insert(t, top, group)
// 't': a generalized multi-expression, viewed as a tree;
// 'top': is false iff the current incarnation of insert() resulted
//     from a recursive self-invocation;
// 'group': if 'top'=false, then 'group'=undefined. Otherwise, 'group'
//     dictates where to put the multi-expr. spawned by the root of 't';
{   // Be t = op(t_1, ..., t_n), op ∈ {⋈, σ, Π, ...},
    // and the t_i being the sub-trees of t. Then:
    for i := 1 to n do
        if t_i is a group then G_i := t_i;
                          else G_i := insert(t_i, false, undefined);
        endif;
    endfor;
    if op is physical then
        // In Cascades, only the top-operator of a rule's after-pattern can
        // be physical. Thus: ('op' is physical) ⇒ ('top' = true).
        Insert the multi-expression op(G_1, ..., G_n) into group;
        return(group);
    else   // op(G_1, ..., G_n) is a logical multi-expression!
        other_group := search_memo(op(G_1, ..., G_n));
        if other_group ≠ NULL then
            if top ∧ (group ≠ other_group)
                then merge(group, other_group);  endif;
            if top
                then return(NULL);
                else return(other_group);  endif;
        else   // Truly a new logical multi-expression.
            if top then
                Insert the multi-expression op(G_1, ..., G_n) into group;
                return(group);
            else
                Create a new group containing exclusively the multi-
                expression op(G_1, ..., G_n) and return this group;
        endif;
    endif;
}; // insert
```

Figure 2.8: insert() inserts generalized multi-expressions into the memo.

It is significant that the function insert() guarantees the multi-expressions of a group to be logically equivalent. However, logically equivalent multi-expressions need not be put into the same group. This is because Cascades puts two multi-expressions into the same group if they are *syntactically equivalent*, i.e. if there's a concatenation of rewriting rules that transforms one multi-expression into the other. Syntactical equivalence does imply semantical equivalence, but the inverse need not be true. Here's an example: Be G a group and $\{A, B, C\}$ its schema. Furthermore, suppose the functional dependency $A \rightarrow B$ holds. Then the multi-expressions $\pi_{\{A,B,C\}}(G)$ and $\pi_{\{A,B\}}(G) \bowtie_{A=A} \pi_{\{A,C\}}(G)$ are logically equivalent [20]. However, if the rule-set cannot handle functional dependencies, then the two multi-expressions are *not* syntactically equivalent!

The question of when syntactical and logical equivalence are identical does not seem to have a simple answer. Obviously, the employed rule-set would have to satisfy some kind of completeness property, similarly to the completeness property of predicate calculus [29].

2.3.3 The Optimization Algorithm

Cascades' optimization algorithm is task-driven: In a top-down, recursive fashion, the optimization task is decomposed into easier tasks, and their solutions are recombined to solve the original optimization task. It would have been possible to implement this algorithm as a set of mutually recursive procedures, one procedure for each optimization task. However, this approach was rejected for the sake of extensibility [14]. Instead, *task objects* are used in Cascades. The state of a task object describes an optimization task that has to be done. Furthermore, each task object disposes of the method perform(). By invoking this method, the object solves the task that is described by its state.

An object of the class TASK_LIST is used to manage the task objects. In the current implementation, TASK_LIST simply is a stack of task objects, but other data structures are conceivable. Figure 2.9 outlines Cascades' optimization algorithm.

```
void optimize(query)
{   task_list := new TASK_LIST();
    Copy the operator tree query into the memo;
    task_list→push(the task of optimizing the top-group of query);
    while not( task_list→empty() ) do
        next_task := task_list→pop();
        next_task→perform();
        // Note: Task objects create and schedule other task objects!
    endwhile;
}; // end optimize()
```

Figure 2.9: The main loop of Cascades' optimization algorithm

It's time to look more closely at the task objects. Task objects are of one of four classes[3]: OPT_GROUP, OPT_EXPR, APPLY_RULE or OPT_INPUTS. The class attributes are context and group for OPT_GROUP – context and log_mult_ex for OPT_EXPR – context, rule, and log_mult_ex for APPLY_RULE – context and phys_mult_ex for OPT_INPUTS. Note that each of these classes has got the attribute context. This reflects the fact that a task object is always associated with a context. The constructors of the above four classes, finally, only initialize their respective class attributes. The method perform(), however, is specific to each class, and is described next:

Be T of class OPT_GROUP: Then T→perform() will augment the group T→group by all syntactically equivalent multi-expressions.

Be T of class OPT_EXPR: Then T→perform() will schedule all possible rules for application on the logical multi-expression T→log_mult_ex.

Be T of class APPLY_RULE: Then T→perform() will fire the rule T→rule for every possible binding between T→log_mult_ex and the before-pattern of T→rule.

Be T of class OPT_INPUTS: Then T→perform() will trigger the optimization of the input groups of the physical multi-expression T→phys_mult_ex. Furthermore, the cost and the synthesized physical properties of T→phys_mult_ex will be calculated.

Before proceeding to the pseudo-code shown below, please notice that it is wasteful to apply the same rule more than once on a given multi-expression. Therefore, each multi-expression possesses a *rule mask* in order to *enable* and *disable* rules. For instance, be M a multi-expression, and be r a rule. Then r can be applied on M if and only if M has got the rule r enabled. Upon applying the rule r, it becomes disabled for application on the multi-expression M. Thus, the rule r won't be applied on M but once. Cascades' implementation of this idea has the following particularities:

- Only logical multi-expressions possess a rule mask, since no rule is ever applied on a physical multi-expression.
- The rule mask can only disable transformation rules. Implementation rules are always enabled!
- Whenever a secondary multi-expression is created, all rules are enabled for it.
- Whenever a primary multi-expression is created, its rule mask is set by the rule that created it. (This feature is needed to build *duplicate-free rule-sets*, see Section 3.3.)

The Figures 2.10, and 2.11, finally, contain the pseudo-code of the *perform()*-methods.

[3]I assume that the promise of an implementation rule is always greater than the promise of a transformation rule. This assumption frees me from the need to consider the task classes EXPLORE_GROUP and EXPLORE_EXPR [14, 21, 25].

```
void OPT_GROUP::perform()
// Remember: 'context' and 'group' are class attributes!
{   if there's already a winner for context then return; endif;
    for each logical multi-expression M in group do
        task_list→push( new OPT_EXPR(context, M) );
    endfor;
}; // OPT_GROUP::perform()

void OPT_EXPR::perform()
// Remember: 'context' and 'log_mult_ex' are class attributes!
{   for each rule r in the rule-set do
        if r is disabled for application on log_mult_ex
            then continue; endif;
        if (the top-operators of log_mult_ex and r are of the same type
          (e.g. ⋈, σ, Π, etc.)) ⋀ (r →opt_promise(context) > 0) then
            Store the rule r and its promise value;
        endif;
    endfor;
    Sort the stored rules in ascending order of promise;
    for each rule r in ascending order of promise do
        task_list→push( new APPLY_RULE(context, r, log_mult_ex) );
    endfor;
}; // OPT_EXPR::perform()

void APPLY_RULE::perform()
// Remember: 'context', 'rule' and 'log_mult_ex' are class attributes!
{   binding := new BINDERY(log_mult_ex, before-pattern of rule);
    while binding→advance() do
        before := binding→extract_expr();
        if rule→condition(before, context) is not satisfied
            then continue; endif;
        after := rule→next_substitute(before);
        if insert(after, true, group of log_mult_ex) = NULL then
            continue;   // 'after' was a duplicate! See Figure 2.8.
        endif;
        new_mult_ex := the primary multi-expression just added to the memo;
        if new_mult_ex is logical then   // 'rule' is a transformation rule
            Set the rule mask of new_mult_ex as required by rule;
            task_list→push( new OPT_EXPR(context, new_mult_ex) );
        elsif new_mult_ex is physical then   // 'rule' is an impl. rule
            task_list→push( new OPT_INPUTS(context, new_mult_ex) );
        endif;
    endwhile;
    if rule is a transformation rule
        then   Disable rule for application on log_mult_ex;   endif;
}; // APPLY_RULE::perform()
```

Figure 2.10: OPT_GROUP–, OPT_EXPR–, APPLY_RULE::perform()

```
CONTEXT adapt_context(context, phys_op, i);
// Returns the context that must be used to optimize the 'i'-th
// input-group of 'phys_op'.

void OPT_INPUTS::perform()
// Remember: 'context' and 'phys_mult_ex' are class attributes!
// Note: 'i' is a class attribute that is set to -1 by the constructor.
{    phys_op := operator of phys_mult_ex;
     if i > 0 then
          InpCtxt := adapt_context(context, phys_op, i);
          grp := the i-th input-group of phys_op;
          // 'grp' has just been optimized for context 'InpCtxt'!
          Search grp for a physical multi-expression M satisfying InpCtxt;
          if successful then  Add M to the winner's circle of group grp;
                         else  return;  endif;
     endif;
     inc(i);
     if i < (arity of phys_op) then
          task_list→push( this );    // Re-schedule the present task!
          task_list→push( new OPT_GROUP(
               adapt_context(context, phys_op, i),
               the i-th input-group of phys_op) );
     else
          Calculate the cost and synthesized physical properties of
          phys_mult_ex;
     endif;
}; // OPT_INPUTS::perform()
```

Figure 2.11: OPT_INPUTS::perform()

2.4 An Example

In the present section I pursue two goals: On the one hand I want to give an example for the mode of operation of Cascades. On the other hand I want to prepare Chapter 3, which has got the following two particularities:

A) It uses the following duplicate-free rule-set for join-order optimization [30]:

R1: **Commutativity** $A \bowtie_0 B \longrightarrow B \bowtie_1 A$.
Disable rules *R1*, *R2*, *R3*, *R4* for application on the new operator $\bowtie_1$.

R2: **Right Associativity** $(A \bowtie_0 B) \bowtie_1 C \longrightarrow A \bowtie_2 (B \bowtie_3 C)$.
Disable rules *R2*, *R3*, *R4* for application on the new operator $\bowtie_2$.

R3: **Left Associativity** $A \bowtie_0 (B \bowtie_1 C) \longrightarrow (A \bowtie_2 B) \bowtie_3 C$.
Disable rules *R2*, *R3*, *R4* for application on the new operator $\bowtie_3$.

R4: **Exchange** $(A \bowtie_0 B) \bowtie_1 (C \bowtie_2 D) \longrightarrow (A \bowtie_3 C) \bowtie_4 (B \bowtie_5 D)$.
Disable rules *R1*, *R2*, *R3*, *R4* for application on the new operator $\bowtie_4$.

B) Its focus is exclusively on logical multi-expressions. Physical multi-expressions, by contrast, play no role in Chapter 3.

To promote both goals, let's optimize the query $(A \bowtie B) \bowtie C$ using the rule-set *R1*, …, *R4*. In order to completely focus on logical multi-expressions the following lines have to be inserted at the very end of OPT_EXPR::perform():

```
for each input-group g of log_mult_ex do
    task_list→push( new OPT_GROUP(context, g) );
endfor;
```

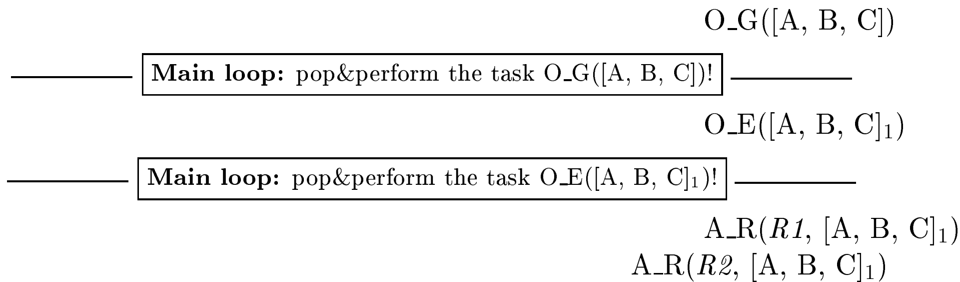
Without these lines the algorithm wouldn't be correct for a rule-set that contains no implementation rules! The purpose of the additional code is to optimize the inputs of a multi-expression before applying any transformation rules to the multi-expression itself.

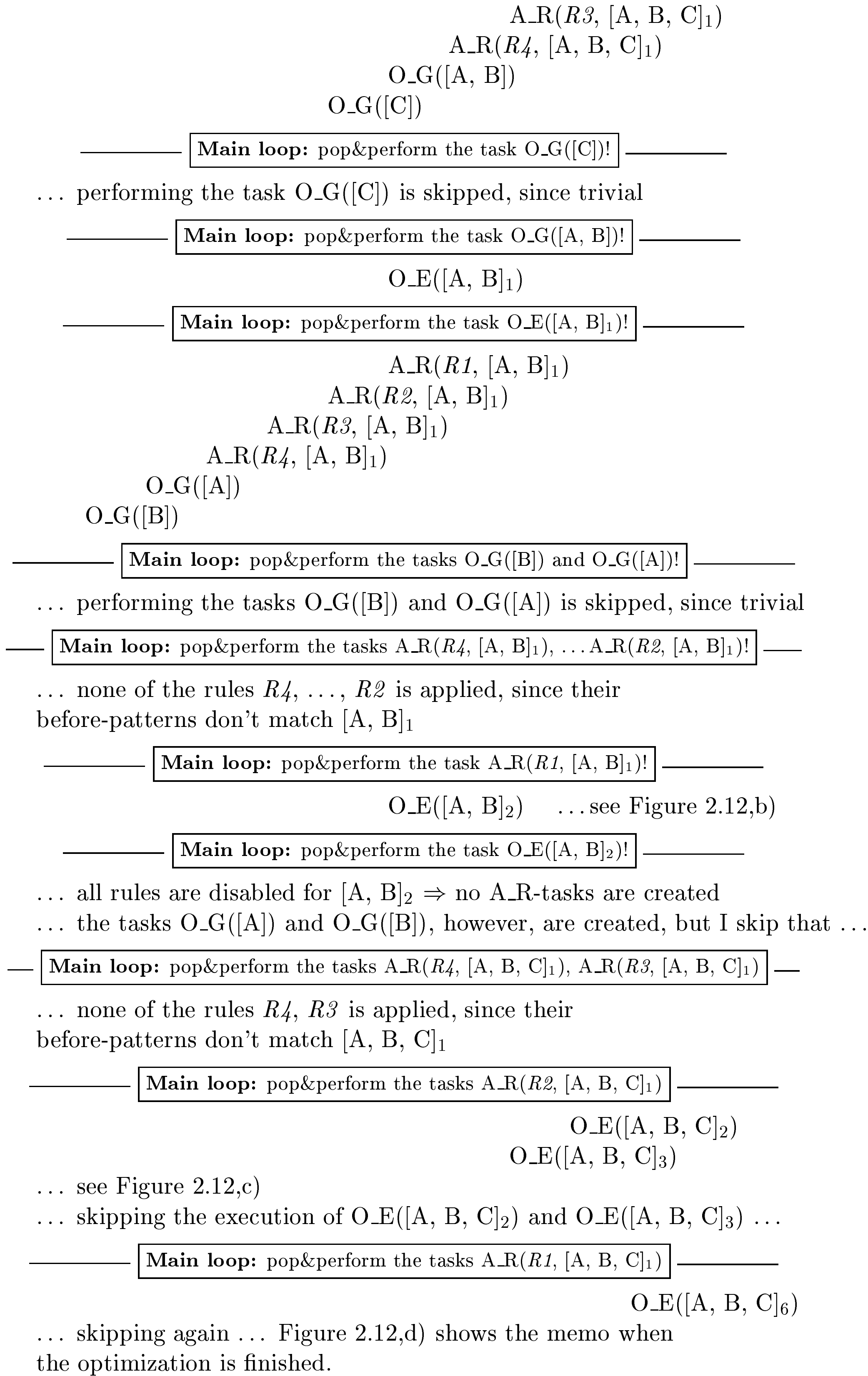
To recap, I'll show in the present section how the Cascades algorithm extended by the above three lines of code optimizes the query $(A \bowtie B) \bowtie C$. The employed rules are *R1*, …, *R4*. The notation has been adapted to this example:

- O_G(*gr*) (O_E(*mex*)) denotes the task that optimizes the group *gr* (the multi-expression *mex*). A_R(*r*, *mex*) is the task that applies rule *r* on multi-expression *mex*.
- The box-notation is used to graphically represent groups. The multi-expressions of a box are sequentially numbered from top to bottom. G_k is written to designate the k-th multi-expression of the group G. This naming scheme is unambiguous since new multi-expressions will always be added to the end of a group.

Figure 2.12,a) shows the memo before the optimization begins. At that moment, the task list only contains O_G([A, B, C]). Here's what happens next:

TOP of task-list BOTTOM of task-list





A_R(*R3*, [A, B, C]$_1$)
A_R(*R4*, [A, B, C]$_1$)
O_G([A, B])
O_G([C])

Main loop: pop&perform the task O_G([C])!

… performing the task O_G([C]) is skipped, since trivial

Main loop: pop&perform the task O_G([A, B])!

O_E([A, B]$_1$)

Main loop: pop&perform the task O_E([A, B]$_1$)!

A_R(*R1*, [A, B]$_1$)
A_R(*R2*, [A, B]$_1$)
A_R(*R3*, [A, B]$_1$)
A_R(*R4*, [A, B]$_1$)
O_G([A])
O_G([B])

Main loop: pop&perform the tasks O_G([B]) and O_G([A])!

… performing the tasks O_G([B]) and O_G([A]) is skipped, since trivial

Main loop: pop&perform the tasks A_R(*R4*, [A, B]$_1$), … A_R(*R2*, [A, B]$_1$)!

… none of the rules *R4*, …, *R2* is applied, since their
before-patterns don't match [A, B]$_1$

Main loop: pop&perform the task A_R(*R1*, [A, B]$_1$)!

O_E([A, B]$_2$) …see Figure 2.12,b)

Main loop: pop&perform the task O_E([A, B]$_2$)!

… all rules are disabled for [A, B]$_2$ $\Rightarrow$ no A_R-tasks are created
… the tasks O_G([A]) and O_G([B]), however, are created, but I skip that …

Main loop: pop&perform the tasks A_R(*R4*, [A, B, C]$_1$), A_R(*R3*, [A, B, C]$_1$)

… none of the rules *R4*, *R3* is applied, since their
before-patterns don't match [A, B, C]$_1$

Main loop: pop&perform the tasks A_R(*R2*, [A, B, C]$_1$)

O_E([A, B, C]$_2$)
O_E([A, B, C]$_3$)

… see Figure 2.12,c)
… skipping the execution of O_E([A, B, C]$_2$) and O_E([A, B, C]$_3$) …

Main loop: pop&perform the tasks A_R(*R1*, [A, B, C]$_1$)

O_E([A, B, C]$_6$)

… skipping again … Figure 2.12,d) shows the memo when
the optimization is finished.

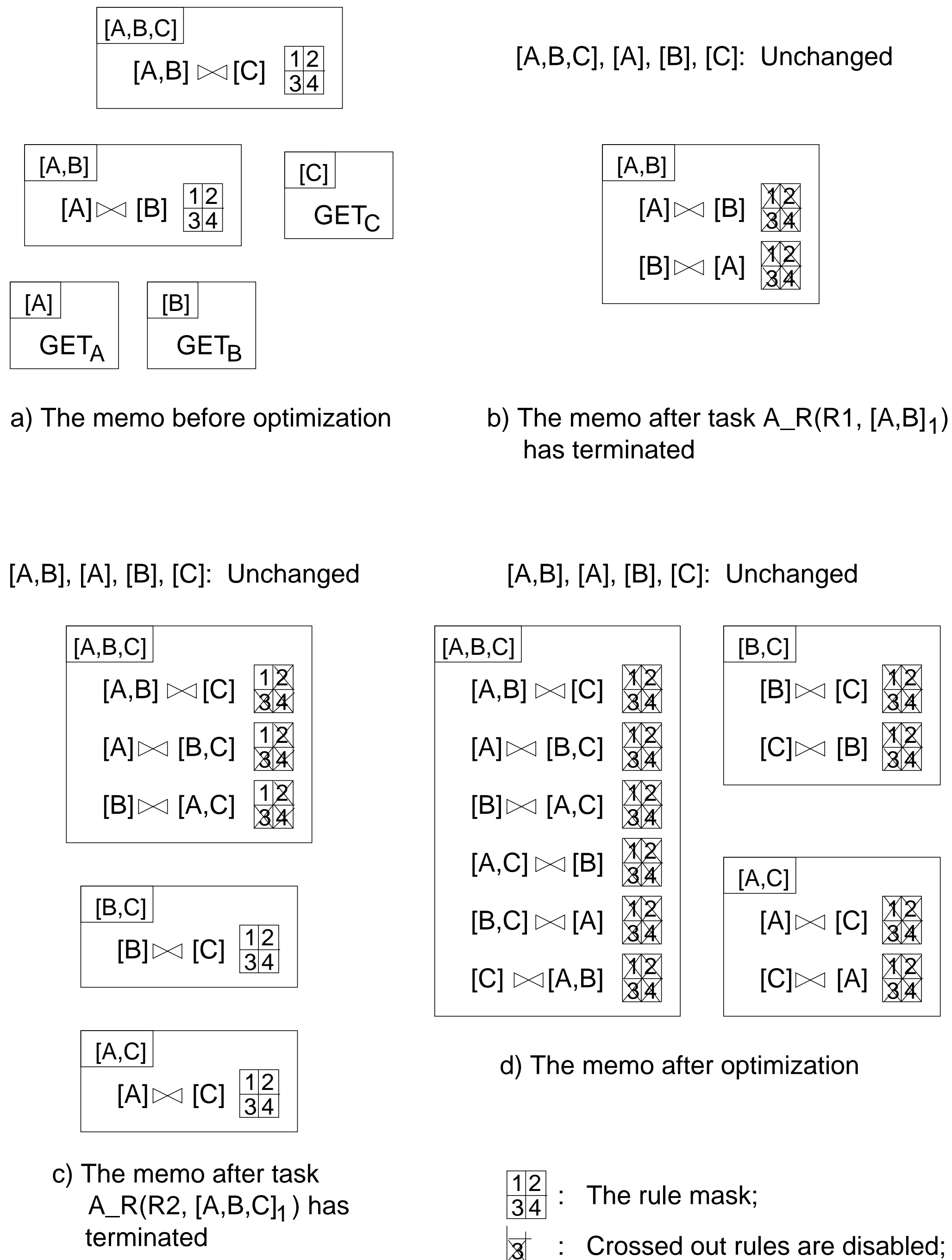


Figure 2.12: Optimizing $(A \bowtie B) \bowtie C$ with Cascades

Chapter 3

Extensibility of Cascades

To get some hands-on experience in extending a Cascades-based optimizer, I was granted access to the source code of the MIDAS database prototype [3]. The optimizer of MIDAS is based on Cascades [13, 21, 25] and, at the outset of my work, only the rules *R1*, ..., *R4* (cf. Section 2.4) were implemented. My goal with respect to the extensibility of Cascades was to implement two new transformation rules that would commute the operators JOIN and GROUP_BY.

As is proclaimed by Cascades' advocates, extending the model did not cause any conceptual problems. Nevertheless it turned out to be quite a lengthy endeavor of more than 1500 lines of C++ code. The real problem, however, was that once the extension was running, it substantially decreased optimizer performance. Indeed, depending on the query that was considered, optimization times assumed 2 to 50 times their initial values. This unexpected result lead me to mathematically analyze what the two new rules mean for the complexity of the optimization process. The present chapter presents this analysis. Here's a more detailed road-map of what lies ahead:

- Section 3.1 introduces the new rules that have been added to the optimizer.
- Section 3.2 mathematically analyzes the search space complexity of the extended rule-set.
- Section 3.3 proves the extended rule-set is to be duplicate-free.
- Section 3.4, finally, summarizes the implications that follow from this chapter for the extensibility of Cascades.

3.1 Commuting JOIN and GROUP_BY

The two transformation rules expounded in the present section are well-known to the research community [1, 8, 9, 40]. Their purpose is to push the logical GROUP_BY operator past a following JOIN operator. This potentially increases the effectiveness of an optimizer since GROUP_BY operators generally

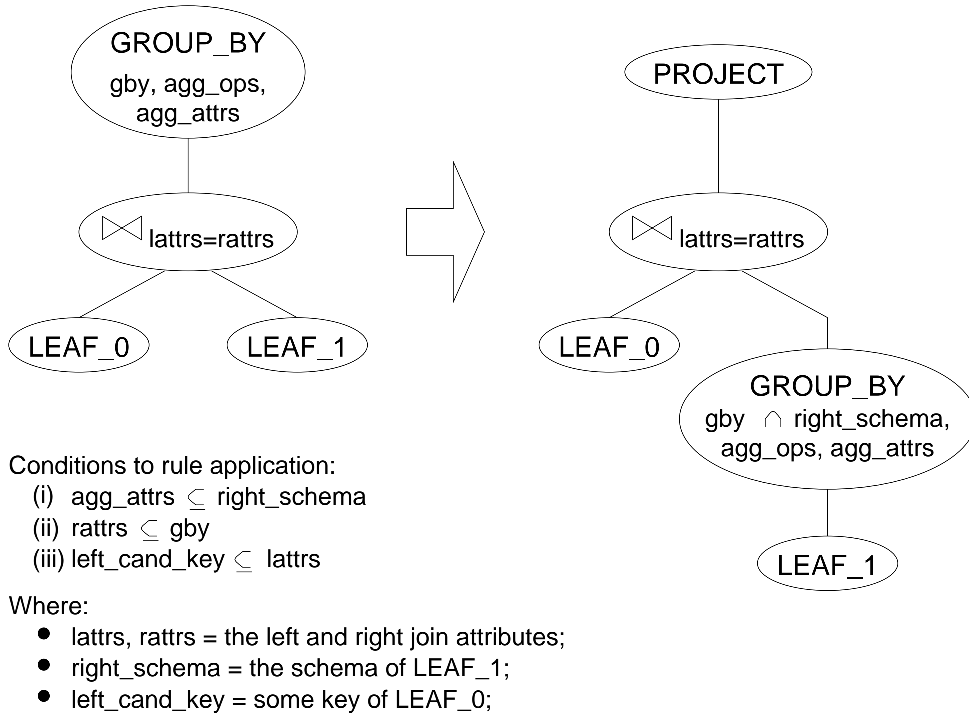


Figure 3.1: The rule *R5*

reduce the cardinality of their input relations. Therefore, an early evaluation of GROUP_BY operators can result in savings in the costs of subsequent JOIN operators.

For the purpose of this chapter, the GROUP_BY operator is assumed to possess three parameters: The grouping attributes (*gby*), the aggregate operators (*agg_ops*), and the attributes the aggregate operators are applied on (*agg_attrs*). Only the built-in aggregate operators COUNT, SUM, AVG, MIN, and MAX of SQL2 [11] are allowed. By way of illustration, consider the following query:

SELECT	age, sex, AVG(shoe_size), SUM(income)
FROM	COMPANY
GROUP BY	age, sex

For this query, it is *gby* = (age, sex), *agg_ops* = (AVG, SUM), and *agg_attrs* = (shoe_size, income). Note that there's a 1-to-1 correspondence between the elements of the lists *agg_ops* and *agg_attrs*.

Figure 3.1 shows the rule *R5*, the first of the two transformation rules that have been implemented. This rule simply interchanges the order in which grouping and joining are done. The PROJECT operator is needed to remove the attributes *schema*(LEAF_0)$\setminus$*gby* from the result. Please note that for rule *R5* to be applicable, the conditions *(i)* – *(iii)* of Figure 3.1 must hold. Otherwise, the before- and after-patter of the rule won't be logically equivalent!

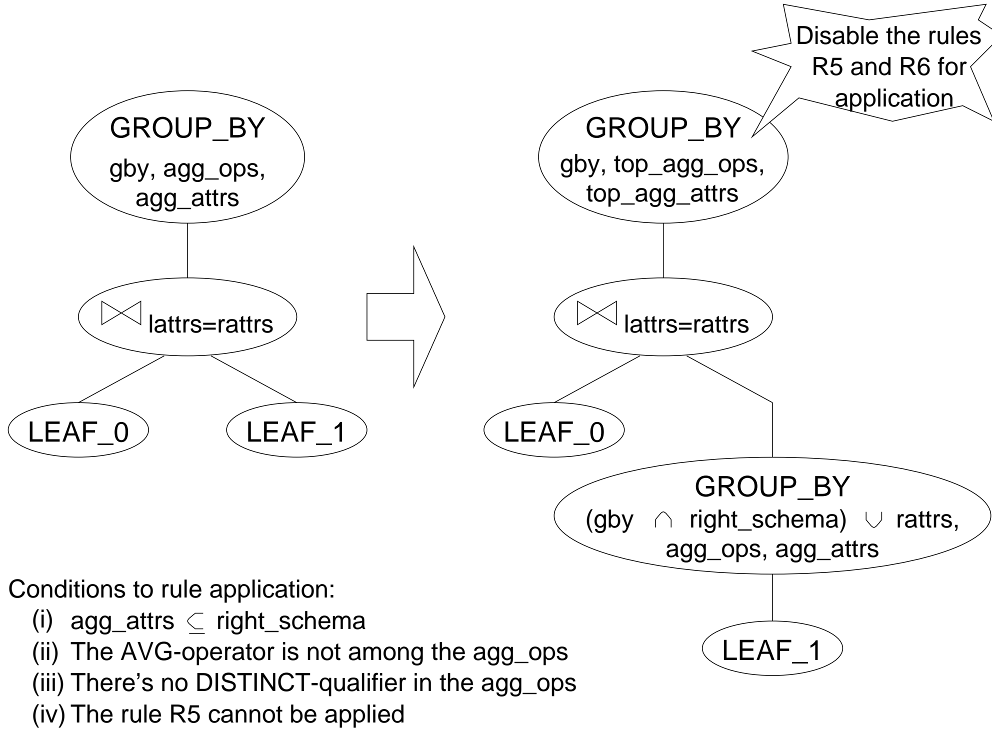


Figure 3.2: The rule *R6*

The conditions on the applicability of the rule *R5* are very restrictive. By relaxing these conditions, one gets the more general rule *R6*, which is shown in Figure 3.2. This rule can be fired in more instances but is less beneficial since it leads to two GROUP_BY operators. Therefore, the rule *R6* must not be fired when the rule *R5* is applicable. This convention is enforced by condition *(iv)* of rule *R6*. Finally, note that for both the rules *R5* and *R6*, no rules can be applied on the top-operators of their respective after-patterns.

3.2 Search Space Complexity

The objective of the present section is to analyze the impact of the two new rules *R5* and *R6* on the complexity of Cascades' optimization process. Subsection 3.2.1 more precisely states this goal. Subsection 3.2.2 gives an example that will prepare the general analysis which is expounded in Subsection 3.2.3. Subsection 3.2.4, finally, discusses how practically relevant the obtained results really are.

3.2.1 Defining the Problem

Please recall that the applicability of the rules *R5* and *R6* is subject to several conditions. The nuisance with these conditions is that they influence the optimization process in an almost unpredictable manner. Take the case of Figure

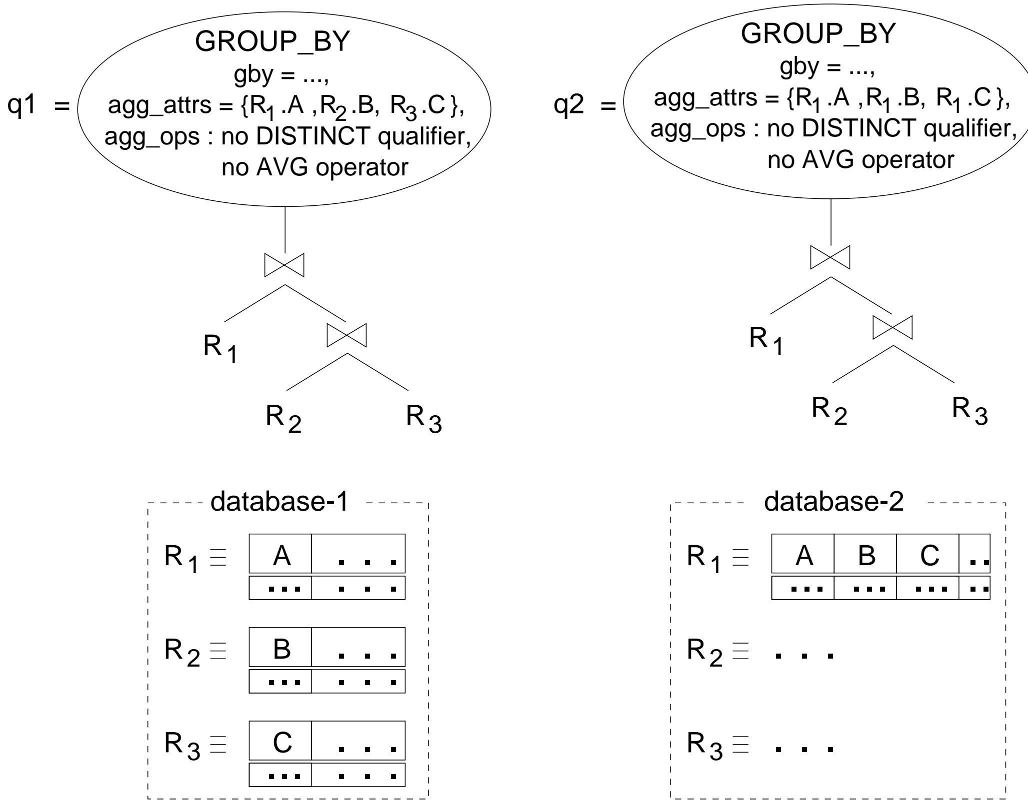


Figure 3.3: A problem caused by the rules *R5* and *R6*

3.3. In this example, the queries $q1$ and $q2$ are structurally identical and only differ in the parameters of their respective GROUP_BY operators. But still, the complexity of the optimization process is different for both queries: If the query $q1$ is optimized for database-1, then none of the two rules *R5*, nor *R6* can be applied. This is because the condition *(i)* of both rules will always evaluate to false. Now consider what happens when the query $q2$ is optimized for database-2. In this case, at least the conditions of rule *R6* are satisfied. Therefore, rule *R6* will be applied, leading to an increase in the complexity of the optimization process. The moral of this example is that structurally identical queries can still lead to optimization processes of different complexity. This effect does not arise for pure join-order optimization!

The above example illuminates just one of the many intricacies introduced by the rules *R5* and *R6*. In the general case, even more factors have a part to play. Therefore, a special class of queries will be defined in order to get a manageable problem. The scope of the forthcoming analysis will then be restricted to the queries of this class. To be concrete, the class of queries having the structure

$$\text{GROUP_BY}_{gby,\, agg_ops,\, agg_attrs}(R_1 \bowtie \ldots \bowtie R_n) \tag{3.1}$$

will be considered here. The R_i are relations and must not be confused with the rules *R1*, ..., *R6*! The following assumptions are made:

(i) $(R_1 \bowtie \ldots \bowtie R_n)$ is any expression joining the n relations $R_1, \ldots, R_n$.

(ii) All the joins are actually Cartesian products, i.e. their predicates are empty.

(iii) The GROUP_BY operator uses neither the AVG operator nor the DISTINCT qualifier.

(iv) There's one relation, say relation R_1, that contains all the aggregate attributes, i.e. $agg_attrs \subseteq schema(R_1)$.

(v) The optimizer uses the rule-set {*R1*, ..., *R6*}.

The class of queries just described has got two remarkable properties. In the first place, the queries of this class are of great practical relevance. Indeed, Subsection 3.2.4 will show that the huge majority of TPC-D queries is at least as complex as the queries considered here. In the second place, the queries of the above class lend themselves very well to a mathematical analysis. This is mainly for the following two reasons:

A) Rule *R5* will never be used while optimizing a query of the above class. To see why, please note that the condition *(iii)* of rule *R5* is never satisfied since all joins are Cartesian products. Thus, $lattrs = \varnothing \not\supseteq left_cand_key$.

B) Given the above assumptions, the condition of rule *R6* boils down to a very simple condition: Rule *R6* is applicable if the relation R_1 is contained in LEAF_1. (Remember: R_1 is assumed to be the relation which satisfies $agg_attrs \subseteq schema(R_1)$.)

The last issue to be addressed is how the complexity of the optimization process is measured. The complexity measure used in this text is the *search space complexity.* The search space complexity counts the number of groups generated during optimization. This quantity is invariant to improvements in Cascades' code, provided they do not abandon the completeness of Cascades' search strategy. Furthermore, the search space complexity also applies to all other top-down optimizers that do without pruning. Therefore, the search space complexity is a better complexity measure than, for example, the time complexity. (Note: The term *search space* is borrowed from artificial intelligence [6]. It designates the set of all objects generated during a search process. The memo obviously is Cascades' search space!)

The forthcoming analysis is going to focus on the performance impact the two new rules *R5* and *R6* had. Therefore, it won't count the absolute number of groups generated during optimization with the rule-set {*R1*, ..., *R6*}. Instead, the analysis will count by how many groups the search space complexity has *increased* after the rules *R5* and *R6* had been added to the rule-set {*R1*, *R2*, *R3*, *R4*}!

3.2.2 An Introductory Example

As explained in the last subsection, the scope of the forthcoming analysis is restricted to the queries of the above class. The goal of the analysis is to count the extra groups Cascades generates because of the rules *R5* and *R6*. To this end, it is essential to understand the pattern according to which new groups are generated by Cascades. The present subsection will explain this pattern with an example using the following query:

$$\mathsf{GROUP_BY}((R_1 \bowtie R_2) \bowtie R_3) \tag{3.2}$$

The parameters of the GROUP_BY don't influence the generation of new groups. Therefore they are not displayed. Needless to say that the assumptions *(i)* – *(v)* of page 32 also hold here! Finally, please recall that the modified Cascades algorithm as described in Section 2.4 on page 25 is used for this example and the rest of the analysis!

Figure 3.4,a) shows the memo after copying in the query (3.2). At that moment, the task-list only contains the task OPT_GROUP(*G1*). This task is popped, and replaced by the task OPT_EXPR(GROUP_BY($[R_1, R_2, R_3]$)), which, when terminated, leaves the following task-list (from TOP to BOTTOM):

OPT_GROUP($[R_1, R_2, R_3]$), APPLY_RULE(*R6*, GROUP_BY($[R_1, R_2, R_3]$))

To be exact, the task APPLY_RULE(*R5*, GROUP_BY($[R_1, R_2, R_3]$)) is also in the task-list. This task, however, can be ignored since rule *R5* will never be applied, see point *A)* on page 32.

Figure 3.4,b) shows the memo, after the task OPT_GROUP($[R_1, R_2, R_3]$) has been performed. The next task on the task-list is APPLY_RULE(*R6*, GROUP_BY($[R_1, R_2, R_3]$)). Please note that rule *R6* can be bound to each of the generalized multi-expressions GROUP_BY($\alpha \bowtie \beta$), $\alpha \bowtie \beta$ being a multi-expression of the group $[R_1, R_2, R_3]$. However, the condition of rule *R6* restricts rule application to those generalized multi-expressions that satisfy $R_1 \subseteq schema(\beta)$. Figure 3.4,c) shows the memo after rule *R6* has been applied.

Let's have a closer look at Figure 3.4,c)! First note that the groups *G5*, *G6*, *G7* of block 2 are pairwise different. To see why, just observe that the multi-expressions GROUP_BY($[R_1, R_2]$), GROUP_BY($[R_1, R_3]$), and GROUP_BY($[R_1]$) of these groups are pairwise syntactically un-equivalent. Therefore no merging will ever take place between the groups of block 2. Now its easy to see that the groups of block 1 are pairwise different, too. Thus far, rule *R6* is therefore responsible for the creation of 6 additional groups! These 6 groups are part of the increase in search space complexity that is caused by the new rules *R5* and *R6*.

The reader might find it interesting to explore how the optimization process proceeds. This, however, is no longer relevant to the general analysis. Therefore, the example ends here.

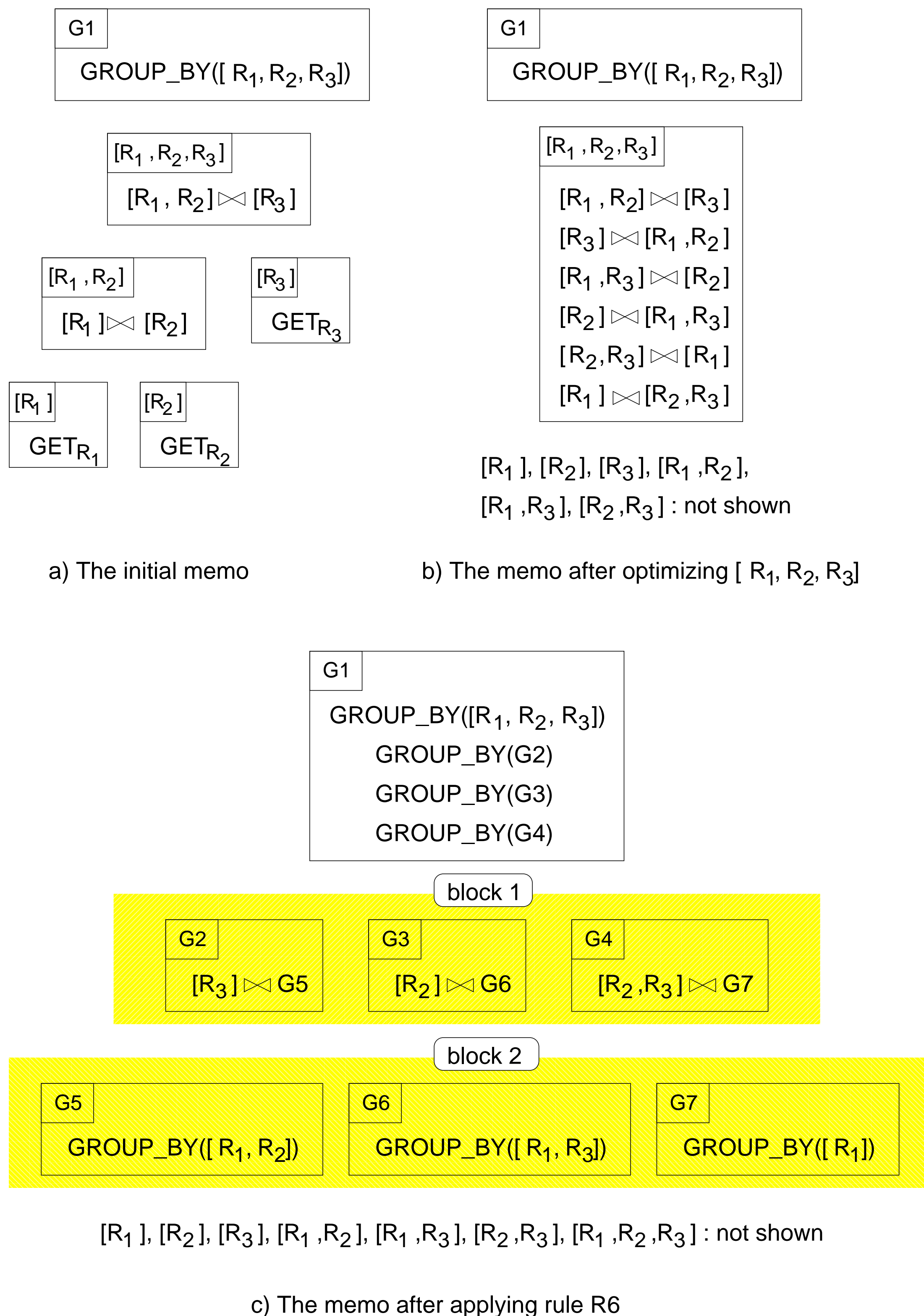


Figure 3.4: Optimizing GROUP_BY$((R_1 \bowtie R_2) \bowtie R_3)$

3.2.3 The General Counting Argument

This subsection proves that the new rules *R5* and *R6* increase the search space complexity of the optimization process by an additional 3^{n-1} groups.

Please recall that the scope of the following proof is restricted to the class of queries defined in Subsection 3.2.1. To make this point very clear, be q the query (3.1) of page 31. Then, the remainder of this subsection examines the optimization process of query q.

First of all, the *signature of a group* must be defined. To this end, be G an arbitrary group produced while optimizing the query q. Then, all the operator trees contained in G have got the same set of leaves. (This can be proved by induction on the number of rule applications.) Therefore it is well-defined to set:

$$\mathit{signature}(G) := \text{The set of leaves of the operator tree } t,\ t \text{ being any operator tree contained in group } G.$$

Take, for example, the groups $G1$, $G2$, and $G5$ of Figure 3.4. Obviously the following equalities hold: $\mathit{signature}(G1) = \mathit{signature}(G2) = \{R_1, R_2, R_3\}$, and $\mathit{signature}(G5) = \{R_1, R_2\}$.

To recap, the pursued goal is to count by how many groups the search space complexity of the optimization process has increased because of rule *R6*. Henceforth, these additional groups will be called *extra groups*. The group $G1$ also counts as an extra group. Strictly speaking, this is not correct since group $G1$ exists independently of rule *R6*. However, the error introduced by this convention is only 1, and the benefit for the following presentation is big.

To begin with, here are two important definitions: Be $\{R_{i_1}, \ldots, R_{i_k}\} \subseteq \{R_1, \ldots, R_n\}$, $R_1 \in \{R_{i_1}, \ldots, R_{i_k}\}$. Then

(i) $g(\,\{R_{i_1}, \ldots, R_{i_k}\}\,) := \{G \mid (G \text{ is an extra group}) \wedge (\mathit{signature}(G) \subseteq \{R_{i_1}, \ldots, R_{i_k}\})\}$.

(ii) $g_k := |\, g(\,\{R_{i_1}, \ldots, R_{i_k}\}\,)\,|$

The numbers g_k are well-defined since the cardinality of $g(\,\{R_{i_1}, \ldots, R_{i_k}\}\,)$ only depends on k, and the fact that $R_1 \in \{R_{i_1}, \ldots, R_{i_k}\}$. Obviously, g_n is the quantity sought after. It can be found, by counting the number of groups in $g(\,\{R_1, \ldots, R_n\}\,)$. To pursue this approach, the set $g(\,\{R_1, \ldots, R_n\}\,)$ is split into two partitions:

$$P_1 := \{G \mid G \in g(\,\{R_1, \ldots, R_n\}\,),\ \mathit{signature}(G) = \{R_1, \ldots, R_n\}\} \quad (3.3)$$

$$P_2 := g(\,\{R_1, \ldots, R_n\}\,) \setminus P_1 \quad (3.4)$$

Generalizing the example of subsection 3.2.2 will reveal that $|P_1| = 2^{n-1}$. In detail, here's what changes when optimizing the query q instead of $\mathsf{GROUP_BY}((R_1 \bowtie R_2) \bowtie R_3)$:

Initially, the group $G1$ contains the multi-expression $\mathsf{GROUP_BY}([R_1, \ldots, R_n])$. Optimizing the group $G1$ first triggers the optimization of the group $[R_1, \ldots, R_n]$. This, for its part, generates all the multi-expressions $[R_{i_1}, \ldots, R_{i_{n-k}}] \bowtie [R_{j_1}, \ldots, R_{j_k}]$, $1 < k < n$, $\{R_{i_1}, \ldots, R_{i_{n-k}}\} \bigcup \{R_{j_1}, \ldots, R_{j_k}\} = \{R_1, \ldots, R_n\}$. When Cascades is done with that, it resumes the optimization of group $G1$. Only rule *R6* can be applied on group $G1$. The before-pattern of rule *R6* can be bound to any of the $2^n - 2$ generalized multi-expressions $\mathsf{GROUP_BY}([R_{i_1}, \ldots, R_{i_{n-k}}] \bowtie [R_{j_1}, \ldots, R_{j_k}])$. However, only $(2^{n-1} - 1)$ of these bindings satisfy the rule's condition, which requires $R_1 \in \{R_{j_1}, \ldots, R_{j_k}\}$. Therefore, rule *R6* can only be fired $(2^{n-1} - 1)$ times, generating a total of $(2^{n-1} - 1)$ new groups in block 1, see Figure 3.4.

Obviously, P_1 just contains the groups of block 1, union the group $G1$. Therefore $|P_1| = (2^{n-1} - 1) + 1 = 2^{n-1}$.

It remains to determine $|P_2|$. P_2 obviously satisfies the equality $P_2 = \bigcup_{i=2}^{n} g(\{R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n\})$. However, $|P_2| \neq (n-1) \times g_{n-1}$, since the intersection of the sets $g(\{R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n\})$, $2 \leq i \leq n$, is not pairwise empty! Therefore the principle of inclusion&exclusion [18] must be used to count $|P_2|$:

$$|P_2| = \sum_{i=1}^{n-1} \binom{n-1}{i} g_{n-i} (-1)^{i+1} \tag{3.5}$$

Plugging $|P_1| = 2^{n-1}$ and equation (3.5) into $g_n = |P_1| + |P_2|$ yields:

$$g_n = 2^{n-1} + \sum_{i=1}^{n-1} \binom{n-1}{i} g_{n-i} (-1)^{i+1} \implies \tag{3.6}$$

$$2^{n-1} = \sum_{i=0}^{n-1} \binom{n-1}{i} g_{n-i} (-1)^{i} \tag{3.7}$$

Because of $2^{n-1} = (3 + (-1))^{n-1} = \sum_{i=0}^{n-1} \binom{n-1}{i} 3^{(n-i)-1} (-1)^i$, it follows that $g_n = 3^{n-1}$. Using a very similar argument, one can show that the number of additional multi-expressions (those that wouldn't be generated without rule *R6*) is $2 \times 4^{n-1} - 3^{n-1}$.

3.2.4 Discussion

To make the result of the last subsection absolutely clear, it must be stressed that Cascades already generates 2^n groups and $3^n + n + 2 - 2^{n+1}$ multi-expressions [30, 36] when using the rule-set $\{$*R1*, …, *R4*$\}$ to optimize the query q. Apart from these quantities, an *additional* 3^{n-1} groups and an *additional*

$2 \times 4^{n-1} - 3^{n-1}$ multi-expressions ensue if the rule *R6* is added to the rule-set. The present subsection will argue that this result – even though derived for a very particular class of queries – is a lower bound for many practically relevant cases.

Practically all TPC-D queries would be in the class of queries defined on page 32, if it wasn't for the assumption that all joins are Cartesian products. This assumption is generally not satisfied. Let us see, what this implies!

First, rule *R5* might become applicable. However, it is easy to see that this effect does not interfere with the above analysis: Just notice that either rule *R5*, or *R6* can be applied on any group. (This is enforced by condition *(iv)* of rule *R6*.) Next, verify that nothing changes in the above derivation if rule *R5* is applied instead of rule *R6*. Thus, if it was only for rule 1 becoming applicable, there would still be 3^{n-1} additional groups and $2 \times 4^{n-1} - 3^{n-1}$ additional multi-expressions.

However, a second effect arises when the join condition is not empty. To see what it is, look back at the definition of rule *R6*. The bottom GROUP_BY operator of the after-pattern has the grouping attributes $(gby \cap right_schema) \cup rattrs$. The fact that *rattrs* is contained in the grouping attributes makes the schema of the bottom GROUP_BY operator dependent on the JOIN from which the *rattrs* come. But the schema of the the GROUP_BY operator influences to which other multi-expressions it is syntactically equivalent. Therefore, depending on the JOIN operator involved when rule *R6* is applied, several syntactically inequivalent multi-expressions $\mathsf{GROUP_BY}_{rattrs}(G)$ will ensue for one and the same group G. Each of these multi-expressions must be put into a group of its own. In consequence, many more groups (and multi-expressions) will be generated when the join conditions are non-trivial. Thus, reality is even worse than what the above analysis predicts!

At the end of such a long section, it might be helpful to recall its starting point. Indeed, the initial objective was to explain why Cascades has become so slow after the rules *R5* and *R6* had been implemented. Obviously, the present section could successfully explain this decrease in performance.

3.3 Duplicate-Freeness of the Extended Rule-Set

Even though the result of the last section might be discouraging, there's also a piece of good news: The extended rule-set, i.e. $\{R1, \ldots, R6\}$, is duplicate-free. Intuitively this means that the rule-set $\{R1, \ldots, R6\}$ is such as to guarantee that a rule can never be applied if the memo already contains its after-pattern. What duplicate-freeness exactly means, and how it can be proved is the subject of the present section.

Subsection 3.3.1 defines duplicates in general and introduces the terminology. The Subsections 3.3.2 and 3.3.3 are specific to the rule-set $\{R1, \ldots, R6\}$ and prove its duplicate-freeness.

3.3.1 Duplicates in General

Duplicates arise if the application of a rule leads to an after-pattern that is already contained in the memo. This happens if and only if the memo already contains the primary multi-expression of the after-pattern. In this case, the primary multi-expression is called a *duplicate*. Duplicates are not reinserted into the memo, see Subsection 2.3.2.

It is almost impossible to overemphasize that duplicates always are *primary* multi-expressions. Of course, inserting an after-pattern into the memo might give rise to secondary multi-expressions which the memo already contains. But these multi-expressions are no duplicates. This convention makes sense since there is no rule-set that could possibly avoid the repeated generation of secondary multi-expression. However, avoiding duplicates, i.e. the repeated generation of primary multi-expressions, is possible, and thus deserves closer consideration!

Duplicates are subdivided into I-duplicates and X-duplicates. The distinction is based on which group of the memo contains the multi-expression that is identical to the duplicate: A primary multi-expression P is called an *I-duplicate* if the group it shall be put into already contains P. If another group of the memo already contains P, then P is called an *X-duplicate*.

Figure 3.5 illustrates these concepts by means of an example: Applying some rule on a multi-expression of the group G generates the after-pattern $G0 \bowtie (G1 \bowtie G2)$. The flow diagram shows how this after-pattern is inserted into the memo, and where duplicates can arise. The functions $\in$, $new()$, and $search_memo()$ are defined as follows:

- $M \in$ memo $:\Longleftrightarrow$ The memo contains the multi-expression M;
- $new(M)$: Creates a new group and initializes it to contain the multi-expression M;
- $search_memo(M)$: Returns the group of the memo that contains the multi-expression M, see Figure 2.8.

A rule-set is called *duplicate-free* if no chain of rule applications ever creates a duplicate. Obviously, it is highly desirable for a rule-set to be duplicate-free. Indeed, rule-sets that are not duplicate-free are frequently too inefficient for commercial applications. Therefore, proving the duplicate-freeness of the rule-set $\{R1, \ldots, R6\}$ constitutes a significant theoretical result.

3.3.2 Some Preliminaries

This and the following subsection contain the proof that the rule-set $\{R1, \ldots, R6\}$ is duplicate-free. The present subsection only introduces some conventions and simplifications. The main part of the proof can be found in Subsection 3.3.3.

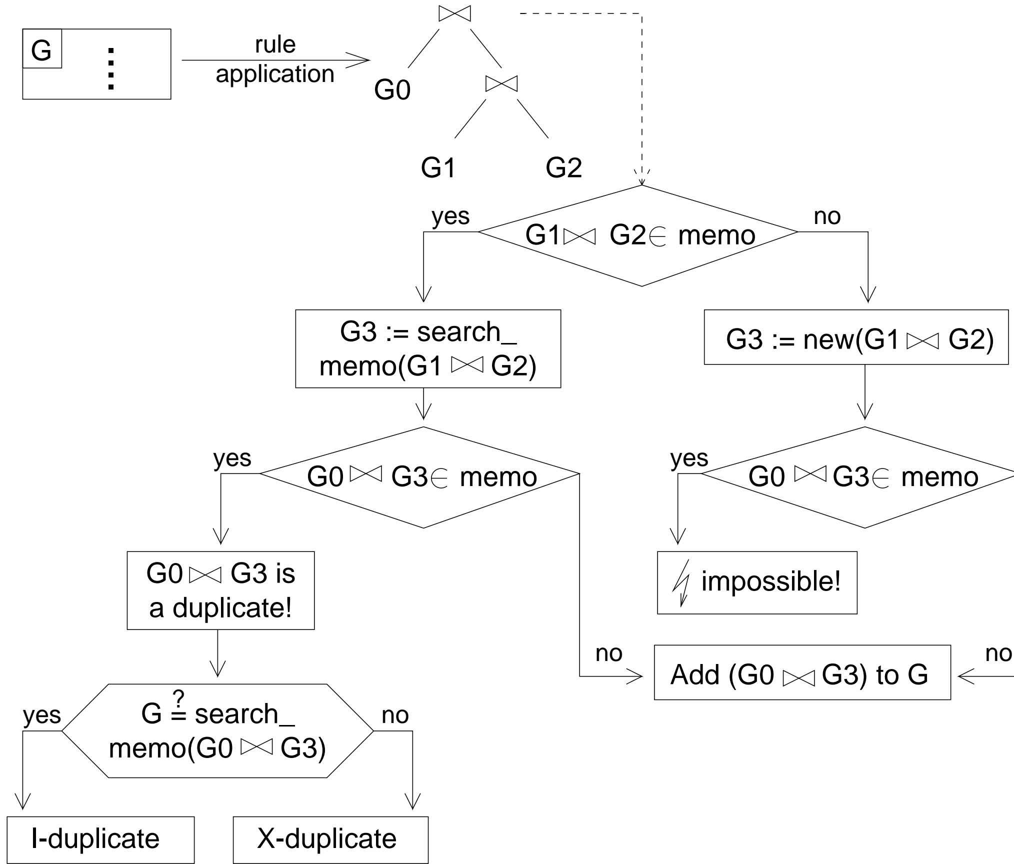


Figure 3.5: Duplicates

First of all, let's agree on the following convention: Whenever there will be mention of an optimization process, it is implicit that the rule-set $\{R1, \ldots, R6\}$ underlies the process. For brevity, however, this will not be mentioned each time! Furthermore, please note that the identifiers R_i, $R_i \neq R_j$, for $i \neq j$, denote relations or views. Don't confuse them with the rules *R1*, ..., *R6*!

The following lemma is central since it allows the further proof to be confined on queries of the type GROUP_BY($R_1 \bowtie \ldots \bowtie R_n$):

Lemma 3.1 *The rule-set* {R1, ..., R6} *is duplicate-free if no duplicates arise while optimizing queries of the type*

$$\mathit{GROUP_BY}(R_1 \bowtie \ldots \bowtie R_n) \tag{3.8}$$

Proof : First of all, note that the rules *R1*, ..., *R6* only deal with the operators JOIN and GROUP_BY. In consequence, the other SQL operators, such as PROJECT, SELECT, UNION, etc., do not participate in the optimization process. The same is true for inner (i.e. non-root) GROUP_BY operators since none of the rules *R1*, ..., *R6* has got a before-pattern that can be bound to a GROUP_BY operator, its father, and its son. Consequently, if duplicates are

possible, then they must arise while optimizing (sub-)trees of type (3.8). This completes the proof.

□

This subsection is concluded by the following definitions:

(i) The *signature of a multi-expression* is the signature of the group it is contained in.

(ii) The first multi-expression that is stored in a group after its creation is said to *initialize* the group.

(iii) A multi-expression of the type GROUP_BY($[R_{i_1}, \ldots, R_{i_j}]$) is said to be *Γ-formed*[1]. The R_{i_x} of this definition stand for arbitrary relations.

(iv) A multi-expression of the type $[R_{k_1}, \ldots, R_{k_l}] \bowtie W$ is said to be $\bowtie$-*formed*, if all multi-expressions of the group W have got GROUP_BY or π as operator. The R_{k_x} of this definition stand for arbitrary relations.

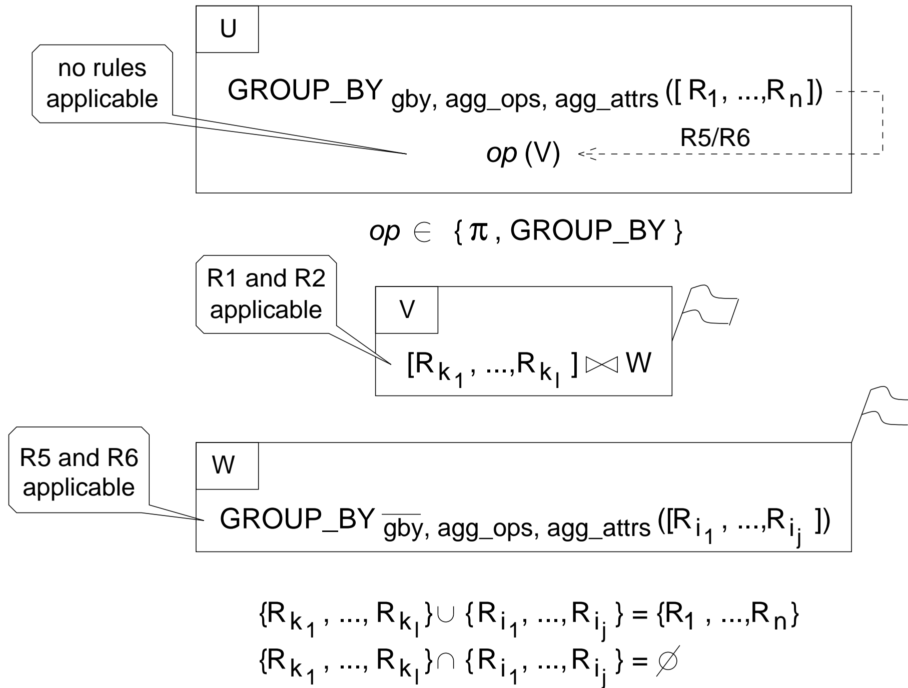
3.3.3 Proving Duplicate-Freeness

Now, the scene is set to prove the duplicate-freeness of the rule-set {*R1*, ..., *R6*}. As is implied by Lemma 3.1, it is sufficient to show that no duplicates arise while optimizing queries of the type GROUP_BY ($R_1 \bowtie \ldots \bowtie R_n$). First of all, note that the generation of duplicates does not depend on the order in which the rules are applied. Thus, one can suppose the rules were applied in the order set by Cascades. So, when a query of the above type is optimized, optimizing the group $[R_1, \ldots, R_n]$ is first completed before the first rule is applied on the multi-expression GROUP_BY($[R_1, \ldots, R_n]$), see Section 3.2. Pellenkoft [30] has shown that optimizing the group $[R_1, \ldots, R_n]$ is duplicate-free. Therefore, it is sufficient to concentrate henceforth on the optimization process that follows after the group $[R_1, \ldots, R_n]$ has been optimized. This part of the optimization process takes off as shown in Figure 3.6,a). Note that Figure 3.6,a) explicitly indicates the parameters of the GROUP_BY operators!

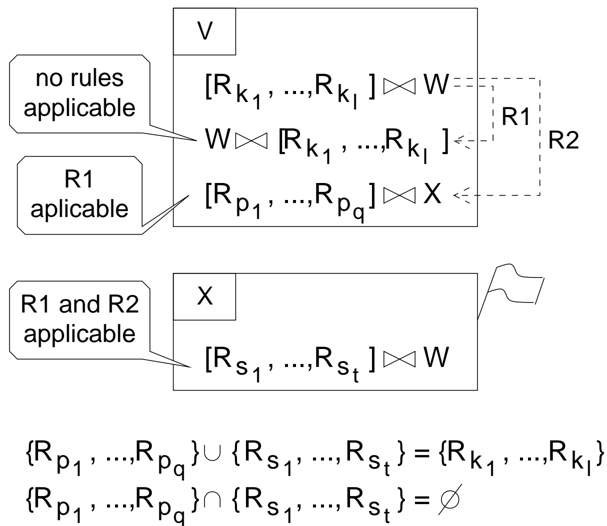
Let's have a closer look at Figure 3.6,a). The figure illustrates how one of the rules *R5* and *R6* is applied on the multi-expression GROUP_BY$_{gby,\, agg_ops,\, agg_attrs}$($[R_1, \ldots, R_n]$). The following steps are taken to insert the after-pattern of the rule into the memo:

1. If the secondary multi-expression GROUP_BY$_{\overline{gby},\, agg_ops,\, agg_attrs}$($[R_{i_1}, \ldots, R_{i_j}]$) is not contained in the memo, then the new group W is created to store it.
2. If the secondary multi-expression $[R_{k_1}, \ldots, R_{k_l}] \bowtie W$ is not contained in the memo, then the new group V is created to store it.

[1] Γ is the Greek letter for "G", like **G**ROUP_BY.



a) Applying one of the rules R5 and R6



b) Applying the rules R1 and R2

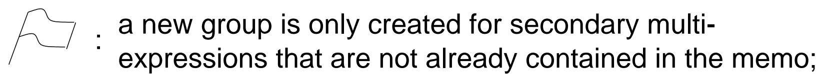


Figure 3.6: How optimization proceeds after $[R_1, \ldots, R_n]$ has been optimized

3. A new primary multi-expression of the type $\pi(V)$ or $\mathsf{GROUP_BY}(V)$ is added to the group U.

⇒ Note that all newly created groups are initialized with multi-expressions that are $\bowtie$-formed or Γ-formed. Furthermore, the parameters *agg_ops* and *agg_attrs* are identical for all Γ-formed multi-expressions.

The newly generated groups and multi-expressions also take part in the optimization process. However, no rule can be applied on the multi-expression that has been added to the group U. Furthermore, optimizing the group W is analogous to optimizing U. Therefore, only the group V needs further examination. Obviously, commutativity (rule *R1*) and right-associativity (rule *R2*) are the only rules that can be applied on the multi-expression $[R_{k_1}, \ldots, R_{k_l}] \bowtie W$ of group V. The rule *R1* is of no further interest since it cannot create new groups (cf. Figure 3.6,b)). Rule *R2*, however, can create a new group. This is illustrated in Figure 3.6,b). The important point to note is that the new group X, provided it is created, is initialized with a $\bowtie$-formed multi-expression. Therefore, optimizing X proceeds analogously to optimizing V.

Using the above considerations, it is easy to prove the following lemma by induction over the number of rule applications. (Note that the first point of the lemma is generally true, and not specific to the current context.)

Lemma 3.2 *The tail-part of the optimization process, i.e. the part that follows after Cascades has optimized the group $[R_1, \ldots, R_n]$, exhibits three important properties:*

(i) New groups are only generated for secondary multi-expressions that are not already contained in the memo.

(ii) All newly created groups are initialized with a $\bowtie$-formed or a Γ-formed multi-expression.

(iii) The aggregate operators and the attributes they are applied on are identical for all Γ-formed multi-expressions.

□

According to point *(ii)* of Lemma 3.2, the rest of the proof can concentrate on groups that were initialized with $\bowtie$-formed or Γ-formed multi-expressions. If no duplicates arise when optimizing these groups, then no duplicates at all arise. This line of reasoning motivates the following two propositions:

Proposition 3.1 *Be G a group that was initialized with a $\bowtie$-formed multi-expression. Then no duplicates arise while optimizing G.*

Proof : The proof treats I-duplicates separately from X-duplicates. First, let's consider I-duplicates. To this end, be $[R_{k_1}, \ldots, R_{k_l}] \bowtie W$ the multi-expression that G was initialized with. Suppose optimizing G generated an I-duplicate. Then a corresponding I-duplicate would be generated when optimizing the group $\overline{G}$ which contains the multi-expression $[R_{k_1}, \ldots, R_{k_l}] \bowtie \overline{W}$, $\overline{W}$ being a relation. This contradicts the fact that no duplicates are generated for pure join-order optimization [30]. Thus, I-duplicates are impossible.

Next, let's look at X-duplicates. To this end, be $[R_{k_1}, \ldots, R_{k_l}] \bowtie W$ the multi-expression that G was initialized with, see Figure 3.6,b). Optimizing G will add new multi-expressions to G. However, all of these multi-expressions will inevitably contain the relations $R_{k_1}, \ldots, R_{k_l}$ and the group W. This is simply because the before- and after-pattern of the rules *R1* and *R2* only differ in their operators, but not in their leaves. The bottom line is that the characteristic tuple $(\{R_{k_1}, \ldots, R_{k_l}\}, W)$ represents a property that is shared by all multi-expressions of the group G.

Suppose now that optimizing G generated an X-duplicate, i.e. a primary multi-expression that is already contained in some other group $\overline{G}$. Then, the multi-expressions in the groups G and $\overline{G}$ must have the same characteristic tuple $(\{R_{k_1}, \ldots, R_{k_l}\}, W)$. However, there's only one $\bowtie$-formed multi-expression having this characteristic tuple. According to Lemma 3.2 *(ii)*, this multi-expression has been used to initialize both, the group G and the group $\overline{G}$. However, according to Lemma 3.2 *(i)*, it is not possible that two different groups are initialized with the same multi-expression. The younger of the two groups would not have been generated. A contradiction to the above assumptions. Thus, X-duplicates cannot arise.

□

Proposition 3.2 *Be G a group that was initialized with a Γ-formed multi-expression. Then no duplicates arise while optimizing G.*

Proof : The proof treats I-duplicates separately from X-duplicates. First, let's consider I-duplicates. To this end, be $\mathsf{GROUP_BY}([R_1, \ldots, R_n])$ the multi-expression initially stored in G. Then, the reasoning goes as follows: If applying a rule on $\mathsf{GROUP_BY}([R_1, \ldots, R_n])$ adds the new multi-expression M to G, then no rules can be applied on M. Therefore, if I-duplicates are possible, then they must be generated in one step by applying a rule on $\mathsf{GROUP_BY}([R_1, \ldots, R_n])$. However, each time a rule is applied, it is bound to a different of the $2^n - 2$ generalized multi-expressions $\mathsf{GROUP_BY}([R_{k_1}, \ldots, R_{k_l}] \bowtie [R_{i_1}, \ldots, R_{i_j}])$. Each of these generalized multi-expressions has got a unique pair $([R_{k_1}, \ldots, R_{k_l}], [R_{i_1}, \ldots, R_{i_j}])$ of input groups, which is copied into the after-pattern of the rule. Thus, the after-patterns, and thereby their primary multi-expressions, become unique, too. This rules out I-duplicates.

Next, let's look at X-duplicates. To this end, be M the multi-expression that G was initialized with. Note that M is Γ-formed. Suppose now that optimizing G generated an X-duplicate, i.e. a primary multi-expression that is already contained in some other group $\overline{G}$. Then $\overline{M}$, the multi-expression that $\overline{G}$ was initialized with, must be Γ-formed, too. Furthermore the equalities

$$signature(M) = signature(\text{“duplicate”}) = signature(\overline{M})$$
$$schema(M) = schema(\text{“duplicate”}) = schema(\overline{M})$$

must hold since the signature and the schema is constant over all multi-expressions of a group.

Be $M = \mathsf{GROUP_BY}_{gby,\, agg_ops,\, agg_attrs}([R_{s_1}, \ldots, R_{s_t}])$ and $\overline{M} = \mathsf{GROUP_BY}_{\overline{gby},\, agg_ops,\, agg_attrs}([R_{p_1}, \ldots, R_{p_q}])$. First, note that the parameters agg_ops and agg_attrs are identical for both multi-expressions. This is implied by Lemma 3.2 *(iii)*. Furthermore, from $signature(M) = signature(\overline{M})$ follows $[R_{s_1}, \ldots, R_{s_t}] = [R_{p_1}, \ldots, R_{p_q}]$. The equality $schema(M) = schema(\overline{M})$ finally implies $gby = \overline{gby}$. Therefore, the two multi-expressions M and $\overline{M}$ are identical. This however contradicts Lemma 3.2 *(i)*: If M was identical to $\overline{M}$, then one of the two groups G and $\overline{G}$ would not have been generated. It follows that X-duplicates cannot arise.

□

The proof is completed by the following corollary, which directly follows from the Propositions 3.1 and 3.2:

Corollary 3.1 *The extended rule-set* {R1, ..., R6} *is duplicate-free.*

□

3.4 Generalization

The present section will elaborate on what follows from the preceding two sections for the extensibility of Cascades. As a starting point, let's recall the motivation for Section 3.2. After the rules *R5* and *R6* had been implemented, it has been observed that Cascades has become unbearably slow. In general, one cannot simply put up with an observation like that. Instead, the decrease in performance must be tracked back to its origins and some countermeasure must be taken. The most natural countermeasure is to modify the rule-set, maybe by tightening the conditions of some rules, maybe by erasing some rules altogether. This kind of work requires an in-depth understanding of the innards of Cascades, far beyond what is needed to implement new rules!

A very common reason for long optimization times are non-duplicate-free rule-sets. Therefore, only duplicate-free rule-sets should be implemented. This, however, is easier said than done, because at the time of this writing, the problems of constructing duplicate-free rule-sets or of proving the duplicate-freeness of a rule-set are not very well understood [30]. Therefore, the problems associated with duplicates constitute another big burden that goes along with extending Cascades.

It is wrong to believe that the kind of problems described here only arises for very complex rule-sets! Indeed, when I began the work on the present chapter, I realized that Keith Billings [1] has been the only author to use the rules *R5* and *R6* in a top-down optimizer. Despite his groundwork, two adjustments were necessary to better adapt the rules *R5* and *R6* to the top-down environment:

1. I have added the condition *(iv)* to the rule *R6* in order to curb the complexity of the optimization process.

2. I decided to disable the rules *R5* and *R6* in the after-pattern of the rule *R6*. Keith Billings implemented a slight variant on this rule: Instead of disabling the rules *R5* and *R6*, a further condition was implemented to restrict the applicability of the rule *R6*. This condition requires that the top-operator of LEAF_1 be different from GROUP_BY (cf. Figure 3.2). It is easy to see that Billings' rule-set is not duplicate-free! Furthermore, I could show that Billings' rule-set is equivalent to the rule-set presented here.

The above considerations make one point very clear: The real problem of extending Cascades lies in finding an appropriate rule-set, not in implementing this rule-set. Finding an appropriate rule-set involves understanding the impact different rule-sets have on the run-time behavior of Cascades. This, however, requires a profound understanding of Cascades! Therefore, contrary to what is commonly said, the mere knowledge of Cascades' interface classes is not sufficient to extend Cascades.

Finally, it should be pretty obvious that all top-down optimizers share the problems described here. Therefore, the statement top-down optimizers were easy to extend must be taken with a grain of salt.

Chapter 4

Improving the Code of Cascades

Cascades is a software package of almost 8000 lines of C++ code. Not surprisingly, there are several passages in the code of Cascades that could have been implemented more efficiently. The present chapter highlights three such passages. Furthermore, improvements upon the existing code are described. The effectiveness of these improvements will be assessed by means of TPC-D queries.

It must be emphasized that the present chapter only treats those improvements that do not prune the search space. Pruning is the subject of the next chapter. Here's finally a more detailed outline of the topics that will be covered in the following sections:

– Cascades uses a "handmade" implementation of dynamic arrays. Section 4.1 scrutinizes what can be gained by switching to a more efficient commercial implementation of dynamic arrays.

– Section 4.2 presents a more efficient implementation of the binding algorithm.

– Section 4.3 explains that the winner's circle should not be confined to storing only winners. Instead, it should also store when there is no winner at all for a given context.

– Section 4.4 finally assesses each of the above improvements. For this purpose the TPC-D queries Q1, Q3, Q5, Q7, Q8, Q9, Q10, and Q13 will be used. It will be shown that all the improvements combined accelerate Cascades by a factor of 2 to 100, depending on the query you look at.

All but the first of the following three improvements originate in Columbia [33, 37], the successor of Cascades. Columbia also prompts several other supposedly efficiency increasing measures. The effectiveness of these measures, however, could not be validated by means of the TPC-D queries. Therefore, these measures were not included in the present chapter. Instead, they are described in Appendix A.

4.1 Dynamic Arrays

Cascades uses a handmade implementation of dynamic arrays. Friends of mine who have switched from similar handmade implementations to commercial implementations have urged me to do alike. To my very surprise, however, I had to find out that the allegedly more efficient commercial implementation slowed Cascades down. This unexpected finding will be explained in two steps. Firstly, it will be expounded how commercial implementations of dynamic arrays work. Thereupon it will become clear why these implementations are not suited to Cascades.

So let's have a closer look at how commercial implementations of dynamic arrays work. The main focus of these implementations is to make the addition operation as fast as possible. The addition operation adds a new element to the end of an array. If the allocated memory block is too small to accommodate the new element, then a larger block of memory must be allocated and the whole array must be copied into this block. This copying consumes time linear in the size of the array and threatens to make the addition operation inefficient.

Commercial implementations meet this possible inefficiency in the following way: Each time the current memory block becomes too small, a new one is allocated and its size is chosen to be twice the size of the old memory block. Thus a lot of excess space is allocated which can be used to handle several of the upcoming addition operations in constant time. This exponential growth in the space allocated for an array causes the amortized time [24, 26] for addition to be constant.

In general, the *amortized time for an operation* is the total time for a sequence of N operations divided by N. By way of illustration, consider a sequence of N addition operations. Obviously, the (2^i+1)-th addition operation, $i = 0, 1, \ldots, \lceil \log_2(N) \rceil - 1$, costs $(2^i + 1)$ time units, because it must copy the array into a new memory block. All other addition operations cost one time unit. Thus, N addition operations costs $N + \sum_{i=0}^{\lceil \log_2(N) \rceil - 1} 2^i \leq 3 \times N$ time units in total. Dividing this quantity by N shows that the average or amortized time for addition is constant.

Obviously, commercial implementations of dynamic arrays only develop their full potential in applications where arrays frequently grow. Cascades, however, is most emphatically not of this type. In fact, Cascades *never* changes the size of a dynamic array once it has been allocated. Therefore, Cascades does not benefit from commercial implementations of dynamic array.

In principle, this section could end here. However, the present thesis is about top-down optimizers in general, not just about their search engines. Furthermore, the model of the Cascades-based top-down optimizer I am using heavily relies on dynamic arrays, too. For these reasons, it was tempting to examine what can be gained by tailoring the implementation of dynamic arrays to the needs of the model. Indeed, this approach could reduces the execution time of the employed top-down optimizer by an average of 6%. The remainder of this section will elaborate on the details of this approach.

First of all, commercial implementations of dynamic arrays turned out to be useless once again. Studying the way the model uses its dynamic arrays, however, revealed a remarkable pattern, which can be summarized in three points:

1. Approximately 60% of all dynamic arrays have size 2 or less at the time they are destroyed.
2. Approximately 95% of all dynamic arrays never grow during their lifetime.
3. Those arrays which grow during their lifetime hardly grow more than once.

These findings have motivated the following strategy for dynamic arrays: Whenever a new array is created, it is assigned enough space for at least two elements. This takes the first of the above three points into account. Furthermore, whenever an array grows beyond its allocated space, just enough new space is allocated to meet the new memory requirements. No extra space is allocated since – according to the last of the above three points – this space is more than likely to be wasted. Implementing this strategy in a very succinct and efficient way lead to the already mentioned 6% reduction in execution time.

4.2 A Better Binding Algorithm

The binding algorithm (cf. Subsection 2.3.1) essentially looks for matchings between the before-pattern of a given rule and the logical multi-expressions of the memo. The large number of logical multi-expressions in the memo makes this task very time-consuming. Indeed, the function BINDERY::advance() which finds the bindings is among the most expensive functions in Cascades. Therefore, it was considered promising to completely revise the binding algorithm.

First of all, revising the binding algorithm did not change the way it basically works. Thus, the pseudo-code of Figure 2.6 remains valid. However, the implementation of the binding algorithm has been trimmed and tuned to make it as efficient as possible. In fact, it was possible to reduce the length of the binding algorithm to half its original length. The details of how this reduction has been achieved are beyond the scope of this text, but can be found in the source code of Cascades (cf. file "opt.C").

At this place, only one improvement to the binding algorithm deserves further consideration. This improvement is closely intertwined with the way the multi-expressions of a group are stored. Cascades has traditionally used a linked list for this purpose. In particular, the logical and physical multi-expressions of a group were put into the very same list. This storage structure, however, makes binding inefficient since binding is exclusively about the logical multi-expressions of the memo. In fact, the physical multi-expressions of the memo do not participate in any bindings. Therefore, they only slow down the binding algorithm because they have to be skipped over.

The obvious remedy is to store the logical and physical multi-expressions of a group in two separate lists. This separation does away with the skipping and therefore accelerates the binding algorithm. Implementing the separation of logical and physical multi-expressions has been done in the course of revising the binding algorithm.

All in all, the new implementation of the binding algorithm saves approximately 25% of optimization time. Section 4.4 contains the exact benchmark results.

4.3 A New Winner's Circle

The concept of a winner's circle has been introduced in Subsection 2.2.2. Very briefly, every group has got a winner's circle where it saves the cheapest possible physical multi-expression for each context it has been optimized for. The multi-expressions in the winner's circle are called winners since they are cheapest among the group's physical multi-expressions which have the same physical properties as the winner.

It is possible that optimizing a group for some context is unsuccessful. In this case, there is no winner for this particular context. Cascades deals with this case by not saving anything in the winner's circle. But just forgetting about the fact that there is no winner for some context seems unwise. After all, the non-existence of a winner is a result, too, even though it's a negative one. Therefore, the implementation of the winner's circle has been changed. In its new implementation, the non-existence of a winner is also stored in the winner's circle. The resulting speedup is tremendous. Indeed, the new implementation makes Cascades up to 80 times faster. In the remainder of this section, the sources of this speedup will be revealed.

Let's consider the example of Figure 4.1. For the sake of simplicity, the example is based on a rule-set that contains no transformation rules. More precisely, the following rule-set is used throughout the example:

$$\{ \quad \mathsf{JOIN} \longrightarrow \mathsf{MERGE_JOIN}, \\ \mathsf{JOIN} \longrightarrow \mathsf{HASH_JOIN}, \\ \mathsf{TIMES} \longrightarrow \mathsf{NESTED_LOOPS_JOIN} \quad \}$$

Be C the context that the group G_n is optimized for. Let us suppose the upper cost bound of C was smaller than the cost of the cheapest physical multi-expression in G_0. This assumption is reasonable since G_0 contains a Cartesian product and Cartesian products are generally very expensive. The consequence of C having such a tight upper cost bound is that there is no winner for the groups $G_0, \ldots, G_n$. The point to be made here is that optimizing the group G_n takes exponential time when non-existence results are discarded, whereas linear time is sufficient when they are saved in the winner's circles of the groups. This assertion will be proved next.

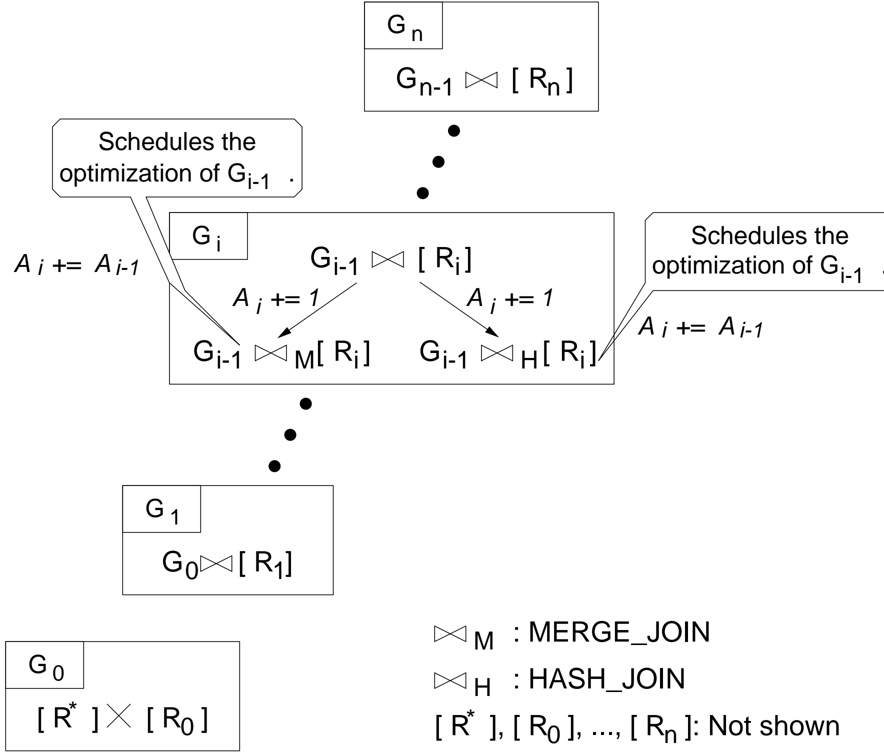


Figure 4.1: There's a penalty on forgetting non-existence results!

The case where non-existence results are discarded is considered first. For simplicity, the number of rule applications is used to measure the time complexity of the optimization process. More precisely, let A_i be the number of rules that are applied in the course of optimizing the group G_i. Obviously $A_0 = 1$, $A_1 = 4$, and more generally

$$A_i = 2 \times (1 + A_{i-1}), \text{ for } i > 0. \tag{4.1}$$

Equation (4.1) is easy to prove. As indicated in Figure 4.1, each of the rules JOIN $\longrightarrow$ MERGE_JOIN and JOIN $\longrightarrow$ HASH_JOIN is applied on the multi-expression $G_{i-1} \bowtie [R_i]$ of group G_i. Furthermore, each of these rule-applications schedules the group G_{i-1} for optimization. But each time the group G_{i-1} is optimized, yet another A_{i-1} rule applications arise. This follows from the fact that non-existence results are not saved. Equation (4.1) follows.

A simple induction is sufficient to show that $A_i = 2^{i+1} + 2^i - 2$ holds. In particular, it follows that $A_n = 2^{n+1} + 2^n - 2 \in O(2^n)$, which completes the first part of the proof. The second part of the proof is to show that $O(n)$ rule applications are enough to optimize the group G_n, provided non-existence results are stored in the winner's circle. This case, however, is very similar to the above case and is thus not considered any further.

It is wrong to believe the above example was artificial. Just observe that Cartesian products are ubiquitous in optimization. Furthermore, their cost is generally prohibitive and thus violates almost any upper cost bound. In consequence, the problem sketched by the above example is lurking just everywhere during the optimization process. Based on this insight, it is understandable why storing non-existence results is so totally effective.

4.4 Results

The improvements described in the preceding three sections have been incorporated into Cascades in the indicated order. Upon completion of each of the improvements, the TPC-D benchmark queries Q1, Q3, Q5, Q7, Q8, Q9, Q10, and Q13 were run to assess efficiency gains. Table 4.1 shows the obtained optimization times measured under low stray load on a 295 MHz Sun UltraSPARC-II processor with 384 MB of main memory and 796 MB of virtual memory. For query Q8, the computer still ran out of virtual memory. Thus no optimization time is available for this query. Furthermore, no exact time is given for those queries that took a second or less to optimize. The reason for that is that the optimization times for these queries proved to be very sensitive to the stray load. Therefore, no meaningful times could be obtained.

The measurements of Table 4.1 are based on a rule-set which comprises the transformation rules *R1*, ..., *R6* plus a suitable set of implementation rules. In particular, the set of implementation rules contains the following four rules for mapping the logical JOIN operator onto a physical one:

- JOIN $\longrightarrow$ NESTED_LOOPS_JOIN,
- JOIN $\longrightarrow$ INDEXED_LOOP_JOIN,
- JOIN $\longrightarrow$ MERGE_JOIN, and
- JOIN $\longrightarrow$ HASH_JOIN.

	Q1	Q3	Q5	Q7	Q8	Q9	Q10	Q13
Initially	$\ll$1s	$\ll$1s	1400s	1.3s	–	38s	1.5s	$\ll$1s
Dynamic arrays	$\ll$1s	$\ll$1s	1310s	$\approx$1s	–	35s	1.3s	$\ll$1s
New binding algorithm	$\ll$1s	$\ll$1s	945s	$\approx$1s	–	28s	$\approx$1s	$\ll$1s
New winner's circle	$\ll$1s	$\ll$1s	12.5s	$\approx$1s	–	5.5s	$\approx$1s	$\ll$1s

Table 4.1: Optimization times on a Sun UltraSPARC-II 295 MHz CPU

Please observe that all four improvements combined result in optimization times which are a sixth to a hundredth of their initial values. Furthermore, it is striking that query Q5 benefits disproportionately more from the above

improvements than any other query. This phenomenon results from paging, or its absence, respectively. Indeed, changing the semantics of the winner's circle saves enough memory to eliminate the need for paging. The resulting speedup is explosive since disk access is very slow. The availability of enough main memory is therefore central to the efficiency of Cascades. Section 6.3 will have more to say on this topic.

Table 4.1 suggests that the above improvements are of little benefit to small, fast-to-optimize queries. To scrutinize this conjecture, I ran the queries Q1, Q3, Q7, Q10, and Q13 under low stray load on a 110 MHz Sun microSPARC-II processor with 64 MB of main memory and 170 MB of virtual memory. This machine is significantly slower than the one that has been used beforehand. In consequence, more meaningful optimization times could be obtained, see Table 4.2. Comparing the Tables 4.1 and 4.2 clearly shows that the efficiency-increasing measures of this chapter are particularly effective for big, complex queries. Smaller, less complex queries, on the other hand, benefit less from these measures. As will be shown in the next chapter, the same is true for group pruning.

	Q1	Q3	Q7	Q10	Q13
Initially	1s	2.2s	6.5s	6.6s	0.5s
Dynamic arrays	1s	2.2s	6.3s	6.3s	0.5s
New binding algorithm	1s	2.2s	6.0s	6.1s	0.5s
New winner's circle	1s	1.4s	3.8s	4.8s	0.5s

Table 4.2: Optimization times on a Sun microSPARC-II 110 MHz CPU

The above benchmark results are in line with what can be expected by looking at the size and complexity of the different queries. First of all, each of the above queries contains a GROUP_BY operator. The number of JOIN operators, however, varies widely from one query to another. This is illustrated by Table 4.3, which clearly explains why the queries Q1, Q3, Q10, and Q13 are fast to optimize, whereas the query Q8 is extremely complex. However, it is not yet clear why the queries Q5, Q7, and Q9 result in so different optimization times, even though they agree on the number of JOIN operators. To explain this phenomenon, it is necessary to look more closely at the individual queries:

Query Q5: The query Q5 is of the type that has been considered in Section 3.2. Thus, the long optimization time should not come as a surprise. Indeed, the optimization time of query Q5 is mainly due to the rules *R5* and *R6*. When these rules are switched off, then optimizing the query Q5 takes just as long as optimizing the query Q7!

Query Q7: The optimizer which I've been using to run the benchmark queries actually consists of a heuristic front-end and a Cascades-based back-end. In the case of the query Q7, the front-end passes a query of the type GROUP_BY($\sigma(R_1 \bowtie \ldots \bowtie R_6)$) to the back-end. Note that because of

	Q1	Q3	Q5	Q7	Q8	Q9	Q10	Q13
No. of JOIN operators	0	2	5	5	7	5	3	1

Table 4.3: Join-complexity of the different TPC-D queries

the SELECT operator, it is not possible to apply the rules *R5* or *R6* on this query. Thus, the optimization time of query Q7 is just the time of optimizing the sub-query $(R_1 \bowtie \ldots \bowtie R_6)$.

Query Q9: The query Q9 is of the type (3.1) (see p. 31), but condition *(iv)* of page 32 is not satisfied. In fact, the GROUP_BY operator of query Q9 has its aggregation attributes spread over two different relations. This restricts the applicability of the rules *R5* and *R6* and places the optimization time of the query Q9 somewhere between the optimization times of the queries Q5 and Q7.

Chapter 5

Group Pruning

Group pruning has been pioneered by Leonard Shapiro and his team [33, 37]. In a nutshell, group pruning prevents the optimization of groups when this is provably not necessary in order to find the optimal plan. Section 5.1 contains a more detailed definition of group pruning and Section 5.2 shows how group pruning has been implemented in Cascades. Section 5.3 derives a lower bound for the cost of groups. This lower bound can be used to increase the effectiveness of group pruning. Section 5.4, finally, presents the TPC-D benchmark results. These results will demonstrate the effectiveness of group pruning. It will be shown that group pruning up to triples the speed of Cascades. Furthermore, it also reduces the memory consumption of Cascades because less groups have to be optimized. Thanks to this effect, the Cascades-based optimizer which I used throughout this thesis succeeded for the first time in optimizing the TPC-D query Q8!

5.1 Definitions

The philosophy of *group pruning* is not to optimize a group when this is provably not necessary for finding the optimal plan. Thus, as its name suggests, group pruning simply eliminates the need to optimize certain groups of the memo. Obviously, there must be some strategy to identify the groups that need not be optimized. This strategy is called the *group pruning strategy.* A group pruning strategy essentially is a condition on groups; if a group satisfies this condition, then it is guaranteed not to contribute to the optimal plan. Thus, such a group need not be optimized.

There might be a dozen different group pruning strategies. The most basic one probably is to underestimate the cost of each group and not to optimize it if this cost estimate exceeds the cost of the best yet found plan. There are more effective and complex group pruning strategies. One of them, namely the one that has been implemented in Cascades, will be described in the following section.

It is common to say that a group has been *pruned* if it has never been optimized. Please observe that a pruned group only contains a single multi-

expression, namely the one it was initialized with. Group pruning avoids to generate all the other multi-expressions the group could possibly contain. Therefore, group pruning can be very effective, indeed!

In preparation of the following section it is essential to precisely define the cost of a physical multi-expression. Please recall from Subsection 2.2.1 that a multi-expression represents a whole bunch of expressions. But these expressions need not agree on their costs! Therefore, it is not clear what the cost of a physical multi-expression really is. Cascades solves this problem with the following definition: The *cost of a physical multi-expression* M is the cost of that expression which is cheapest among all the expressions represented by M.

5.2 The Group Pruning Strategy of Cascades

In order to understand the group pruning strategy of Cascades it is indispensable to be familiar with the concept of a context. This concept has been introduced in Subsection 2.2.2, but is reviewed here for ease of reading. A context fixes the goal pursued in optimizing a group. This goal consists in finding the cheapest possible plan that has the properties demanded by the context. In particular, the optimization result must cost less than the *upper cost bound* of the context. Finally, please recall that optimizing a group fails if no plan can be found that satisfies the context at hand.

One more preliminary remark is in order. It should be stressed that Cascades' group pruning strategy is very closely intertwined with the underlying cost model. In fact, a thorough and precise description of group pruning in Cascades has to be based on some cost model. Throughout this chapter, a *sequential cost model* [21] is assumed, but any other cost model could have been used instead. The advantage of the sequential cost model, is its simplicity. Indeed, the sequential cost model defines the cost of a plan to be the sum of its operator costs. Obviously, the sequential cost model is suited for modeling single processors.

Next, let's see how group pruning works in Cascades. To this end, consider a group G that is optimized for some context C. In particular, consider the physical multi-expression P, which we assume to be contained in G. Furthermore, let's agree upon the following nomenclature:

- Be *LocalCost* the processing cost caused by the operator of P.
- Be G_i, the i-th input group of P.
- Be *InputCost*[] an array with one entry for each input group G_i of G. After the group G_i has been optimized, the cost of the resulting winner is stored in *InputCost*[i]. Until then, an underestimate of this quantity is stored instead.
- Be $UCB(C)$ the upper cost bound of the context C.

Now, suppose the following condition becomes true after some (but not all) of the input groups G_i have been optimized:

$$\mathit{LocalCost} \ + \sum_i \mathit{InputCost}[i] \ > \ \mathit{UCB}(C) \tag{5.1}$$

Under these circumstances, the multi-expression P has obviously no chance of ever satisfying the context C. In consequence, there is no point in optimizing the outstanding input groups of P. Unfortunately, it is still possible that these input groups have to be optimized some time later. Therefore, deciding on which groups have been pruned and which ones have not can only be done after the optimization process has terminated.

Let's have a closer look at the cost underestimates which are stored in $\mathit{InputCost}[i]$ until the exact values become available. It is easy to see that Cascades' group pruning strategy is getting all the more effective the higher these underestimates are. Therefore, it is important for Cascades to dispose of good cost underestimates. How such cost underestimates can be derived will be discussed in Section 5.3.

Please observe next that the upper cost bound $\mathit{UCB}(C)$ also influences how effective Cascades' group pruning strategy is. A lower upper cost bound is obviously better. Therefore, the following two techniques have been implemented in Cascades to derive as small upper cost bounds as possible:

Intra-Group Improvement: Reconsider the physical multi-expression P and suppose that optimizing it is not prematurely aborted because of condition (5.1) becoming true. Then, all the input groups of P are optimized and the cost and synthesized physical properties of P are calculated. Cascades now tries to decrease the upper cost bound of the current context C. This, however, is only feasible if P satisfies the context C. In this case, the inequality $\text{cost}(P) \leq \mathit{UCB}(C)$ holds. Cascades thereupon sets the upper cost bound of C equal to the cost of P. This step is correct since P constitutes a potential winner for group G and context C. Therefore, it is pointless to generate multi-expressions which are more expensive than P. Setting $\mathit{UCB}(C)$ equal to $\text{cost}(P)$ formalizes this idea: The subsequent optimization process shall be dedicated to finding physical multi-expressions which are less expensive than P.

Inter-Group Improvement: Once again, consider the physical multi-expression P. In particular, direct your attention on the i-th input group of P. Before this input group can be optimized, it is indispensable to determine an appropriate context. This context fixes the goal pursued in optimizing the input group. As is known, the context dictates an upper cost bound which the optimization result must not exceed. To make this upper cost bound as small as possible, it is set equal to

$$\mathit{UCB}(C) \ - \ \left(\mathit{LocalCost} \ + \sum_{j \neq i} \mathit{InputCost}[j] \right)$$

Let's put the two techniques of intra- and inter-group improvement into context. To this end, let's consider the entire optimization process. At its very beginning there is only a single optimization task whose purpose is to optimize the top-group of the given query. The context of this task is such as to have an infinite upper cost bound. This reflects Cascades' goal to find the cheapest possible plan, no matter how expensive it is. The important thing to note here is that inter-group improvement is of no help when the upper cost bound is infinite. Fortunately, an infinite upper cost bound does not interfere with the technique of intra-group improvement. In fact, intra-group improvement gets rid of the infinite upper cost bound as soon as the first physical multi-expression satisfying the context has been found. Thereupon, inter-group improvement becomes effective, too, and derives even smaller upper cost bounds. Obviously the two techniques work very well together.

Intra-group improvement works particularly well when the implementation rules have got higher promise values than the transformation rules. In this case, the implementation rules are applied before the transformation rules, which results in Cascades generating and costing the first plans right after the optimization process has begun. Intra-group improvement immediately takes advantage of these early plans and effectively lowers the upper cost bound of the respective context.

In Cascades, the group pruning code is completely confined to the method OPT_INPUTS::perform(). This shouldn't be surprising, since that is where the optimization of input groups is triggered. The Figures 5.1 and 5.2 finally show the new implementation of the method OPT_INPUTS::perform(). The new passages which implement the group pruning have been marked to facilitate the comparison between the old and new versions of OPT_INPUTS::perform().

5.3 A Lower Bound for Groups

As explained in the last section, Cascades sometimes needs to underestimate the cost of a winner. These underestimates are plugged into the array *InputCost*[] until the exact values become available. Cascades uses *lower bounds* in order to underestimate the cost of winners. The goal of the present section is to derive the formula that Cascades uses for calculating lower bounds.

By definition, a *lower bound for the group* G is just any value which is less than the cost of the cheapest plan in G. In consequence, no matter what context the group G is optimized for, the cost of the resulting winner will always exceed the lower bound of the group. Therefore, lower bounds really underestimate the cost of winners.

The remainder of this section is dedicated to deriving a lower bound for groups. To this end, consider a group G and suppose its schema contains attributes from the base relations $R_1, \ldots, R_n$. Furthermore, let's agree on the following definitions:

– $|G|$ denotes the cardinality of the group G.

```
CONTEXT adapt_context(context, phys_op, i);
// Returns the context that must be used to optimize the 'i'-th
// input-group of 'phys_op'.

void OPT_INPUTS::perform()
// Remember: 'context' and 'phys_mult_ex' are class attributes!
// Note: 'i' is a class attribute that is set to -1 by the constructor.
{    phys_op := operator of phys_mult_ex;

     if i = −1 then                              // NEW!!  NEW!!  NEW!!
          // The class attributes 'LocalCost' and 'InputCosts[]' are initialized
          // on the first (and no other) execution of this method.
          LocalCost := Cost caused by the operator phys_op;
          for j := 0 to arity(phys_op) − 1 do
               InpCtxt := adapt_context(context, phys_op, j);
               grp := the j-th input-group of phys_op;
               Search the winner's circle of group grp for a multi-expression
               satisfying the context InpCtxt;
               if a winner has been found then
                    InputCosts[j] := Cost of this winner;
               elsif no winner exists then
                    return;
               else
                    // A winner exists, but hasn't been found yet. Therefore,
                    // an underestimate of its actual cost will have to do:
                    InputCosts[j] := LowerBound(grp);
               endif;
          endfor;
     endif;

     if i > 0 then
          InpCtxt := adapt_context(context, phys_op, i);
          grp := the i-th input-group of phys_op;
          // 'grp' has just been optimized for context 'InpCtxt'!
          Search grp for a physical multi-expression M satisfying InpCtxt;
          if successful then  Add M to the winner's circle of group grp;
                      else  return;  endif;
          InputCost[i] := Cost of M;               // NEW!!  NEW!!  NEW!!
     endif;

     if LocalCost + Σ_j InputCost[j]  > UCB(context) then
          return;                                  // NEW!!  NEW!!  NEW!!
     endif;
```

Figure 5.1: Group pruning in OPT_INPUTS::perform()

```
    inc(i);
    if i < (arity of phys_op) then
        InpCtxt := adapt_context(context, phys_op, i);
        UCB(InpCtxt) := UCB(context) -                  // NEW!!  NEW!!
                        ( LocalCost + Σ_{j≠i} InputCost[j] );
        task_list→push( this );    // Re-schedule the present task!
        task_list→push( new OPT_GROUP(InpCtxt,
                            the i-th input-group of phys_op) );
    else
        Calculate the cost and synthesized physical properties of
        phys_mult_ex;
        if phys_mult_ex satisfies context then          // NEW!!  NEW!!
            UCB(context) := cost(phys_mult_ex);
        endif;
    endif;
}; // OPT_INPUTS::perform()
```

Figure 5.2: Group pruning in OPT_INPUTS::perform(), continued

- For any attribute X, let cucard(X) stands for the column unique cardinality of X [25, 35].
- For each relation R_i, $R_i \in \{R_1, \ldots, R_n\}$, let cucard($R_i$) denote the maximum value of cucard($R_i.X$), with the maximum ranging over all attributes $R_i.X$ in the schema of G. The cucard($R_i.X$) values of this definition refer to the column unique cardinalities at the group G, **not** at the base relation R_i. In this place, Cascades depends on the model to provide column unique cardinalities among its synthesized logical properties. That way, column unique cardinality estimates become available for each group of the memo. It goes without saying that the lower bound formula of this section cannot be used with a model that does not feature column unique cardinalities among its synthesized logical properties!
- Be $TOUCHCOPY$ a constant such that $TOUCHCOPY \times card$ is a lower bound for the cost of any join whose output-cardinality is $card$. Note that this lower bound property is independent of whether the join is a MERGE_JOIN, a HASH_JOIN, or whatsoever. In any case, the join must be more expensive than $TOUCHCOPY \times card$!
- Be $FETCH$ the amortized cost of reading one byte from disk.

The $FETCH$ value obviously describes a hardware property of the computer which is running Cascades. Therefore, it is easy to determine this value. Finding a suitable $TOUCHCOPY$ value, on the other hand, is more tricky. Essentially, this is an ad hoc activity which requires some feeling. A very simple approach

would be to set *TOUCHCOPY* equal to two times the amortized cost of passing a tuple from one operator to another plus the amortized cost of comparing two tuples. Another approach is to study the different physical JOIN operators of the cost model and to derive a *TOUCHCOPY* value from their respective cost formulas. Which approach is best, generally depends on the particularities of the cost model at hand.

Finally, without loss of generality, the condition $\text{cucard}(R_1) \leq \ldots \leq \text{cucard}(R_n)$ is assumed to hold. Please keep this assumption in mind while reading on.

Now the scene is set to derive a lower bound for the group G. This lower bound is the sum of two components: A lower bound on the join cost, and a lower bound on the disk access cost. The *lower bound on the join cost* underestimates the cost of joining the relations $R_1, \ldots, R_n$. The *lower bound on the disk access cost* underestimates the cost of reading the relations $R_1, \ldots, R_n$ from disk. Thus, the lower bound for the group G only takes account of those costs that stem from JOIN or SCAN operators. A more accurate lower bound would also consider the costs caused by other operators, such as SELECT, PROJECT, and so on. This, however, is highly complicated and therefore left to further research.

The following quantity obviously is a lower bound on the disk access cost:

$$\left(\sum_{i=1}^{n} \text{cucard}(R_i) \times \text{byte_size}(R_i)\right) \times \mathit{FETCH} \tag{5.2}$$

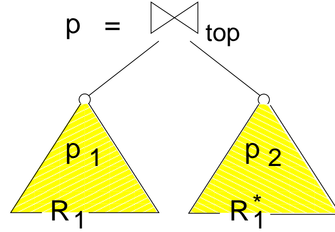
Finding a lower bound on the join cost is more difficult. The problem is that all possible ways of joining the relations $R_1, \ldots, R_n$ must be considered somehow. Otherwise, one cannot be sure that there is no cheaper way of joining the relations. The following lemma shows how this quantification over all possible join orders can be handled.

Lemma 5.1 *For any plan p in G, the cost of the join operators in p is at least*

$$\left(|G| + \sum_{i=2}^{n-1} \mathit{cucard}(R_i)\right) \times \mathit{TOUCHCOPY}. \tag{5.3}$$

Proof : Be p an arbitrary plan of the group G. I will show that p's join operators really cost more than the quantity of formula (5.3). To this end, note that p compulsorily contains $n-1$ join operators. Be $\{\bowtie_1, \ldots, \bowtie_{n-1}\}$ the set of these join operators. Then, there is a surjective function $\varphi : \{\bowtie_1, \ldots, \bowtie_{n-1}\} \longrightarrow \{R_2, \ldots, R_n\}$, such that $\text{schema}(\bowtie_i) \cap \varphi(\bowtie_i) \neq \varnothing \; \forall i$, i.e. the schema of $\bowtie_i$ contains attributes from the base relation $\varphi(\bowtie_i)$. Figure 5.3 sketches how the function φ can be found. (By the way, the reason for excluding R_1 and no other relation from the range of φ is that its cucard value has been assumed to be minimal among all the $\text{cucard}(R_i)$ values.)

The following observation is central to the proof: The output relation of join $\bowtie_i$ contains $\text{cucard}(\varphi(\bowtie_i))$ tuples or more. Thus, a lower bound for the



Without loss of generality:

- Be R_1 a leave of sub-plan p_1.
- Be R_1^*, $R_1^* \in \{R_1, \ldots, R_n\}$, any leave of the sub-plan p_2.

Then, the function φ is constructed recursively:

(1) Find the corresponding functions
$\varphi_1: \{\bowtie_i \mid \bowtie_i \in p_1\} \rightarrow \{R_i \mid R_i \text{ leaf of } p_1\} \setminus \{R_1\}$, and
$\varphi_2: \{\bowtie_j \mid \bowtie_j \in p_2\} \rightarrow \{R_j \mid R_j \text{ leaf of } p_2\} \setminus \{R_1^*\}$
for the sub-plans p_1 and p_2.

(2) Define:
$\varphi(\bowtie_{top}) := R_1^*$.
$\varphi(\bowtie_i) := \varphi_1(\bowtie_i), \forall \bowtie_i \in p_1$.
$\varphi(\bowtie_j) := \varphi_2(\bowtie_j), \forall \bowtie_j \in p_2$.

Figure 5.3: Existence of the function φ

cost of join $\bowtie_i$ is $TOUCHCOPY \times \text{cucard}(\varphi(\bowtie_i))$. Furthermore, the sum of these quantities for $i = 1, \ldots, n-1$ yields a lower bound for the cost of all the join operators in p. This lower bound can be further improved. To this end, be $\bowtie_{top}$ the top join of p. Then $|G| \times TOUCHCOPY$ generally constitutes a better lower bound than $\text{cucard}(\varphi(\bowtie_{top})) \times TOUCHCOPY$. This leads to the following lower bound for the cost of joining:

$$\left(|G| + \sum_{i=1}^{n-1} \text{cucard}(\varphi(\bowtie_i)) - \text{cucard}(\varphi(\bowtie_{top})) \right) \times TOUCHCOPY. \tag{5.4}$$

Formula (5.4) can be further simplified. First of all, the equation $\sum_{i=1}^{n-1} \text{cucard}(\varphi(\bowtie_i)) = \sum_{i=2}^{n} \text{cucard}(R_i)$ holds since φ is surjective. Furthermore, the inequality $\text{cucard}(\varphi(\bowtie_{top})) \leq \text{cucard}(R_n)$ is implied by the assumption $\text{cucard}(R_1) \leq \ldots \leq \text{cucard}(R_n)$ on page 60. It follows that the quantity of formula (5.4) is equal to, or larger than the quantity of formula (5.3). This completes the proof.

□

Combining formula (5.2) and Lemma 5.1 yields the following lower bound for the group G:

$$\begin{aligned}\text{LowerBound}(G) &= \left(\sum_{i=1}^{n-0} \text{cucard}(R_i) \times \text{byte_size}(R_i)\right) \times \mathit{FETCH} \\ &+ \left(|G| + \sum_{i=2}^{n-1} \text{cucard}(R_i)\right) \times \mathit{TOUCHCOPY}\end{aligned}$$

This lower bound is only correct for $n \geq 2$. For $n = 1$, there is no need for a lower bound, since the cost for reading a relation from disk can be calculated exactly.

5.4 Results

Table 5.1 merges the Tables 4.1 and 4.2 from Section 4.4, and adds a new row for group pruning. Needless to say that the depicted optimization times were measured under the same circumstances as in Section 4.4. Note, however, that the table contains measurements from two different machines. These are the same machines that have already been introduced in Section 4.4.

Let's have a closer look at Table 5.1. It is easy to see that group pruning almost triples the speed of Cascades for the more complex queries such as Q5 and Q9. The smaller, simpler queries, on the other hand, benefit less from group pruning. Furthermore, it is striking that query Q8 could be optimized for the first time. The reason for this is that group pruning saves quite a bit of memory, since less groups have to be optimized. By the way, the immense optimization time of query Q8 is due to paging.

	UltraSPARC-II, 295 MHz			microSPARC-II, 110 MHz				
	Q5	Q8	Q9	Q1	Q3	Q7	Q10	Q13
Initially	1400s	–	38s	1s	2.2s	6.5s	6.6s	0.5s
Dynamic arrays	1310s	–	35s	1s	2.2s	6.3s	6.3s	0.5s
New binding algorithm	945s	–	28s	1s	2.2s	6.0s	6.1s	0.5s
New winner's circle	12.5s	–	5.5s	1s	1.4s	3.8s	4.8s	0.5s
Group pruning	3.3s	1000s	2.8s	1s	1.3s	2.9s	3.8s	0.5s

Table 5.1: Optimization times for TPC-D queries

The paper [33] contains a more detailed performance analysis of group pruning. Very briefly, [33] explains that group pruning is particularly effective for chain queries [28]. Its benefit for star queries [28], on the other hand, is less pronounced. For more details, please refer to the cited papers.

Chapter 6

Conclusion and Future Work

Since the first release of Cascades has become available in 1994, it has constantly been enhanced by Professor Shapiro from the Portland State University. The efficiency increasing measures described in the Chapters 4 and 5 also originate in his research. Currently, Professor Shapiro is experimenting with new techniques that are potentially beneficial to the efficiency of Cascades. The present chapter will sketch some of these techniques. Furthermore, first results regarding their usefulness will be given. Here's a more detailed description of the topics covered in this chapter:

- Section 6.1 introduces global epsilon pruning. This is a heuristic pruning technique that speeds up the optimization process at the risk of possibly missing the optimal plan.

- Section 6.2 describes an effort to accelerate Cascades by improving its dynamic memory management.

- Section 6.3 presents first steps towards reducing Cascades' memory requirements. Research in this direction is important because it will help reduce the risk of paging, which completely paralyzes Cascades.

- Section 6.4, finally, summarizes the whole thesis. Based on this summary, a more integral way of comparing top-down and bottom-up optimizers is suggested.

6.1 Global Epsilon Pruning

Global epsilon pruning is motivated by the following observation: During the optimization process, it is generally not very promising to spend lots of time optimizing sub-queries that contribute little to the overall cost of the final plan. Instead, optimizing a sub-query should be stopped as soon as the cost of the best yet found plan drops below some threshold. This plan can be considered sufficiently good, given the fact that it only accounts for a marginal fraction of

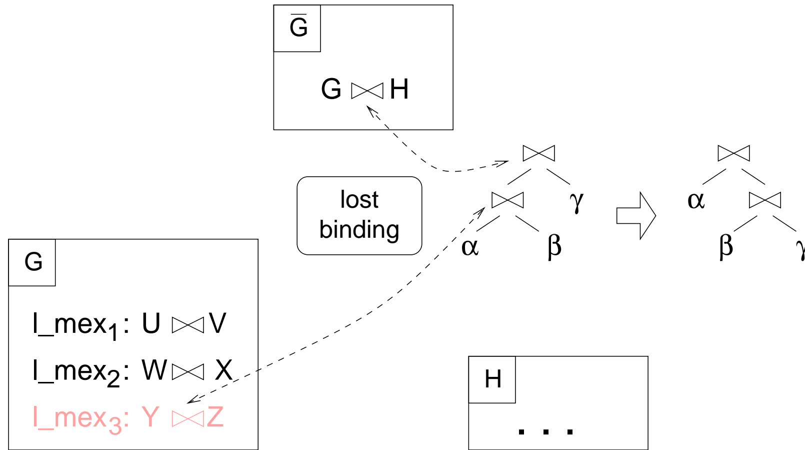


Figure 6.1: Global epsilon pruning has got undesired side effects

the total cost. The hope is that the resulting savings in optimization time are big, whereas the decline in plan-quality is relatively small.

The implementation of global epsilon pruning is very simple: Before optimizing a query starts, a (global) epsilon value ε is determined. How this can be done will be discussed in a moment. First, let's consider how this epsilon value is used to prune the search space. The strategy is to stop optimizing a group as soon as a plan is found that is less expensive than ε. This plan is considered sufficiently good, and becomes the winner for the context and group at hand. No more effort is put into finding a potentially better plan.

There's no real theory for finding an appropriate epsilon value. Two methods, however, have become popular. The first method uses an heuristic pre-optimizer to estimate the cost of the optimal plan. Then ε is set to some fraction of this cost. An alternative method takes the desired maximum cost and takes some fraction of this value for ε. Epsilon values always are magic numbers.

Global epsilon pruning has got three serious problems, which constitute the reason for which it has not been implemented in Cascades:

Loss of Optimality: Unlike group pruning, global epsilon pruning might fail to find the optimal plan!

Global Repercussions: Consider the scenario shown in Figure 6.1. The group G comprises 3 logically equivalent multi-expressions. Because of global epsilon pruning, however, only the first two multi-expressions are found. Then, optimizing G is stopped because a plan is found whose cost is less than ε. Later, when the group $\overline{G}$ is optimized, all the rule bindings that involve the multi-expression l_mex_3 are not found and their respective after-patterns are not inserted into the memo. In consequence, some multi-expressions of the group $\overline{G}$ are not found. This is tantamount to saying that some of the multi-expressions in $\overline{G}$ are pruned away! Furthermore, this kind of *undesired* pruning propagates upwards in the hierarchy

of groups. Therefore, the impact of global epsilon pruning is not confined to the groups it is applied to. Instead, global epsilon pruning has got undesired repercussions on the whole optimization process.

Unpredictability: It is natural to expect the optimization time to decrease monotonously as the epsilon value grows. In reality, however, the optimization time behaves as shown in Figure 6.2. This behavior will be explained by means of an example: Consider a group G whose only multi-expression is $\sigma(H)$. For simplicity, let's suppose the σ operator has a constant cost of 20 cost units.

Recall that Cascades first triggers the optimization of the group H, and subsequently starts applying rules on G. Suppose that optimizing H without global epsilon pruning generates a sequence of 7 plans whose costs are

95cu, 90cu, 85cu, 60cu, 45cu, 10cu, and 8cu (cu $\equiv$ **c**ost **u**nits).

In contrast, for global epsilon pruning with $\varepsilon = 40$cu ($\widehat{\varepsilon} = 50$cu), only 6 plans (5 plans) are generated before optimizing the group H is stopped. Thereafter, the group G is optimized. For $\varepsilon = 40$cu, the very first plan of G is a winner since its cost is $10\text{cu} + 20\text{cu} = 30\text{cu}$, which is less than $\varepsilon = 40$cu. Thus, for $\varepsilon = 40$cu, only $6 + 1 = 7$ plans are generated before a winner is found. For $\widehat{\varepsilon} = 50$cu, on the other hand, the first plan of G costs $45\text{cu} + 20\text{cu} = 65\text{cu}$ and is not a winner. Suppose 3 more plans have to be generated before a winner for the group G is found. Then, a total of $6 + 3 = 9$ plans are generated, which is two more plans than in the case of $\varepsilon = 40$cu!

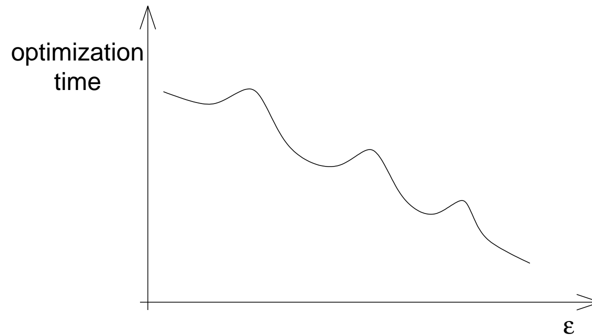


Figure 6.2: A greater epsilon will not always speed up the optimization process

Obviously, there is still more research needed to derive a useful pruning technique from global epsilon pruning. At the moment, however, the disadvantages of global epsilon pruning still outweigh the benefits. Therefore, global epsilon pruning has not been implemented in Cascades.

6.2 Memory Management

In Cascades, virtually all data is kept on the heap. No matter whether you consider task objects, group objects, or whatever, it's all on the heap! Therefore, the efficiency of the dynamic memory management is of utmost importance to Cascades. Every improvement in this respect will be beneficial to Cascades.

In C++, the default versions of `new` and `delete` are perfectly adequate for general-purpose memory management. However, their wide applicability inevitably leaves room for some improvements in their speed and memory utilization. This is especially true for applications like Cascades that dynamically allocate a large number of small objects.

Matters such as coping with memory requests of arbitrary sizes or dealing with machine-dependent alignment restrictions are responsible for the reduced efficiency of the default `new` and `delete` operators. Fortunately, C++ offers the possibility to implement customized versions of the `new` and `delete` operators. Professor Shapiro has used this possibility to experiment with a technique known as *memory pooling*.

The idea of memory pooling is simple: For each class, a class-specific version of the operators `new` and `delete` is implemented. The class-specific `new` operator allocates memory in large chunks using a low-level function such as `malloc`. This chunk of memory is used to maintain a pool of memory blocks, each block being just large enough to accommodate one object of the respective class. Later, when `new` is called, one block is removed from the pool; `delete`, finally, reinserts the block into the pool. Both operations are very fast since they only require some pointers to be reset.

Up to the present, memory pooling has only lead to a modest 10% decrease in the optimization time, but the code has still to be tuned. For these reasons, memory pooling has not been taken over into Cascades.

6.3 Memory Requirements

At present, Cascades has got one big problem: It easily runs out of main memory. The consequence is that paging sets in and the computation becomes seemingly endless. The present section will first explain where Cascades' excessive memory consumption comes from. Thereupon, two possible countermeasures will be sketched.

Let's start with the reason for Cascades' excessive memory consumption. To this end, consider a group that has just been optimized. Note that all the multi-expressions of the group have to stay in memory until the very end of the optimization process. Not a single of the group's multi-expressions can be discarded beforehand, since a parent group might need it later on. (Note the similarity to the situation of Figure 6.1!) The result is that an exponential number of multi-expressions is generated and stored in the course of optimization. Thus, it is no surprise that memory becomes a problem for big queries.

There are two possible solutions to Cascades' memory problem. The first is to limit the harm done by paging. This means to increase the locality of Cascades' memory access schema. Indeed, if Cascades was perfectly local, then paging would be no performance issue. At present, however, memory access is completely random in Cascades. How this can be changed is currently examined by Professor Shapiro at the Portland State University.

The second possible solution to Cascades' memory problem is to use less memory. To this end, the concept of *interesting orders* [7] has been applied to Cascades. Interesting orders have been pioneered by bottom-up optimizers. They describe those physical properties of a group that might be needed by a parent groups. Thus, interesting orders propagate top-down.

To see how interesting orders help save memory, let us consider a group that has just been optimized. Then, the interesting orders of this group make it possible to "filter" the physical multi-expressions of the group: For each interesting order I, only the cheapest physical multi-expression having the physical properties described by I is kept in memory. All the other physical multi-expressions are discarded. Note that interesting orders do not interfere with logical multi-expressions. They have to be kept in memory, just as before.

As just explained, interesting orders only eliminate the need to store certain physical multi-expressions. Logical multi-expressions are not affected by interesting orders. Nevertheless, the memory consumption of Cascades is expected to decrease quite substantially as a result of interesting orders. This expectation is based on the fact that a group generally contains two to three times more physical than logical multi-expressions. Unfortunately, at the time of this writing, I do not dispose of any measurements that could verify the usefulness of interesting orders for Cascades.

6.4 Conclusion

To me, there is not one all-embracing conclusion from this thesis. I rather feel that each of the main chapters teaches one lesson about top-down optimizers. Here is my view of what can be learned from the different chapters:

Top-Down Optimization in a Nutshell:

Chapters 2 & 4: The search engine of a top-down optimizer is an extremely complex piece of software. Indeed, most experts will agree that the code of a top-down optimizer is more difficult to understand than the code of a bottom-up optimizer.

Chapter 3: Finding duplicate-free rule-sets with a reasonably low search space complexity is the central problem in extending a top-down optimizer.

Chapter 5: Top-down optimizers lend themselves very well to group pruning. This technique is particularly effective with chain queries.

Chapter 6: The most pressing challenge of top-down optimizers is to reduce their excessive memory consumption.

I consider the above four lessons to be particularly important because they give the impetus to a more complete comparison between top-down and bottom-up optimizers. Traditionally, the two optimizer types have always been compared along the dimension of extensibility. However, a more complete comparison should take other factors into consideration, too. Table 6.1 shows my view of how top-down optimizers compare to bottom-up optimizers.

	Bottom-Up Optimizers	**Top-Down Optimizers**
Extensibility	It is not yet clear which optimizer type is more extensible!	
Time Efficiency	Traditionally superior.	Strongly catching up!
Space Efficiency	Satisfactory.	Poor, but improvement is possible.
Software Complexity	High.	The search engine is *extremely* complex!

Table 6.1: Top-down versus bottom-up

Appendix A

Failures in Tuning Cascades

Columbia [33], the successor of Cascades, claims to derive its superior efficiency from a couple of implementation tricks [37]. It was part of my research in the efficiency of top-down optimizers to evaluate each of these implementation tricks. Some of them turned out to be very effective, but others proved to be useless. The effective tricks have already been documented in the Chapters 4 and 5. This chapter is about the following two tricks that didn't result in any measurable reduction in the execution time of Cascades:

- It is a frequent operation in Cascades to search the memo for a given logical multi-expression. Therefore it seemed promising to implement this operation as efficiently as possible. Section A.1 describes such an implementation which is based on hashing. Unfortunately, Cascades did not run any faster in response to the new implementation. Some experimenting with different algorithms for searching the memo finally revealed that this operation has practically no impact on the speed of Cascades.

- Section A.2 describes a more memory-efficient representation of multi-expressions. Even though this mainly saves memory, it was expected to also save computation time. This expectation was based on the fact that allocating and deallocation memory on the heap are computationally expensive operations. Unfortunately, the resulting savings proved to be insignificant.

A.1 Searching the Memo

This section reconsiders the function search_memo() which is shown in Figure 2.8 on page 20. This function checks if a given logical multi-expression is contained in the memo. Making the function search_memo() run as fast as possible was considered important because it is called several times for every generalized multi-expression that is inserted into the memo.

Figure 2.8 shows that the old implementation of search_memo() linearly scans through the memo to check whether it contains a given multi-expression

or not. This actually is a simplification. In reality, search_memo() employs a very simple hash algorithm. This hash algorithm uses a hash table of fixed size, and implements each hash bucket as a linked list. The main weakness of the hash algorithm is its hash function, which puts almost all the multi-expressions into the very same hash bucket. In consequence, the new implementation of search_memo() tries to improve the hash function. Everything else is left as it is.

The rest of this section is dedicated to the new hash function. First of all, the new hash function uses all three components of a multi-expression (i.e. the name of its operator, the parameters of its operator, and the input groups of its operator) to compute a hash value. That brings up the question of how the hash value is derived from these three components. Most traditional hash functions would randomize each component and XOR the results. This approach, however, was rejected since randomizing was considered a too expensive operation. Instead of randomizing, the function lookup2() is employed. lookup2() is a modification of a function that was originally proposed by Bob Jenkins [22]. Most importantly, the function lookup2() possesses three essential features:

- It only uses very simple and fast bit operations to calculate its return value. Therefore lookup2() is very efficient compared to many randomization algorithms.
- It makes every bit of its return value dependent on every bit of its input key. In consequence, different input keys are rather likely to yield different return values.
- It takes a second input value aside from the key. This input value is a seed that can be used to parameterize the function. Cascades employs the seed to successively incorporate all three components of a given multi-expression into the final hash value. This is illustrated in Figure A.1.

The algorithm of the function lookup2() is rather tricky and therefore left out here. The interested reader, however, can find it in the source code of Columbia (cf. file "supp.cpp").

```
integer hash(mult_ex)
// Returns the hash value for the multi-expression 'mult_ex'.
{    op := operator( mult_ex );
     seed_1 := lookup2( name of op, 0 );
     seed_2 := lookup2( parameters of op, seed_1 );
     hash_val := lookup2( input groups of op, seed_2 );
     return(hash_val mod table_size);
}    // hash()
```

Figure A.1: Pseudo-code for the use of lookup2()

A.2 A More Succinct Representation for Multi-Expressions

Cascades uses objects of the class EXPR_LIST (cf. Figure A.2) to represent multi-expressions. Since multi-expressions occupy the main part of the search space memory, it is very critical to make this data structure as succinct as possible. It is easy to see that the last four attributes of the class EXPR_LIST are not needed. Just notice that the arity of the operator can be gotten from the operator itself. Therefore, there is no need to cache the operator's arity in each multi-expression. Furthermore, the other three attributes, i.e. task_no, rewritten, and local_cost, were intended to support functions that have never been implemented. Thus, these attributes can be culled, too.

```
class EXPR_LIST : public OPT_OBJECT
{
  private :
    GROUP_NO      group_no;        // group the multi-expr. belongs to
    OP_ARG        * op_arg;        // the operator
    GROUP_NO      * input_groups;  // the input groups

    COST          * cost;          // cost, only for phys. multi-expr.
    SYNTH_PHYS_PROP  * phys_prop;
                  // synth. phys. prop.; only for physical multi-expr.

    EXPR_LIST     * group_next;    // pointer to the next multi-expr. in
    EXPR_LIST     * bucket_next;   // the same group / hash bucket

    BIT_VECTOR    dont_fire;       // if bit is 1, don't fire that rule

    int           arity;           // cache arity of the operator
    int           task_no;         // for book keeping only
    RWSTATE       rewritten;       // not used
    COST          * local_cost;    // not used

  public :
    ...
}   // class EXPR_LIST
```

Figure A.2: The class EXPR_LIST (slightly simplified)

Please note that even though trimming the class EXPT_LIST did not speed up Cascades, it still constitutes a significant improvement over the former situation. In fact, removing the superfluous attributes reduces the memory con-

sumption of each multi-expression to $^3/_4$ of its former value[1]. The importance of this reduction cannot be overemphasized, since slashing their excessive memory consumption is the next big challenge top-down optimizers have to face!

[1] The attribute dont_fire is 16 bytes long.

Bibliography

[1] Keith Billings. A TPC-D Model for Database Query Optimization in Cascades. Master's thesis, Portland State University, 1997. URL: http://www.cs.pdx.edu/~kgb/t/title.shtml.

[2] J. Blakeley and N. Martin. Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis. In *Proceedings of the IEEE International Conference on Data Engineering*, 1990.

[3] G. Bozas, M. Jaedicke, A. Listl, B. Mitschang, A. Reiser, and S. Zimmermann. On Transforming a Sequential SQL–DBMS into a Parallel One: First Results and Experiences of the MIDAS Project. In *Proc. of the 2nd Int. Euro-Par Conference, Parallel Processing, Lyon, Springer-Verlag, Berlin*, 1996.

[4] Paul Brown. Implementing the Spirit of SQL-99. In *Proceedings of the 1999 ACM SIGMOD, Philadelphia, USA*, volume 28, 1999.

[5] P. Celis. The Query Optimizer in Tandem's ServerWare SQL Product. In *Proceedings of VLDB*, 1996.

[6] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, 1985.

[7] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Seattle, Washington, USA*, 1998.

[8] S. Chaudhuri and K. Shim. Including Group-By in Query Optimization. In *Proceedings of the 20th International VLDB Conference, Santiago, Chile*, 1994.

[9] S. Chaudhuri and K. Shim. An Overview of Cost-based Optimization of Queries with Aggregates. In *Bulletin of the Technical Committee on Data Engineering (IEEE)*, volume 18, 1995.

[10] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, 6 edition, 1995.

[11] C. J. Date and Hugh Darwen. *A Guide to the SQL Standard.* Addison-Wesley, 1993.

[12] R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Addison-Wesley, 2 edition, 1994.

[13] Michael Fleischhauer. Parallelisierung relationaler Datenbankanfragen in MIDAS. Master's thesis, Technische Universitaet Muenchen, 1997.

[14] Goetz Graefe. The Cascades Framework for Query Optimization. In *Bulletin of the Technical Committee on Data Engineering (IEEE)*, volume 18, 1995.

[15] Goetz Graefe. The Microsoft Relational Engine. In *Proceedings of Data Engineering Conference*, 1996.

[16] Goetz Graefe and D. DeWitt. The EXODUS Optimizer Generator. In *ACM SIGMOD Conference*, 1987.

[17] Goetz Graefe and W. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *IEEE Conference of Data Engineering, Wien*, 1993.

[18] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, 1990.

[19] L. Haas, W.Chang, and G. Lohman. Starburst Mid-Fight: As the Dust Clears. In *IEEE Transactions on Knowledge and Data Engineering*, 1990.

[20] I. J. Heath. Unacceptable File Operations in a Relational Database. In *Proceedings of the ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, California*, 1971.

[21] Matthias Hilbig. Entwicklung eines kostenbasierten Anfrageoptimierers fuer das relationale Datenbanksystem MIDAS. Master's thesis, Technische Universitaet Muenchen, 1998.

[22] Bob Jenkins. Hash Functions for Hash Table Lookup. In *Dr. Dobb's Journal*, 1997.

[23] Nicolai Josuttis. *Objektorientiertes Programmieren in C++.* Addison-Wesley, 1995.

[24] Dexter C. Kozen. *The Design and Analysis of Algorithms.* Springer Verlag, 1992.

[25] Kay Krueger-Barvels. Entwicklung eines regelbasierten Anfrageoptimierers fuer das relationale Datenbanksystem MIDAS. Master's thesis, Technische Universitaet Muenchen, 1998.

[26] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996.

[27] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *ACM SIGMOD Conference on the Management of Data*, 1997.

[28] K. Ono and G. M. Lohman. Measuring the Complexity of Join Enumeration in Query Optimization. In *Proceedings of the 16th VLDB Conference, Brisbane, Australia*, 1990.

[29] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[30] A. Pellenkoft, C. A. Galindo-Legaria, and M. Kersten. The Complexity of Transformation-Based Join Enumeration. In *Proceedings of the 23rd VLDB Conference, Athen, Griechenland*, 1997.

[31] P. Selinger, M. Astahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of SIGMOD*, 1979.

[32] Kim Shanley. Transaction Processing Performance Council. URL: http://www.tpc.org.

[33] Leonard Shapiro, David Maier, Keith Billings, Yubo Fan, Bennet Vance, Quan Wang, and Hsiao mi Wu. Group Pruning in the Columbia Optimizer. URL: http://www.cs.pdx.edu/~len/columbia.html.

[34] Susanne Stamp. Anfrageoptimierung von OQL mit dem Cascades Optimizer Framework. Master's thesis, Technische Universitaet Muenchen, 1996.

[35] A. Swami and K. B. Schiefer. On the Estimation of Join Result Sizes. In *Proceedings of Advances in Database Technology – EDBT, Cambridge, UK*, 1994.

[36] Bennet Vance. *Join-order Optimization with Cartesian Products.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1998.

[37] Yongwen Xu. Efficiency in the Columbia Database Query Optimizer. Master's thesis, Portland State University, 1998. URL: http://www.cs.pdx.edu/~len/columbia.html.

[38] S. Christodoulakis Y. E. Ioannidis. On the Propagation of Errors in the Size of Join Results. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Colorado, USA*, 1991.

[39] W. P. Yan and P.-A. Larson. Eager Aggregation and Lazy Aggregation. In *Proceedings of the 21st VLDB Conference, Zurich, Schweiz*, 1995.

[40] Weipeng P. Yan and Per-Ake Larson. Eager Aggregation and Lazy Aggregation. In *Prodeedings of the 21st VLDB Conference, Zurich, Switzerland*, 1995.

Index

Γ-formed, 40
$\bowtie$-formed, 40

After-pattern, 6
Amortized time, 47

Before-pattern, 6
Binding, 16–17, 48–49

Context, 15–16, 55

Duplicate-free rule-set, 22, 24, 37
Duplicates
  definition, 38
  duplicate-freeness, 38
  I-duplicates, 38
  X-duplicates, 38
Dynamic arrays, 47

Efficiency, 2, 51–52, 62, 66, 68
Expression, 3
  representation of ˜ , 15
Extensibility, 2, 5, 10, 44–45, 68

Generalized multi-expression, 17
Global epsilon pruning, 63
Group, 12–15
Group pruning, 54
Group pruning strategy, 54

Hashing, 70

Implementation rule, 16
Input cost, 55
Inter-group improvement, 56, 57
Interesting orders, 67
Intra-group improvement, 56, 57

Leaves, 10
Local cost, 55
Logical equivalence
  of expressions, 3
  of multi-expressions, 16
Lower bound for groups, 57, 61

Memo, 12–15
Memory pooling, 66
Model, 9
Multi-expression, 12–15, 71
  generalized, *see* Generalized multi-expression
  logical, 16
  physical, 16
  primary, 19, 38
  secondary, 19, 38

Operator
  logical, 3
  physical, 3
Operator tree, 3
  logical, 3
  physical, 3

Plan, 3
Property
  logical, 2
  physical, 2
  required, 11
  synthesized, 11

Query optimizer, 1, 4
  bottom-up, 4, 5
  cost-based, 4
  hard-coded, 5, 6
  purely heuristic, 4
  rule-based, 5, 6
  top-down, 4, 6

Rewriting rules, 6, 10
Rule mask, 22
Rule-set, 5, 6

Search engine, 9
Search space complexity, 32
Sequential cost model, 55
Signature
    of a group, 35
    of a multi-expression, 40
Syntactical equivalence, 21

Task object, 21
Transformation rule, 16

Upper cost bound, 16, 55

Winner's circle, 15–16, 49