Join-order Optimization with Cartesian Products

Bennet Vance
B.A., Yale University, 1976
M.S., Stanford University, 1981

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science and Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

January 1998

The dissertation "Join-order Optimization with Cartesian Products" by Bennet Vance has been examined and approved by the following Examination Committee:

David Maier
Professor
Thesis Research Adviser

James Hook
Associate Professor

Jonathan Walpole
Associate Professor

Len Shapiro
Professor
Portland State University

Acknowledgments

This dissertation owes a debt to many individuals. Dave Maier, my adviser, deserves special mention for his role in guiding and reviewing my work, and for bringing to this role the same clarity, perceptiveness, and patience that distinguish his teaching in the classroom. Jim Hook, Len Shapiro, and Jon Walpole also deserve credit for their considerable efforts as readers on my dissertation committee. I am grateful for their comments and encouragement, and for all that they have taught me over the years.

The following individuals gave me valuable feedback on earlier presentations of the material in this dissertation: Khalid Alnafjan, Roger Barga, Roberto Bayardo, César Galindo-Legaria, Goetz Graefe, Joe Hellerstein, Bala Iyer, Donald Kossmann, Guy Lohman, Bill McKenna, Guido Moerkotte, and Vijay Sarathy. For their comments, questions, conversation, and encouragement, I am deeply grateful.

Steve Otto generously gave of his time to explain to me the principle of Chained Local Optimization. Scott Daniels, Leo Fegaras, Gail Mitchell, and Stan Zdonik helped shape my thinking about query optimization. Scott Daniels and Jon Inouye gave me technical assistance in setting up my experiments and in typesetting this document. Rik Smoody contributed both moral and financial support to my research lab. Mike Carey has been helpful in numerous ways, not least with bibliographic questions. For all these forms of assistance, I am grateful.

Support for this work was provided in part by the Advanced Research Projects Agency, ARPA order number 18, monitored by the US Army Research Laboratory under contract DAAB-07-91-C-Q518, and by NSF grant IRI 91 18360. The support of these agencies is gratefully acknowledged.

Contents

List of Tables

List of Figures

Abstract

Join-order Optimization
with
Cartesian Products

Bennet Vance

Supervising Professor: David Maier

Join-order optimization plays a central role in the processing of relational database queries. This dissertation presents two new algorithms for join-order optimization: a deterministic, exhaustive-search algorithm, and a stochastic algorithm that is based on the deterministic one. The deterministic algorithm achieves new complexity bounds for exhaustive search in join-order optimization; and in timing tests, both algorithms are shown to run many times faster than their predecessors. In addition, these new, fast algorithms search a larger space of join orders than is customary in join-order optimization. Not only do they consider all the so-called *bushy* join orders, rather than just the *left-deep* ones, but—what is more unusual—they also consider all join orders that contain *Cartesian products.* The novel construction of these algorithms enables them to search a space including Cartesian products without paying the performance penalty that is conventionally associated with such a search.

Chapter 1

Introduction

This dissertation presents new results on the problem of *join-order optimization.* This introductory chapter gives an overview of the problem, and summarizes the claims and contributions of the present work. The introduction closes with a road map of the remainder of the dissertation.

1.1 The Problem

The problem of join-order optimization arises in the context of *relational query processing.* Recall that to retrieve information from a relational database, one ordinarily poses a *query* expressed in some variant of the language SQL (Structured Query Language) [11, 32]. Consider an SQL query of the form

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

where A, B, and C denote *relations*, and where the ellipsis (...) represents a predicate, or possibly a conjunction of many predicates. (Chapter 2 gives a more concrete example with additional detail; the present example is sketchy in the interest of brevity.) After parsing this query, a relational database management system might represent it internally as the *join* of A, B, and C, i.e., as the relational algebra expression $A \bowtie B \bowtie C$.

Now here is the problem. Because the join operator ($\bowtie$) is commutative and associative,

the join of A, B, and C may be written as

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

or as

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

or as

$$B \bowtie (A \bowtie C), \tag{1.3}$$

or indeed in any of a number of other ways—12 of them altogether. These 12 different expressions are *semantically equivalent* in the sense that they all evaluate to the same result. But viewed operationally, they are *not* equivalent; for example, $(A \bowtie B) \bowtie C$ suggests first joining A with B, and then joining that intermediate result with C, whereas $B \bowtie (A \bowtie C)$ suggests first joining A with C, and then joining B with that intermediate result. These operational differences are significant because the *computational cost* of evaluating one expression (measured in disk and CPU time) may be vastly different from that of evaluating another, semantically equivalent expression. To perform well, a database management system must make a judicious choice of expressions to use as the basis for query evaluation. The process of choosing from among the available alternatives is called *join-order optimization* (or simply *join optimization*).

There are, as noted, 12 alternatives in the case of a three-way join (i.e., a join involving three relations). In general, for an n-way join [29, 31, 54], the number of alternatives is

$$\frac{(2n-2)!}{(n-1)!}. \tag{1.4}$$

This quantity grows at an explosive, faster-than-exponential rate. The time complexity of join-order optimization, however, is not quite as unfavorable as formula (1.4) might lead one to expect; it is possible to search the space of alternatives exhaustively without examining each alternative separately from the others. Even so, the time complexity of join-order optimization remains exponential. Ibaraki and Kameda have shown the problem to be NP-complete [25].

1.2 Join-order Optimization in Practice

The intractability of join-order optimization leads to a dilemma: To optimize, or not to optimize? On the one hand, if one does *not* optimize, query evaluation may take an inordinately long time; in this case, we shall refer to the query evaluation as having a high *cost*. On the other hand, if one *does* optimize, it may turn out that the optimization itself takes an inordinately long time, quite possibly defeating the purpose of optimization. In the latter case, we shall refer to the optimization as being very *time-consuming*, or as entailing a large computational *effort*. Thus, we make a verbal distinction between the *time* or *effort* of optimization, and the *cost* of query evaluation, though it is actually *time* we are concerned about in either instance.

In practice, the effort required for join-order optimization typically becomes prohibitive when n (the number of relations being joined) reaches a value somewhere in the teens. Values of n in this range can arise when users submit SQL queries with large numbers of relations in the **FROM** clause, but they can also arise in other ways that users may not even be aware of. For example, queries that make use of *views* [11, 32] often generate hidden joins, as do queries with *path expressions* [8, 53, 62]. As databases become more complex, and as they incorporate additional facilities that automatically generate joins "underneath the covers," queries with large values of n are likely to become more and more common.

In the case of such queries, while the effort required for join-order optimization may be excessive, the consequences of forgoing optimization are likely to be equally unacceptable. The resolution of the dilemma is to compromise on what is meant by optimization. A literal reading of the word "optimization" would require a join-order optimizer to choose, from among the available alternatives, a join expression whose cost was *minimal*. Such expressions are sometimes called *exact solutions* to the optimization problem; they may also be referred to as *exact optima* or *true optima*. But a looser reading of the word "optimization" allows for the selection of an alternative whose cost is merely *low*, and not necessarily minimal. Such alternatives may be referred to as *approximate optima* or *near optima*. In general, an approximate optimum can be had for much less effort than an exact

one. The practical objective of join-order optimization is to strike a balance between the quality of the solution obtained, and the ease of obtaining it.

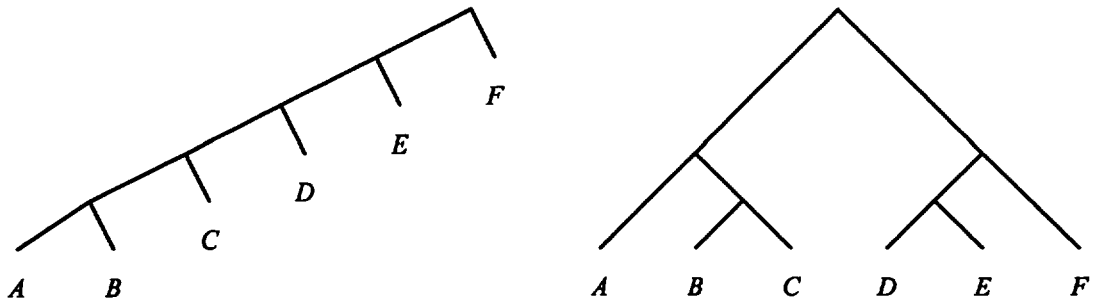
Optimizers employ a variety of tactics for reducing optimization effort. In this work, the following tactics shall concern us especially:

1. *Exclusion of Cartesian products.* As will be explained in greater detail in Chapter 2, some "joins" are actually Cartesian products, denoted by the operator $\times$. For example, consider an SQL query that includes A and B in the **FROM** clause, as in the query illustrated at the outset. Depending on what was in the query's **WHERE** clause, the join of A and B might turn out to be a Cartesian product, and in that case would ordinarily be written as $A \times B$, rather than as $A \bowtie B$.

   As a rule, Cartesian products entail very high evaluation costs; on this basis, most query optimizers will not even consider expressions that involve Cartesian products (unless they cannot be avoided). By excluding such expressions, a query optimizer reduces the size of its search space, and reduces optimization effort accordingly. But at the same time, it risks yielding a suboptimal solution in those cases where the true optimum contains a Cartesian product [45].

2. *Restriction of the search to left-deep join expressions.* Left-deep expressions have the form illustrated in Figure 1.1(a), with nesting of join operators occurring only in the left-hand inputs. The more general space of *bushy* expressions, of which Figure 1.1(b) is an instance, allows nesting in both the left-hand and right-hand inputs.

   The space of left-deep expressions grows far less quickly than the space of bushy expressions; given n input relations, there are only $n!$ different left-deep expressions, in contrast to the $((2n-2)!)/((n-1)!)$ bushy expressions reported in formula (1.4) above. On the other hand, the optimal bushy expression is sometimes far superior to the best left-deep one [45], and never inferior, since the bushy expressions subsume the left-deep ones.



$((((A \bowtie B) \bowtie C) \bowtie D) \bowtie E) \bowtie F$

(a) Left-deep

$(A \bowtie (B \bowtie C)) \bowtie ((D \bowtie E) \bowtie F)$

(b) Bushy

Figure 1.1: Left-deep and bushy expressions

3. *Use of stochastic search.* This tactic has a rather different character from the previous two. The previous two tactics focused on reducing the size of the search space, while the idea behind stochastic search is to explore the full "breadth" of the search space—however large it may be—without searching it exhaustively.

   Typically a stochastic search examines only a tiny fraction of the expressions in the given space. Needless to say, by leaving much of the space unexamined, such a search runs the risk of overlooking the truly optimal solutions.

In each instance, the guarantee of true optimality is sacrificed in the interest of making the optimization process more tractable. (Optimizers use many other effort-reducing tactics as well, some of which do *not* compromise optimality.)

Tactics 1–3 above need not be mutually exclusive. Much work on stochastic join-order optimization has focused on the space of left-deep join expressions [26, 56]; and regardless of any other tactics they employ, *nearly all* join-order optimizers exclude Cartesian products. In fact, in those instances where optimizers support consideration of Cartesian products at all, they do so only as an option, or only in restricted contexts; exclusion of Cartesian products has always been the default.

1.3 Claim and Synopsis of Dissertation

The claim of this dissertation is as follows: *There is no benefit in excluding Cartesian products in join-order optimization.* We make this claim in the context of bushy join-order optimization; it may well hold in the context of left-deep optimization as well, but the bushy case is more interesting, and, at least on the surface, more challenging.

The defense of this claim rests on theoretical and empirical analyses of two new join-order optimization algorithms presented in this work. The first of these algorithms performs an exhaustive search; the second, a stochastic search. Both algorithms explore the space of bushy expressions, and neither excludes Cartesian products.

The exposition takes the following shape. We begin by considering the question of what is involved in optimizing expressions that contain *only* Cartesian products, and no joins. We present an algorithm for exhaustively searching the space of bushy Cartesian products, and show that the time complexity of this algorithm is actually *lower* than that of the bushy exhaustive-search join-order optimizers described in the literature. Moreover, by observing the numerical behavior of the pertinent complexity measures, we predict that the Cartesian product optimization algorithm can be made to run very fast. Experiments on an implementation of the algorithm bear out this prediction.

We then take the next step: to extend the Cartesian product optimization algorithm so that it accommodates join operators as well. Such is the structure of the algorithm, and such the nature of the extension, that the accommodation of joins need not impair the optimizer's speed; nor does it necessitate any restriction of the search space. As the algorithm makes no fundamental distinction between join and Cartesian product operators, it chooses Cartesian products over joins whenever they are appropriate.

With a join-order optimizer now in hand, we move on to empirical evaluation of our method. Extensive timings show that our optimizer runs faster than previous join-order optimizers by *several orders of magnitude.* Such comparisons must be treated with caution, however; in some instances, the improvement can be partly attributed to the use of faster, more modern hardware, or to simplifying assumptions in our timing experiments. For these reasons the central claim of the dissertation narrowly focuses on inclusion of Cartesian

products, and not on optimizer speed. Nonetheless, the apparent large speed advantage of our method is suggestive to say the least—particularly as it appears to hold even against optimizers that were developed and benchmarked contemporaneously with our own.

To make the case that our method's inclusion of Cartesian products is not a liability, we must consider the effects of the simplifying assumptions in our experiments. One cannot just do away with simplifying assumptions, for there is no such thing as "the general case" in join-order optimization: All timings of join-order optimizers, and not just ours, necessarily deal with special cases, and make simplifying assumptions of one kind or another. (To date, the literature has not settled on any standard set of special cases and assumptions, and there is considerable variety in the way these issues are handled in different experiments.) To better understand the potential liabilities of our algorithm, we investigate a pair of metrics that we refer to as *cost-function execution counts.*

By studying cost-function execution counts, we find that, as originally presented, our method in fact *can* suffer as a result of its consideration of Cartesian products. But the evidence of this effect also suggests corrective measures, which dramatically reduce the effect. With the corrective measures in place, we see that as a rule, cost-function execution counts need not be significantly higher when Cartesian products are included in the search than when they are not. This result carries the implication that whether or not our optimizer retains a speed advantage under varying assumptions, it is most unlikely to become *slower* than other optimizers.

There remains an Achilles' heel in our argument in defense of including Cartesian products in the search. When the number of relations n becomes "large"—meaning somewhere in the teens, as noted above—all exhaustive-search optimizers become overwhelmed by the exponential complexity of join-order optimization. But there are special cases where join-order optimization *without* Cartesian products has merely *polynomial* complexity [45], whereas the complexity of join-order optimization *with* Cartesian products is *always exponential* [6, 49]. In these special cases, our exhaustive-search method becomes uncompetitive.

Yet we cling tenaciously to our claim that Cartesian products need not be excluded. All that is needed at larger n is a more powerful optimization technique than exhaustive

search; and so we introduce a stochastic join-order optimization technique to extend the power of our exhaustive-search method. Again we find that our stochastic technique outpaces previous stochastic techniques by several orders of magnitude, while obtaining solutions of extremely high quality. It performs especially well on the class of problems whose complexity is polynomial when Cartesian products are excluded; but it achieves this high level of performance *without excluding Cartesian products.*

Although the presentation of these results is set in a relational context, our techniques extend naturally to object-oriented databases as well; the relevant principles are sketched in the concluding commentary.

1.4 Contributions

In the course of defending the claim discussed above, this dissertation makes the following contributions to the understanding of join-order optimization:

- It demonstrates that the worst-case time complexity of the Starburst optimizer, as described by Ono and Lohman [44, 45], is $O(4^n)$.
- It gives the first detailed account of an exhaustive-search, bushy join-order optimization algorithm with a worst-case time complexity of $O(3^n)$, and with a worst-case space complexity of $O(2^n)$.
- It shows how this algorithm can be implemented with very low overhead.
- It presents, in detail, a stochastic join-order optimization algorithm that achieves extremely high-quality solutions in a small fraction of the time required by previous algorithms.
- It takes the first steps toward a systematic approach to the benchmarking of join-order optimization.

A few words of comment will help to clarify what is new, and what is not new, about our complexity results.

Ono and Lohman [44, 45] were the first to observe that join-order optimization has time complexity $O(3^n)$ in the worst case. But interestingly, in analyzing their own algorithm for the Starburst optimizer, they examined only the *number of join expressions* it considers, and not the complexity of the loop that *generates* those join expressions. When the generation loop is taken into account, the complexity of their pseudo-code proves to be $O(4^n)$, as we shall show.

In the time since the work of Ono and Lohman, there may well have been variations on their implementation that actually achieved $O(3^n)$ time complexity. The literature is inconclusive. Ganguly et al. [15] give $O(3^n)$ as the complexity of bushy join-order optimization, but again this complexity figure is based on the *number* of join expressions considered, not on an algorithmic analysis. No evidence is presented of an implementation that in fact achieves the stated complexity. As the Starburst example illustrates, it cannot be taken for granted that execution time is proportional to the number of cases considered.

The pseudo-code in the present work addresses these matters with care, leaving no doubt as to its $O(3^n)$ time complexity; in empirical trials we also verify that the claimed complexity is actually achieved. Very recent work by Pellenkoft, Galindo-Legaria, and Kersten [46] presents an alternative approach to join-order optimization with worst-case time complexity $O(3^n)$—though their approach has a higher *space* complexity than ours. We shall comment on these matters further in due course.

1.5 Road Map

The remaining chapters of this dissertation are as follows:

2 Background and Related Work introduces concepts and conventions pertaining to join-order optimization, discusses some of the difficulties inherent in the problem, and surveys previous approaches to the problem.

3 Cartesian Product Optimization introduces the *Blitzsplit* algorithm for Cartesian product optimization, and analyzes the complexity of the algorithm.

4 **Lightweight Implementation of Cartesian Product Optimization** shows how the abstractly presented algorithm of Chapter 3 can be given an extremely fast concrete realization.

5 **Support for Join Predicates** discusses the accommodation of join predicates in the Blitzsplit algorithm, which enables the algorithm to optimize joins, and not just Cartesian products.

6 **Performance Analysis** gives empirical results on the Blitzsplit algorithm's performance in join-order optimization, and analyzes those results.

7 **Pruning Cost Computations** draws on observations from the performance analysis to show how the Blitzsplit algorithm's performance can be improved through pruning of cost computations.

8 **A Stochastic Extension** builds on the foundation of the deterministic Blitzsplit algorithm, and describes the *Stochastic Bushwhack* algorithm for join-order optimization.

9 **Performance of the Stochastic Extension** gives empirical results on the Stochastic Bushwhack algorithm's performance in join-order optimization, which suggest a refinement to the Stochastic Bushwhack algorithm called the *Recursive Bushwhack* algorithm.

10 **Conclusion** summarizes the ground covered, discusses several open issues, and offers a few words of commentary.

Chapter 2

Background and Related Work

In the introduction, we saw the essence of the join-order optimization problem: Once a database query has been translated into an algebraic expression, there are likely to be many equivalent alternatives to that expression, and it is an optimizer's task to choose among them on the basis of expected evaluation cost. But that brief characterization of the problem overlooks the many complications and subtleties involved in join-order optimization.

In this chapter we examine the problem in more detail. We shall introduce various concepts and conventions of join-order optimization, and point out difficulties that one cannot readily overcome without making simplifying assumptions. We shall also discuss the approaches that have been taken to solving the problem in previous work.

We begin by reviewing some of the fundamentals of relational databases and the relational algebra. We go on to discuss the role of join-order optimization in the larger context of *query processing*. We then describe some of the technical difficulties that arise in join-order optimization in practice. Foremost among these difficulties is that of accurately estimating the evaluation cost of a query expression. We shall go into some detail in considering the nature of the estimation problem, and in describing the techniques that have been developed to deal with it. In the course of the discussion, we shall lay the conceptual groundwork for the remaining chapters of this dissertation.

2.1 Relational Databases

In this section and in the following two sections, we review some of the basics of relational databases and relational query processing. We begin here by describing the representation of data in relational databases and by introducing pertinent terminology.

2.1.1 Illustration of Basic Concepts

For illustration, Figure 2.1 presents a sample relational database that supports inventory management and customer billing for an auto-parts dealer. This database consists of six *relations* (also known as *tables*); each relation is a set of *tuples* (or *rows*), and each tuple consists of a collection of *attributes* (or *columns*). Relations lend themselves to tabular representation because the tuples within a given relation have a fixed number of attributes. The *names* of these attributes are fixed as well, and because they are fixed, they appear as column headings in the illustrated tables.

The **Customer** relation in the figure consists of two tuples, each representing a customer of the auto-parts store, and each containing a C_CUSTNO attribute and a C_NAME attribute. The illustrated **Customer** relation informs us that there is a customer named Kinbote, who has been assigned a customer number of 401, and another customer named Quilty, whose customer number is 402. The customer number plays the role of *primary key* for the relation; that is, each customer is assigned a unique customer number, and consequently the tuple in the **Customer** relation that gives information about a particular customer can be located using the customer number.

Note that the attribute names C_CUSTNO and C_NAME in the **Customer** relation both begin with the prefix "C_." The prefix serves as a reminder of the relation to which these attributes belong. Throughout this dissertation we shall follow the convention of prefixing each attribute name with the first letter of the corresponding relation name.

The **Employee** relation is analogous to the **Customer** relation, but gives information about the store's employees rather than its customers. In addition, the **Employee** relation has a third attribute E_MGR that gives the employee number of each employee's manager. Thus, employee number 3, Tom, is managed by employee number 5, namely Ray; and Ray,

Customer

C_CUSTNO	C_NAME
401	Kinbote
402	Quilty

Employee

E_EMPNO	E_NAME	E_MGR
3	Tom	5
5	Ray	5

Order

O_ORDERNO	O_CUSTNO	O_SOLDBY
1001	401	3
1002	402	3

LineItem

L_ORDERNO	L_PARTNO	L_QTY
1001	7007	2
1002	7007	1
1002	8008	8

Part

P_PARTNO	P_DESCR
7007	Head Lamp
8008	Spark Plug

Source

S_PARTNO	S_SUPPL
7007	Acme
8008	Acme
8008	Jolt
8008	Nuke
8008	Zap

Figure 2.1: A sample relational database

we see, is also managed by employee number 5—in other words, Ray is his own manager.

The **Order** relation records orders for parts. We see that order number 1001 was placed by customer number 401 (Kinbote), and was taken by (or "sold by") employee number 3 (Tom); order number 1002 was placed by customer number 402 (Quilty), and like the other order, was "sold by" employee number 3 (Tom). The O_ORDERNO attribute is the primary key of this relation, since it uniquely identifies each order. The O_CUSTNO and O_SOLDBY attributes, on the other hand, are *foreign keys*. A foreign key may be thought of as a kind of *pointer* or *reference* to another tuple; for example, the foreign key values 401 and 402 in the O_CUSTNO attribute of the **Order** relation refer to the tuples for Kinbote and Quilty in the **Customer** relation. Through foreign keys, the disparate relations of a relational database become connected so as to form a coherent whole.

Although a foreign key in one relation usually refers to a tuple of some *other* relation,

it is also possible for a foreign key to refer to a tuple of the *same* relation. The E_MGR attribute of the **Employee** relation, discussed above, is an instance of this special kind of foreign key: the manager of one employee is just another employee (or possibly, as we have seen, the *same* employee).

2.1.2 Further Details of the Sample Database

There are still three relations in the sample database that we have not mentioned: the **LineItem**, **Part**, and **Source** relations. These relations give additional information about customer orders—specifically, about the *parts* that have been ordered.

The **Order** relation discussed above contains only *some* of the information about customer orders. The reason why it gives only partial information is that order information is somewhat complicated, and cannot easily be represented in tabular form. In a single order, a customer may request various parts, each of which appears as a separate line item on the customer's invoice. The number of line items varies from order to order, but since the format of the tuples of the **Order** relation is fixed, this format cannot accommodate varying numbers of line items. Consequently, our sample database omits line-item information from the **Order** relation, and instead places this information in a separate relation—the **LineItem** relation. Our database thus *decomposes* the order information into the **Order** and **LineItem** relations.

Each tuple of the **LineItem** relation represents one line item of one order. Accordingly, a **LineItem** tuple specifies the order to which it belongs (through the foreign key L_ORDERNO), as well as the part ordered (L_PARTNO) and the quantity ordered (L_QTY). The part is identified only by a part number, which serves as a foreign key referencing the **Part** relation. The **Part** relation, in turn, has a P_PARTNO attribute as its primary key, and a P_DESCR attribute that gives brief descriptions of the parts.

The **Source** relation gives information about wholesalers (or other suppliers) from whom parts can be obtained. The S_PARTNO attribute, a foreign key referencing the **Part** relation, identifies a part, while the S_SUPPL attribute names a supplier who supplies that part.

Thus, starting from the **LineItem** relation, we see that order number 1001 specifies a

request for 2 head lamps (i.e., part number 7007); according to the **Source** relation, this part is supplied by Acme. Order number 1002 specifies a request for 1 head lamp, and in a separate line item, a request for 8 spark plugs (i.e., part number 8008). As the **Source** relation shows, the spark plugs can be obtained from any of four suppliers: Acme, Jolt, Nuke, or Zap.

2.2 The Relational Algebra

The *relational algebra* [11, 32, 38] is an algebraic language for manipulating relations to obtain new relations.

There are six fundamental operators in the relational algebra: *select* (σ), *project* (π), *Cartesian product* ($\times$), *set union* ($\cup$), *set difference* ($\setminus$ or $-$), and *rename* (ρ or δ). In addition, expressions in the relational algebra frequently make use of several *derived* operators that can be defined in terms of the fundamental operators. By far the most important of the derived operators is *join* ($\bowtie$).

In the present work, we shall be concerned with just a few of the relational operators: *select*, *Cartesian product*, and *join*. We review the definitions and essential properties of these operators below.

In discussing these operators, and throughout this work, we shall use the capital letters A, B, C, and so on, to represent arbitrary relations. These letters may also represent relation-valued expressions, provided the internal structure of these expressions is of no concern; but we shall use the letter E, sometimes with a subscript or prime (e.g., E_0'), when we wish to focus attention on an expression's structure. When we need to refer to a large or indeterminate number of relations, we shall call them R_0, R_1, R_2, and so on; we will also use the letter R with other subscripts (or no subscript) to represent relations in particular roles. The small letters p and q, occasionally with primes (p' and q'), shall represent predicates. The *cardinality* of a relation A (i.e., the *number of tuples* in A) will be denoted $|A|$. Later on we shall introduce additional notation as needed.

2.2.1 The *Cartesian Product* Operator

In ordinary mathematical usage, the Cartesian product of two sets is the set of all ordered pairs of elements drawn from the two sets. For example, the Cartesian product of $\{1, 2, 3\}$ and $\{a, b\}$ is $\{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$. Since a relation is just a set of tuples, the set-theoretic notion of Cartesian product makes sense for relations as well. However, the relational definition of Cartesian product differs subtly from the set-theoretic one. In the relational Cartesian product, pairs of tuples drawn from the two operands are *concatenated* to form new tuples, rather than simply being combined into ordered pairs.

Figure 2.2(a) shows the Cartesian product of the **LineItem** and **Part** relations from our sample database. Figure 2.2(b) shows the Cartesian product of same two relations with their roles reversed—that is, with **LineItem** as the *right-hand* operand of $\times$, and **Part** as the *left-hand* operand. Figure 2.2(c) shows another Cartesian product that has **Part** as the left-hand operand—this time the right-hand operand is the **Source** relation. In each instance, the result is a set of *tuples*, not a set of *pairs of tuples*; in other words, the Cartesian product yields a result that is itself a relation.

A property of relations that we have left implicit until now is that the attribute names in a given relation are all distinct, so that each attribute name unambiguously identifies a unique column of the relation. To preserve this property of relations under the Cartesian product operator, we must forbid the construction of a product $A \times B$ if there is any overlap between the attribute names of A and those of B; and in particular, we must forbid products of the form $A \times A$. However, it should be understood that these restrictions do not alter the expressive power of the relational algebra: where necessary, conflicts among attribute names can be resolved through the use of *rename* operators. (We do not discuss *rename* operators in detail, as we shall have no need for them here.)

The relations shown in Figures 2.2(a) and (b) are actually considered to be *the same relation.* In the tabular representation of a relation, the order of the rows and columns has no significance from the standpoint of relational theory. It is merely convenient to show the columns of a Cartesian product in the same order in which they appear in the operands, and to show the rows in an order based on the order of the operand rows,

LineItem × Part

L_ORDERNO	L_PARTNO	L_QTY	P_PARTNO	P_DESCR
1001	7007	2	7007	Head Lamp
1001	7007	2	8008	Spark Plug
1002	7007	1	7007	Head Lamp
1002	7007	1	8008	Spark Plug
1002	8008	8	7007	Head Lamp
1002	8008	8	8008	Spark Plug

(a) A relational Cartesian product

Part × LineItem

P_PARTNO	P_DESCR	L_ORDERNO	L_PARTNO	L_QTY
7007	Head Lamp	1001	7007	2
7007	Head Lamp	1002	7007	1
7007	Head Lamp	1002	8008	8
8008	Spark Plug	1001	7007	2
8008	Spark Plug	1002	7007	1
8008	Spark Plug	1002	8008	8

(b) An alternative representation of the same product

Part × Source

P_PARTNO	P_DESCR	S_PARTNO	S_SUPPL
7007	Head Lamp	7007	Acme
7007	Head Lamp	8008	Acme
7007	Head Lamp	8008	Jolt
7007	Head Lamp	8008	Nuke
7007	Head Lamp	8008	Zap
8008	Spark Plug	7007	Acme
8008	Spark Plug	8008	Acme
8008	Spark Plug	8008	Jolt
8008	Spark Plug	8008	Nuke
8008	Spark Plug	8008	Zap

(c) Another relational Cartesian product

Figure 2.2: Examples of the relational Cartesian product

with the leftmost attributes varying most slowly. But discounting order, the tables in Figures 2.2(a) and (b) contain identical information.

Algebraic Properties of Cartesian Product The algebraic significance of the equivalence of Figures 2.2(a) and (b) is that the relational Cartesian product is *commutative*: For arbitrary relations A and B, we have

$$A \times B = B \times A. \tag{2.1}$$

The same is not true for the set-theoretic Cartesian product, which is commutative only *up to isomorphism.* For example, the set-theoretic product $\{1\} \times \{a\}$ is $\{(1, a)\}$, whereas the product $\{a\} \times \{1\}$ is $\{(a, 1)\}$. Although $(1, a)$ contains the same information as $(a, 1)$, the two cannot be considered identical, because by definition they are *ordered* pairs—hence order cannot be discounted.

In a similar vein, the relational Cartesian product is *associative*, while the set-theoretic Cartesian product is associative only up to isomorphism. For example, in the set-theoretic setting, the product $(\{1\} \times \{2\}) \times \{3\}$ gives $\{((1, 2), 3)\}$, whereas $\{1\} \times (\{2\} \times \{3\})$ gives $\{(1, (2, 3))\}$; evidently these two results differ structurally. But such structural differences do not arise in relational Cartesian products, again because the operand tuples are combined by concatenation, and not by pairing. For example, Figure 2.3 shows the three-way Cartesian product of the **LineItem**, **Part**, and **Source** relations. This product can be regarded equally well as the product of the relation in Figure 2.2(a) and the **Source** relation, or as the product of the **LineItem** relation and the relation in Figure 2.2(c). In general, for arbitrary A, B, and C, we have the law

$$(A \times B) \times C = A \times (B \times C). \tag{2.2}$$

It is important to emphasize that despite their structural differences, the relational and set-theoretic Cartesian products are *isomorphic* to one another. One aspect of this isomorphism is that the cardinalities of both kinds of Cartesian products can be computed in the same way. For arbitrary relations A and B,

$$|A \times B| = |A| \cdot |B|. \tag{2.3}$$

LineItem × Part × Source

L_ORDERNO	L_PARTNO	L_QTY	P_PARTNO	P_DESCR	S_PARTNO	S_SUPPL
1001	7007	2	7007	Head Lamp	7007	Acme
1001	7007	2	7007	Head Lamp	8008	Acme
1001	7007	2	7007	Head Lamp	8008	Jolt
1001	7007	2	7007	Head Lamp	8008	Nuke
1001	7007	2	7007	Head Lamp	8008	Zap
1001	7007	2	8008	Spark Plug	7007	Acme
1001	7007	2	8008	Spark Plug	8008	Acme
1001	7007	2	8008	Spark Plug	8008	Jolt
1001	7007	2	8008	Spark Plug	8008	Nuke
1001	7007	2	8008	Spark Plug	8008	Zap
1002	7007	1	7007	Head Lamp	7007	Acme
1002	7007	1	7007	Head Lamp	8008	Acme
1002	7007	1	7007	Head Lamp	8008	Jolt
1002	7007	1	7007	Head Lamp	8008	Nuke
1002	7007	1	7007	Head Lamp	8008	Zap
1002	7007	1	8008	Spark Plug	7007	Acme
1002	7007	1	8008	Spark Plug	8008	Acme
1002	7007	1	8008	Spark Plug	8008	Jolt
1002	7007	1	8008	Spark Plug	8008	Nuke
1002	7007	1	8008	Spark Plug	8008	Zap
1002	8008	8	7007	Head Lamp	7007	Acme
1002	8008	8	7007	Head Lamp	8008	Acme
1002	8008	8	7007	Head Lamp	8008	Jolt
1002	8008	8	7007	Head Lamp	8008	Nuke
1002	8008	8	7007	Head Lamp	8008	Zap
1002	8008	8	8008	Spark Plug	7007	Acme
1002	8008	8	8008	Spark Plug	8008	Acme
1002	8008	8	8008	Spark Plug	8008	Jolt
1002	8008	8	8008	Spark Plug	8008	Nuke
1002	8008	8	8008	Spark Plug	8008	Zap

Figure 2.3: A three-way relational Cartesian product

For example, from Figure 2.1 we see that the cardinality of the **LineItem** relation is 3, and that that of the **Part** relation is 2. Hence the cardinality of their Cartesian product is seen in Figure 2.2(a) to be $3 \cdot 2 = 6$. The cardinality of the three-way Cartesian product in Figure 2.3 is just the product of the cardinalities of the **LineItem**, **Part**, and **Source** relations: $3 \cdot 2 \cdot 5 = 30$.

2.2.2 The *Select* Operator

The *select* operator has just one operand relation, and yields a subset of the tuples in that operand as its result. The result omits all operand tuples that do not satisfy a specified predicate. The usual notation for this operation is $\sigma_p(A)$, where A is a relation, and p is the selection predicate.

Figure 2.4 illustrates the application of selection to the relations that appear in Figures 2.2 and 2.3. The examples in Figure 2.4 specify selection predicates that weed out tuples in which there is a disagreement among the different "part number" attributes. Thus, in Figure 2.4(a), the selection predicate is L_PARTNO = P_PARTNO, so that the result of the selection will retain only those tuples in which the **LineItem** and **Part** attributes are related. For example, in the second tuple of Figure 2.4(a), the appearance of 1002 under L_ORDERNO, and Head Lamp under P_DESCR, reflects the fact that order number 1002 includes a line item for a head lamp. By contrast, in the second tuple of the Cartesian product in Figure 2.2(a), the appearance of 1001 together with Spark Plug has no particular meaning—it is just one of all possible juxtapositions. Extracting the meaningful tuples from a Cartesian product is one of the most important uses of selection.

In Figure 2.4(d), the selection predicate is the *conjunction* L_PARTNO = P_PARTNO $\wedge$ P_PARTNO = S_PARTNO, which weeds out any tuple in which the attributes L_PARTNO, P_PARTNO, and S_PARTNO are not all three the same.

In the examples above, the predicates (or predicate conjuncts) all have the form *attribute* = *attribute*. But a selection predicate may also compare an attribute to a *constant*, and the comparison operator need not test for *equality*. For example, L_QTY < 5 would be an acceptable selection predicate.

$\sigma_{\text{L\_PARTNO=P\_PARTNO}}(\textbf{LineItem} \times \textbf{Part})$

L_ORDERNO	L_PARTNO	L_QTY	P_PARTNO	P_DESCR
1001	7007	2	7007	Head Lamp
1002	7007	1	7007	Head Lamp
1002	8008	8	8008	Spark Plug

(a) A selection applied to the result in Figure 2.2(a)

$\sigma_{\text{L\_PARTNO=P\_PARTNO}}(\textbf{Part} \times \textbf{LineItem})$

P_PARTNO	P_DESCR	L_ORDERNO	L_PARTNO	L_QTY
7007	Head Lamp	1001	7007	2
7007	Head Lamp	1002	7007	1
8008	Spark Plug	1002	8008	8

(b) A selection applied to the result in Figure 2.2(b)

$\sigma_{\text{P\_PARTNO=S\_PARTNO}}(\textbf{Part} \times \textbf{Source})$

P_PARTNO	P_DESCR	S_PARTNO	S_SUPPL
7007	Head Lamp	7007	Acme
8008	Spark Plug	8008	Acme
8008	Spark Plug	8008	Jolt
8008	Spark Plug	8008	Nuke
8008	Spark Plug	8008	Zap

(c) A selection applied to the result in Figure 2.2(c)

$\sigma_{\text{L\_PARTNO=P\_PARTNO} \wedge \text{P\_PARTNO=S\_PARTNO}}(\textbf{LineItem} \times \textbf{Part} \times \textbf{Source})$

L_ORDERNO	L_PARTNO	L_QTY	P_PARTNO	P_DESCR	S_PARTNO	S_SUPPL
1001	7007	2	7007	Head Lamp	7007	Acme
1002	7007	1	7007	Head Lamp	7007	Acme
1002	8008	8	8008	Spark Plug	8008	Acme
1002	8008	8	8008	Spark Plug	8008	Jolt
1002	8008	8	8008	Spark Plug	8008	Nuke
1002	8008	8	8008	Spark Plug	8008	Zap

(d) A selection applied to the result in Figure 2.3

Figure 2.4: Examples of the selection operation

Whenever a relational expression involves selection, one must take care that the expression be *well-formed.* A selection $\sigma_p(A)$ is well-formed provided that all attributes mentioned by p are in fact attributes of A, and provided that the comparisons involved are type-correct. For example, $\sigma_{\text{L\_QTY}<5}(\textbf{LineItem})$ is well-formed, but $\sigma_{\text{L\_QTY}<5}(\textbf{Part})$ is not, since L_QTY is not an attribute of the **Part** relation. Likewise, the expression $\sigma_{\text{L\_QTY}=\text{'Zap'}}(\textbf{LineItem})$ is ill-formed, since L_QTY represents numeric data, while 'Zap' is a text string.

Algebraic Properties of Select A selection operation whose predicate is a conjunction can also be expressed as a succession of selections. That is, for all A, p, and q,

$$\sigma_{p \wedge q}(A) = \sigma_p(\sigma_q(A)), \tag{2.4}$$

and similarly for predicates of more than two conjuncts. Since Boolean conjunction is commutative, it follows directly that

$$\sigma_p(\sigma_q(A)) = \sigma_q(\sigma_p(A)). \tag{2.5}$$

From these laws we see that each conjunct of a compound predicate can be viewed as a selection predicate in its own right. It is a convention in the query-optimization literature to use the term *predicate* to refer both to the whole of a predicate and to its individual conjuncts. When there is no danger of confusion, we shall use the term both ways here as well.

The significance of laws (2.4) and (2.5) lies largely in their interaction with the following algebraic laws that relate σ and $\times$. Provided that both the left- and right-hand sides are well-formed, we have, for arbitrary A, B, and p,

$$\sigma_p(A \times B) = \sigma_p(A) \times B \tag{2.6}$$

$$\sigma_p(A \times B) = A \times \sigma_p(B). \tag{2.7}$$

Note that given a particular choice of A, B, and p, the well-formedness condition will ordinarily be met by *at most one* of (2.6) and (2.7). Only in the degenerate case where p mentions no attributes from A or B will both (2.6) and (2.7) hold at the same time.

Using (2.4) and (2.5) together with (2.6) and (2.7), one can carry out transformations such as the following, in which we assume that p' depends only on attributes of C:

$$\sigma_{p \wedge p'}(A \times \sigma_q(B \times C)) = \sigma_p(\sigma_{p'}(A \times \sigma_q(B \times C))) \tag{2.8}$$

$$= \sigma_p(A \times \sigma_{p'}(\sigma_q(B \times C))) \tag{2.9}$$

$$= \sigma_p(A \times \sigma_q(\sigma_{p'}(B \times C))) \tag{2.10}$$

$$= \sigma_p(A \times \sigma_q(B \times \sigma_{p'}(C))). \tag{2.11}$$

In this sequence of rewrites, the conjunct p' has slid past two Cartesian product operators—and also past the operator σ_q—to attach itself to a term deep inside the original expression. Such transformations are often referred to as *pushing down* a predicate, or as *pushing a select* past other operators.

As noted above, (2.6) and (2.7) are applicable only insofar as both sides of these equations are well-formed. To illustrate a violation of the well-formedness condition, consider the use of (2.6) to rewrite the expression

$$\sigma_{\text{S\_SUPPL='Zap'}}(\textbf{Part} \times \textbf{Source}) \tag{2.12}$$

to

$$\sigma_{\text{S\_SUPPL='Zap'}}(\textbf{Part}) \times \textbf{Source}. \tag{2.13}$$

The original expression makes sense, but the rewritten expression is nonsense, because S_SUPPL is not an attribute of **Part**. On the other hand, the application of (2.7) to the same expression would yield the legitimate rewritten form

$$\textbf{Part} \times \sigma_{\text{S\_SUPPL='Zap'}}(\textbf{Source}). \tag{2.14}$$

2.2.3 The *Join* Operator

As we saw above, Cartesian products often contain tuples in which unrelated information is tacked together arbitrarily; these products become useful only after the meaningless tuples are discarded. For this reason, Cartesian products are frequently subject to subsequent

selection—so frequently that the idiom $\sigma(\ldots\times\ldots)$ has its own special notation. The *join* operator ($\bowtie$) combines Cartesian product with selection:

$$A \bowtie_p B \stackrel{\text{def}}{=} \sigma_p(A \times B). \tag{2.15}$$

In the context of join notation, the predicate p is referred to as a *join predicate.* If p is a conjunction, then each of its conjuncts may be considered a separate join predicate.

Seen in the light of definition (2.15), the expressions in Figures 2.4(a), (b), and (c) can all be viewed as examples of join operations. Indeed the join operations in these examples are of a particular kind, and are referred to as *equijoins*—joins whose predicates specify that certain attributes from the left-hand operand are *equal* to certain attributes from the right-hand operand. (More generally, an equijoin may equate values *derived* from the respective operand attributes.)

Shortly we will see that in Figure 2.4(d), the selection over a three-way Cartesian product can be rewritten as a three-way join. We will also remark on the fact that the join operator—and the equijoin in particular—is important to query processing for reasons over and above notational convenience. But first let us consider some algebraic properties of the two-way join.

Algebraic Properties of Two-way Join Given that the join operator is defined in terms of the Cartesian product operator, it should come as no surprise that these two operators have similar algebraic properties. Commutativity of join follows immediately from (2.15) and from the commutativity of the Cartesian product. For example, in Figures 2.4(a) and (b), the Cartesian products inside the selection operations are equivalent by commutativity; and since the selection predicate is also the same in both instances, Figures 2.4(a) and (b) necessarily yield the same result relations (except for attribute order). Using join notation, the equivalence of these two result relations may be stated as

$$\textbf{LineItem} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} \textbf{Part} = \textbf{Part} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} \textbf{LineItem}. \tag{2.16}$$

More generally, for arbitrary A, B, and p, we have

$$A \bowtie_p B = B \bowtie_p A. \tag{2.17}$$

The join operator likewise inherits the Cartesian product's amenability to predicate pushing. For arbitrary A, B, p, and q, we have

$$\sigma_p(A \bowtie_q B) = \sigma_p(A) \bowtie_q B \tag{2.18}$$

$$\sigma_p(A \bowtie_q B) = A \bowtie_q \sigma_p(B), \tag{2.19}$$

subject to the same provisos regarding well-formedness that applied in the case of the Cartesian product.

The Cartesian product may in fact be regarded as a special case of join. When the join predicate is vacuously true, then the selection in the definition of the join operator (2.15) has no effect, and the join degenerates to a Cartesian product. This special case is important enough to state as a law:

Law *For all A and B,*

$$A \bowtie_p B = A \times B \tag{2.20}$$

when $p \equiv$ True.

The *empty* predicate may be thought of as a conjunction with zero conjuncts, and hence as vacuously true. By a literal reading, therefore, the expression $A \bowtie B$ must be taken as synonymous with $A \times B$ [38].

However, more often than not we shall use the notation $A \bowtie B$ informally to mean "the join of A and B under whatever predicate or predicates *belong to* (or *are applicable to*) the join." Our uses of the unannotated symbol $\bowtie$ in Chapter 1 were of this informal variety—the predicates were implicit. The premise was that a collection of predicates had been supplied in the **WHERE** clause of an SQL query, and that these were the predicates we had to work with. Loosely speaking, the predicates that "belong to" a join of the form $A \bowtie B$ would be those that mention attributes from both A and B—and do not mention attributes *outside* of A and B.

Note that in some expressions, it may happen that *no predicates* belong to a given join operator. Thus, the unannotated expression $A \bowtie B$, with implicit predicates, may denote either a bona fide join, *or* a Cartesian product.

Three-way Join and Join Associativity Now let us consider the matter of three-way joins. In the expression in Figure 2.4(d), we have a selection over a three-way Cartesian product. By splitting apart the conjuncts of the selection predicate, and by pushing down one of the conjuncts, we can derive the equivalent expression

$$\sigma_{\text{P\_PARTNO=S\_PARTNO}}(\sigma_{\text{L\_PARTNO=P\_PARTNO}}(\textbf{LineItem} \times \textbf{Part}) \times \textbf{Source}). \tag{2.21}$$

We can then use the join operator to abbreviate this expression as

$$(\textbf{LineItem} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} \textbf{Part}) \bowtie_{\text{P\_PARTNO=S\_PARTNO}} \textbf{Source}. \tag{2.22}$$

Alternatively, we could have pushed down the *other* conjunct of the selection predicate to obtain

$$\sigma_{\text{L\_PARTNO=P\_PARTNO}}(\textbf{LineItem} \times \sigma_{\text{P\_PARTNO=S\_PARTNO}}(\textbf{Part} \times \textbf{Source})); \tag{2.23}$$

the abbreviated form of the latter expression would then have been

$$\textbf{LineItem} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} (\textbf{Part} \bowtie_{\text{P\_PARTNO=S\_PARTNO}} \textbf{Source}). \tag{2.24}$$

The equivalence of (2.22) and (2.24) seems to suggest associativity of the join operator.

But the question of join associativity is tricky. Suppose we commute the inner join in (2.22) to obtain

$$(\textbf{Part} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} \textbf{LineItem}) \bowtie_{\text{P\_PARTNO=S\_PARTNO}} \textbf{Source}, \tag{2.25}$$

and then attempt to change the association of this last expression. The resulting expression,

$$\textbf{Part} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} (\textbf{LineItem} \bowtie_{\text{P\_PARTNO=S\_PARTNO}} \textbf{Source}), \tag{2.26}$$

is not well-formed. This example shows that join associativity cannot be taken for granted.

There are several ways to present join associativity in a manner that guards against ill-formed expressions. The most straightforward approach is simply to state an associativity law with a well-formedness condition, as follows:

Law *For all A, B, C, p, and q,*

$$(A \bowtie_p B) \bowtie_q C = A \bowtie_p (B \bowtie_q C), \tag{2.27}$$

provided that both sides are well-formed.

But this formulation is unnecessarily restrictive, in that it requires that the join predicates be the same on both sides of the equation.

Relaxing this restriction, one obtains an alternative, more widely applicable, law:

Law *For all A, B, C, p, q, p', and q',*

$$(A \bowtie_p B) \bowtie_q C = A \bowtie_{p'} (B \bowtie_{q'} C), \tag{2.28}$$

provided that both sides are well-formed, and provided that $p \wedge q = p' \wedge q'$.

Note that if A, B, C, p, and q are given, then it is always possible to find p' and q' that satisfy the requirements of the law; and conversely, if A, B, C, p', and q' are given, then it is always possible to find suitable p and q. However, when p and q are among the givens, it is not always possible to find suitable, *nonvacuous* p' and q'; and the analogous caveat applies when p' and q' are given, and p and q are to be found. Consequently, changing the association of a three-way join may effectively result in the introduction of a Cartesian product.

To conclude, if Cartesian products are to be avoided, then join is not always associative. On the other hand, if the Cartesian product operation is considered to be merely a special case of join, then join always *is* associative. Using our informal notation for join with implicit predicates, we may state the general join-associativity law as follows:

Law *For all A, B, and C,*

$$(A \bowtie B) \bowtie C = A \bowtie (B \bowtie C), \tag{2.29}$$

where the join operators are implicitly qualified by all applicable predicates.

The validity of (2.29) in general is a consequence of the following fact: Any predicate applicable to $(A \bowtie B) \bowtie C$ is also applicable to $A \bowtie (B \bowtie C)$, and vice versa.

The Role of Join in Query Evaluation The join operator is not only a notational convenience, but also plays an important role in efficient query evaluation.

As we have seen, the expression $A \bowtie_p B$ is algebraic shorthand for $\sigma_p(A \times B)$. It is therefore certainly possible to compute the result of such a join by first computing the Cartesian product $A \times B$, and then applying the selection operation σ_p. But such an approach to computing a join is likely to be extremely inefficient, for the following reason.

Suppose that A and B each have cardinality 1,000. Then there are 1,000,000 tuples in the Cartesian product $A \times B$, and the cost of computing this product is proportional to its size. Now the selection operation may cause most of those 1,000,000 tuples to be discarded, so the final result of $\sigma_p(A \times B)$ may just have, say, 2,000 tuples. In such a case, the computation of the intermediate result represented by the Cartesian product mostly goes to waste.

Join-processing algorithms provide an efficient alternative. Where applicable, such algorithms can compute $A \bowtie_p B$ directly, bypassing the Cartesian product computation. (Join-processing algorithms are almost always applicable when $A \bowtie_p B$ is an equijoin; in other cases, they may not be.)

There are numerous join-processing algorithms [43]; among the most widely used are *hash join*, *merge join*, and *index-nested-loops join*. Each of these algorithms has distinctive performance characteristics, but generally the cost of computing $A \bowtie_p B$ with such an algorithm is roughly proportional to the cardinality of one the relations A, B, or $A \bowtie_p B$. Usually (though not always) the cost is proportional to the cardinality of the *largest* of these relations.

In any event, if A and B each have cardinality 1,000, and if $A \bowtie_p B$ has cardinality 2,000, then the number of tuples that must be examined or generated in the course of join processing will typically lie in the thousands, in contrast to the million-odd tuples that must be processed when the join is computed by way of a Cartesian product. Examples of this kind demonstrate the tremendous *practical* importance of the join operator.

2.2.4 Summary

In this section we have taken a brief tour of three relational operators—Cartesian product, select, and join—that are central to the development of the ideas in this work. We have seen that relational Cartesian product and join are commutative and associative (subject to restrictions), and that selection operators can be "pushed past" both of the other two operators. These algebraic properties provide the foundation for the manipulations involved in join-order optimization.

In addition, we noted that the cardinality of a Cartesian product is equal to the product of the cardinalities of its operands. The latter property will prove important in judging the relative merits of different join orders.

2.3 Query Processing

Let us now briefly consider what happens, from start to finish, when an SQL query is submitted to a relational database management system for processing.

2.3.1 A Sample Query

Suppose one wished to issue the following information retrieval request against our sample database of Figure 2.1 above:

> *List the order number, part number, quantity ordered, and part description for all orders for parts that can be supplied by Zap.*

One might formulate this question as an SQL query in the manner illustrated in Figure 2.5(a). The symbol * in the query's **SELECT** clause indicates that the query result should include all attributes from all the relations mentioned in the **FROM** clause. Note the predicates in the **WHERE** clause that equate a part-number foreign key with the corresponding primary key in the **Part** relation. These predicates ensure that each tuple of the query result will give coherent information about a particular part. In addition, the predicate S_SUPPL = 'Zap' ensures that only those parts that are supplied by Zap will appear in the query result. As it happens, the query result in this example consists of a single tuple, as illustrated in Figure 2.5(b).

```
SELECT  *
FROM    LineItem, Part, Source
WHERE   L_PARTNO = P_PARTNO AND
        P_PARTNO = S_PARTNO AND
        S_SUPPL = 'Zap'
```

(a) An SQL query

L_ORDERNO	L_PARTNO	L_QTY	P_PARTNO	P_DESCR	S_PARTNO	S_SUPPL
1002	8008	8	8008	Spark Plug	8008	Zap

(b) The query result

Figure 2.5: An SQL query and its result

Examining the data in Figure 2.1 in terms of the illustrated query, one can see that the query result shown makes intuitive sense. We next consider the process by which a database system can obtain such a result mechanically.

2.3.2 Phases of Query Processing

To process a query and obtain a result relation, the query-processing subsystem of a relational database management system goes through several phases, roughly as follows:

1. *The translation phase.* The query processing system's *parser* translates the SQL query into an *internal representation.* Let us assume that the internal representation is an expression in the relational algebra.

2. *The optimization phase.* The system's *query optimizer* may rewrite the relational algebra expression obtained in the translation phase to a *different* but *semantically* equivalent expression. Such rewrites are based on laws of the relational algebra, as discussed in Section 2.2 above. The objective of rewriting is to obtain an expression that has lower *cost* (i.e., that may be evaluated more efficiently) than the original.

In addition, the optimizer replaces the so-called *logical* operators of the rewritten expression—operators such as join, select, and Cartesian product—with suitable *physical* operators. For example, a given logical *join* operator will be replaced by a *merge join*, *hash join*, or *index-nested-loops join* operator, or by some other physical join operator that specifies a particular join algorithm. Analogous physical alternatives will also replace any other relational operators.

The expression that results when all logical operators have been replaced by physical operators is called a *query plan*, or simply a *plan*. The term *query plan* has an abundance of synonyms in the literature; a query plan may also be referred to as a *QEP* (for *query execution plan* or *query evaluation plan*), as an *access plan*, an *operator tree*, a *processing tree*, or a *join-processing tree*. (There are many additional variants of these terms.) The latter terms involving the word *tree* reflect the fact that query plans are often depicted graphically as in Figure 1.1 on page 5. We shall have occasion to refer to plans as *processing trees* in the later part of this work, when we shall deal with explicit representations of query plans as tree structures.

3. *The execution* or *evaluation phase*. The system's *execution engine* evaluates the query plan. That is, it executes the algorithms specified by the query plan's physical operators, using the specified relations as inputs.

This characterization of query processing is inexact in various ways. For example, commercial query processing systems cannot use the relational algebra as an internal representation for queries, because the relational algebra has less expressive power than the full SQL language. But for our purposes, the relational algebra is expressive enough. We shall comment further on the form of the optimizer input later on.

Likewise, query plans in actual systems are not simply expressions involving physical operators. Rather, they are complex data structures that give the execution engine detailed instructions on the handling of the query's operators. However, we shall not burden ourselves with such details here.

In fact, we shall further simplify by generally not distinguishing between logical and physical join operators. (Much of the join-optimization literature does likewise.) We

may imagine that our execution engine supports only a single join algorithm; in that case, all logical join operators map to the same physical join operator, and so there is no need to specify the physical operator. Alternatively, allowing for the possibility of multiple join algorithms, we may think of a logical join operator as representing the *lowest-cost* alternative of the physical join operators. The latter interpretation requires a more complicated *cost model*—a topic to which we shall return in Section 2.6 below—but permits us to deal exclusively with logical join operators, without loss of generality.

2.3.3 Processing of the Sample Query

To make concrete the brief explanation of query processing just given, let us now consider how it applies to the sample query of Figure 2.5(a).

Translation In the translation phase, this query must be translated into an expression in the relational algebra. The first step in this translation is to construct the Cartesian product of the relations in the sample query's **FROM** clause:

$$\textbf{LineItem} \times \textbf{Part} \times \textbf{Source} \tag{2.30}$$

Because the Cartesian product is associative, this expression is unambiguous without parentheses. But later on, it may become confusing to work with expressions whose association is left unspecified. Let us therefore arbitrarily associate the product (2.30) to the left:

$$(\textbf{LineItem} \times \textbf{Part}) \times \textbf{Source} \tag{2.31}$$

This Cartesian product will become the argument of a selection operation that applies the complex predicate from the sample query's **WHERE** clause:

$$\sigma_{\text{L\_PARTNO}=\text{P\_PARTNO} \wedge \text{P\_PARTNO}=\text{S\_PARTNO} \wedge \text{S\_SUPPL}=\text{'Zap'}}((\textbf{LineItem} \times \textbf{Part}) \times \textbf{Source}) \tag{2.32}$$

In (2.32) we now have a relational algebra expression that is equivalent to the sample query. The translation was particularly simple in this case because the sample query's **SELECT** clause, through the symbol *, requested *all* attributes of the participating

tables. In the more general situation where only *some* of the attributes are requested, the translation would have had to attach an additional operation (a *projection*). However, the optimization problem remains essentially the same whether all attributes are requested, or just a subset of them. For simplicity, we shall consider only the former case here.

Optimization (Part I) Let us suppose now that (2.32) becomes the input to a query optimizer. The optimizer must find a query plan equivalent to (2.32) that can be evaluated at low cost—preferably at a cost that is minimal for all alternative plans.

For the purposes of illustration, we will break the optimization process into two subphases. The objective of the first subphase is to perform "pre-optimization" transformations that are not based on cost analysis, but are deemed advantageous on the basis of *a priori* considerations. In particular, it may make sense to push all the conjuncts of the selection predicate as far down as they will go—as well as to convert the Cartesian product operators into joins. Thus, we obtain

$$(\textbf{LineItem} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} \textbf{Part}) \bowtie_{\text{P\_PARTNO=S\_PARTNO}} (\sigma_{\text{S\_SUPPL='Zap'}}(\textbf{Source})). \tag{2.33}$$

The transformation of (2.32) into (2.33) is achieved through repeated application of the predicate-decomposition rule (2.4), the predicate push-down rules (2.6) and (2.7), and the definition of join (2.15). This transformation has *heuristic* merit because

1. It is generally beneficial to apply all predicates as "early" as possible, so as to reduce the cardinalities of operands and intermediate results;

2. A Cartesian product that serves as the argument of a selection on fully pushed-down predicates should always be converted to a join, for the reasons discussed in Section 2.2.3 above.

There are exceptions to the first point, as we shall remark later, but overall it remains an excellent heuristic. Note that the second point does *not* say that expressions with Cartesian products are necessarily inferior to expressions without them; it makes no judgment about a Cartesian product that is not immediately surrounded by a selection.

With (2.33) we have completed the first optimization subphase. But before we proceed, it is worth observing several facts about pushed-down predicates.

1. Predicates that mention attributes from only one relation, such as the predicate S_SUPPL = 'Zap' in the sample query, are special. Such predicates can always be pushed down past all the Cartesian product and join operators in a query, and so become tightly bound to the relation to which they apply. In our example, the predicate S_SUPPL = 'Zap' becomes attached to the **Source** relation in the subexpression $\sigma_{\text{S\_SUPPL='Zap'}}(\textbf{Source})$. Because this subexpression contains no join operators, it can be treated for the purposes of join-order optimization as an *atomic unit*—as if it were a stored relation in the database. We may ignore predicates that depend on only one relation, except inasmuch as they affect the characteristics (e.g., the cardinality) of that relation.

2. Predicates that mention attributes from *zero* relations—i.e., predicates that mention no attributes at all—are also special. We have no such predicates in our sample query, and for good reason: such predicates make no sense. A predicate that mentions no attributes must be equivalent either to the constant *True* or to the constant *False*. In the former case, it has no effect on the query result; in the latter case, it ensures that the query result will be empty. Neither behavior is likely to be useful, and on this basis we discuss zero-relation predicates no further.

3. Once the zero-relation and one-relation predicates have been excluded from consideration, we are left with the join predicates—predicates that become attached to join operators in the course of predicate push-down. Assuming that these predicates have been pushed down as far as possible, each conjunct will become attached to the join operator to which it "belongs," in the sense described in Section 2.2.3 above.

In light of the last observation, the following unannotated join expression may be regarded as equivalent to (2.33):

$$(\textbf{LineItem} \bowtie \textbf{Part}) \bowtie (\sigma_{\text{S\_SUPPL='Zap'}}(\textbf{Source})). \tag{2.34}$$

Later on, we will be able to profit from the fact that expressions such as (2.34) are unambiguous.

Optimization (Part II) Above we derived the expression

$$(\textbf{LineItem} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} \textbf{Part}) \bowtie_{\text{P\_PARTNO=S\_PARTNO}} (\sigma_{\text{S\_SUPPL='Zap'}}(\textbf{Source})) \tag{2.35}$$

as the result of the first optimization subphase. We now consider the second subphase, in which the optimizer performs a cost-based analysis of algebraically equivalent alternatives to (2.35)—specifically, alternatives that can be reached by way of join commutativity and associativity. In keeping with the predicate push-down heuristic discussed above, it is customary to perform join reassociation in such a way that predicates remain pushed down as far as possible in the reassociated expressions.

For simplicity, let us imagine that in the present example, the join commutativity and associativity laws lead to only one algebraic alternative to (2.35), namely

$$\textbf{LineItem} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} (\textbf{Part} \bowtie_{\text{P\_PARTNO=S\_PARTNO}} (\sigma_{\text{S\_SUPPL='Zap'}}(\textbf{Source}))). \tag{2.36}$$

Then what the optimizer must do at this stage is to *estimate* the evaluation cost of each of (2.35) and (2.36), and choose between them accordingly.

To give fair consideration to each expression, the optimizer needs to find the physical operator (i.e., the algorithm) best suited to each logical operator in the expression, and to base its cost estimates on the operator assignments so determined. However, it is possible to make *rough* cost estimates without considering physical operator assignments, by instead basing the estimates on properties of the logical expressions. In the case at hand, one can get a sense of the relative costliness of (2.35) and (2.36) by considering the cardinalities of the intermediate results that arise in each instance.

The intermediate result in the case of (2.35) comes from the subexpression

$$\textbf{LineItem} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} \textbf{Part}, \tag{2.37}$$

while the intermediate result in the case of (2.36) comes from the subexpression

$$\textbf{Part} \bowtie_{\text{P\_PARTNO=S\_PARTNO}} (\sigma_{\text{S\_SUPPL='Zap'}}(\textbf{Source})). \tag{2.38}$$

LineItem $\bowtie_{\text{L\_PARTNO=P\_PARTNO}}$ **Part**

L_ORDERNO	L_PARTNO	L_QTY	P_PARTNO	P_DESCR
1001	7007	2	7007	Head Lamp
1002	7007	1	7007	Head Lamp
1002	8008	8	8008	Spark Plug

(a) Intermediate result under left association

Part $\bowtie_{\text{P\_PARTNO=S\_PARTNO}}$ ($\sigma_{\text{S\_SUPPL='Zap'}}$(**Source**))

P_PARTNO	P_DESCR	S_PARTNO	S_SUPPL
8008	Spark Plug	8008	Zap

(b) Intermediate result under right association

Figure 2.6: Possible intermediate results in the evaluation of a three-way join

Figures 2.6(a) and (b) shows the respective results of these two-way joins. (The relation in Figure 2.6(a) is the same as that illustrated previously in Figure 2.4(a); and the relation in Figure 2.6(b) is a *subset* of the relation illustrated in Figure 2.4(c)—namely, the subset for which the value of the S_SUPPL attribute is 'Zap'.) Evidently (2.37) yields a larger intermediate result than (2.38), and since both (2.35) and (2.36) yield the same *final* result, we see that (2.35) entails the generation of a larger number of tuples overall. Thus, going by the number of tuples produced, one would have to conclude that (2.36) was the more economical expression, and hence should be considered the optimal join plan (or the basis for such a plan).

The cardinalities in our example are too small to be significant, but it is not hard to see how the effect illustrated can result in vastly different intermediate result cardinalities for different join orders. When the differences are large enough, the cardinalities alone suffice to settle the question of which of two join orders is less costly. In cases where the differences are subtler, more careful analysis is required: the optimizer must estimate the number of disk blocks that would be read and written using each available join algorithm, as well as the amount of CPU time the algorithm would expend in comparing and combining input tuples to produce output tuples.

But whether or not such detailed analysis turns out to be necessary in a given instance, the assessment of join order costs depends heavily on intermediate result cardinalities. To a large extent, the problem of accurate cost estimation is a problem of accurate cardinality estimation; and because we have thus far said nothing about how cardinality estimation is accomplished, we shall return to this question in Section 2.4 below.

Execution Once the optimizer has created a query plan based on expression (2.38), query execution is straightforward. Conceptually, at least, execution proceeds "bottom-up," starting with the innermost operators of the plan, and proceeding outwards.

As the first step in the execution of our sample query, a selection algorithm will cast out tuples of the **Source** relation in which the S_SUPPL attribute is not `'Zap'`. Next, the result of this selection will become the right-hand input to a join algorithm that will yield the relation shown in Figure 2.6(b). (In an actual database system, these selection and join steps might be combined into a single operation.) Finally, a second application of a join algorithm will combine the **Part** relation with the intermediate result of Figure 2.6(b), and so yield the final query result illustrated in Figure 2.5(b).

Query execution is often *pipelined* [17], meaning that successive operators run concurrently, and not in the strictly serial fashion suggested above. The primary motivation for pipelining is to avoid storing intermediate results. Without pipelining, an intermediate result is first produced as the output of one operator, and subsequently consumed as the input of another operator. In between the execution of those two operators, the intermediate result must be stored in its entirety—which means that if it is large, it must be stored on disk. Thus, the producing operator must write its output to disk, and the consuming operator must read its input from disk.

When pipelined, the producing and consuming operators execute in tandem. Each tuple emitted by the producing operator immediately becomes an input tuple of the consuming operator, and so neither operator has to access the disk. Consequently, pipelining may substantially reduce execution costs, and careful cost analyses must take this effect into account. On the other hand, pipelining does not change the *number* of tuples produced or consumed by an operator. Intermediate-result cardinalities give a good rough

gauge of execution costs whether or not execution is pipelined.

2.3.4 Discussion

In this section we have placed the problem of query optimization in context. The optimization phase of query processing is sandwiched between a translation phase, which prepares an SQL query for optimization, and an execution phase, which produces the query result, and in which the execution costs that the optimizer has sought to minimize are actually incurred.

Above we imagined the optimizer input to be an expression in the relational algebra. As we noted, the relational algebra lacks the generality required to serve as the input to commercial query optimizers; on the other hand, for the specialized problem of *join-order* optimization, the relational algebra offers *more* generality than is really needed. A join-order optimizer needs to know what relations are to be joined, and under what predicates; beyond that, the information furnished by a relational algebra expression is not useful, but merely distracting. For example, the initial join order that the input expression happens to have will likely change in the course of optimization—and the same goes for the assignment of join predicates to particular join operators. Accordingly, much of the join-optimization literature supposes a simpler representation of the input query that gives only the information required. We shall do likewise in the present work, as we explain in more detail below.

In any event, it is not enough for an optimizer to be provided with the query to be optimized. As we have seen, to do its job the optimizer must estimate costs, and to estimate costs it must estimate cardinalities. The optimizer therefore requires, as part of its input, the information on which these estimates are based. Our next concern is the manner in which this information is provided.

2.4 Cardinality Estimation and Predicate Selectivity

To estimate the cost of a query plan, a query optimizer requires cardinality estimates both for the query's *base* relations (i.e., the relations stored in the database), and for the

intermediate and final *result* relations that the plan computes in the course of its execution. Base-relation cardinalities tend to be readily available, as database management systems typically maintain a tuple count for each relation stored in a database. But estimating the cardinalities of the *results* of relational operations is another matter; it is a difficult topic with an extensive literature of its own. Fortunately, we can avoid delving into this topic in any depth.

In the study of join-order optimization, it is conventional to regard the problem of estimating result cardinalities as consisting of two parts:

1. The first part of the problem is to estimate *predicate selectivity* values (to be defined presently) for a query's predicates. This part of the problem is hard.

2. The second part of the problem is to use the predicate selectivity values to compute cardinalities. This part of the problem is straightforward.

By convention, join-order optimizers take the predicate selectivity values as *given*, so that the hard part of the problem is relegated to another program component. All we need to worry about here is the *use* of selectivity values in estimating cardinalities. But as we shall see, even this relatively straightforward part of the problem is not without its tricky aspects.

2.4.1 Concept and Properties of Selectivity

The notion of predicate *selectivity* may be defined informally as follows: The *selectivity* of a selection predicate p is the *probability* that a given tuple will satisfy p.

To state the matter another way, when the selection operator σ_p is applied to some relation A, the selectivity of p is the proportion of tuples of A that "survive" the selection:

$$selectivity(p) = \frac{|\sigma_p(A)|}{|A|}. \tag{2.39}$$

It follows that, given the selectivity of p and the cardinality of A, one can estimate the cardinality of $\sigma_p(A)$ as

$$|\sigma_p(A)| = selectivity(p) \cdot |A|. \tag{2.40}$$

For example, consider the selection $\sigma_{\text{S\_SUPPL}=\text{'Zap'}}(\textbf{Source})$. From Figure 2.1 we see that the **Source** relation has five tuples, and that only one of them satisfies the predicate S_SUPPL = 'Zap'. Hence, *selectivity*(S_SUPPL = 'Zap') = 1/5. Conversely, if we were given the selectivity value 1/5, together with the cardinality 5 of the **Source** relation, we could predict that the result of the selection in question would have cardinality $(\frac{1}{5}) \cdot 5 = 1$.

It goes without saying that in general, a predicate's selectivity depends on the contents of the database, and is subject to change as the database evolves. In other words, selectivity is not an intrinsic property of a predicate, but is a property of the predicate with respect to the database state. In our examples of predicate selectivity, we take the database state illustrated in Figure 2.1 as a given.

Selectivity of Join Predicates The example just given involved a predicate that mentioned only one relation. But the notion of selectivity applies equally to *join* predicates. Consider the relationship between the relation illustrated in Figure 2.2(a) and that illustrated in Figure 2.4(a). The former relation, **LineItem** $\times$ **Part**, has cardinality 6, while the latter relation, $\sigma_{\text{L\_PARTNO}=\text{P\_PARTNO}}(\textbf{LineItem} \times \textbf{Part})$, has cardinality 3. The predicate L_PARTNO = P_PARTNO therefore has selectivity 3/6, or 1/2.

More generally, when a selection operation is applied to a Cartesian product $A \times B$, the relationship expressed in (2.39) becomes

$$\textit{selectivity}(p) = \frac{|\sigma_p(A \times B)|}{|A \times B|} = \frac{|A \bowtie_p B|}{|A \times B|} = \frac{|A \bowtie_p B|}{|A| \cdot |B|}. \tag{2.41}$$

Hence, given the selectivity of p and the cardinalities of A and B, one can estimate the cardinality of $A \bowtie_p B$ as

$$|A \bowtie_p B| = \textit{selectivity}(p) \cdot |A| \cdot |B|\,. \tag{2.42}$$

Selectivity of Conjunctions Now observe what happens when we replace the predicate p in (2.39) with the conjunction $p \wedge q$, and then apply transformation (2.4):

$$\textit{selectivity}(p \wedge q) = \frac{|\sigma_{p \wedge q}(A)|}{|A|} = \frac{|\sigma_p(\sigma_q(A))|}{|A|}. \tag{2.43}$$

Multiplying both the numerator and denominator of the right-hand side by $|\sigma_q(A)|$, and then rearranging and simplifying, we obtain

$$selectivity(p \wedge q) = \frac{|\sigma_p(\sigma_q(A))|}{|A|} \cdot \frac{|\sigma_q(A)|}{|\sigma_q(A)|} \tag{2.44}$$

$$= \frac{|\sigma_p(\sigma_q(A))|}{|\sigma_q(A)|} \cdot \frac{|\sigma_q(A)|}{|A|} \tag{2.45}$$

$$= selectivity(p) \cdot selectivity(q). \tag{2.46}$$

Through the combined application of (2.40), (2.42), and (2.46), an optimizer can estimate all join-result cardinalities, given just the base-relation cardinalities and the selectivities of individual predicate conjuncts.

2.4.2 Difficulties with Selectivity

There are several difficulties with the idea of basing cardinality computations on predicate selectivities. To begin with, the selectivity estimates furnished to optimizers tend in practice to be rather haphazard; often they are pure invention. Consequently, even if an optimizer succeeds in finding the query plan with lowest estimated cost, there is no assurance that that plan is in fact the cheapest, or for that matter, that it is much good at all.

There are also technical and conceptual difficulties with the notion of selectivity. We discuss several of these difficulties below.

Non-independence of Predicates Consider the cardinalities of the relations illustrated in Figures 2.2(a) and (c) and in Figure 2.3; their cardinalities are 6, 10, and 30, respectively. Now consider the cardinalities of the selections from these relations illustrated in Figures 2.4(a), (c), and (d); the selection results have cardinalities of 3, 5, and 6, respectively. Dividing the latter cardinalities pairwise by the former, we obtain the

following selectivity values for the predicates that appear in Figure 2.4:

$$\textit{selectivity}(\text{L\_PARTNO} = \text{P\_PARTNO}) = 3/6 = 1/2 \tag{2.47}$$

$$\textit{selectivity}(\text{P\_PARTNO} = \text{S\_PARTNO}) = 5/10 = 1/2 \tag{2.48}$$

$$\textit{selectivity}(\text{L\_PARTNO} = \text{P\_PARTNO} \wedge \text{P\_PARTNO} = \text{S\_PARTNO}) = 6/30 = 1/5 \tag{2.49}$$

Evidently something is not right here, since the third predicate is the conjunction of the first two, and so by (2.46), its selectivity should be *the product of the first two selectivities*, i.e., $1/2 \cdot 1/2 = 1/4$.

The problem is that—*even under the assumption of a fixed database state*—a given predicate's selectivity is not a fixed quantity, but depends on the predicate's context within a query. In particular, as this example shows, the selectivity of one predicate can be influenced by another predicate in the same selection; selectivities can also be influenced by prior selections.

How important are such effects? Since in practice selectivities are only estimates, and are not very accurate in the first place, it cannot do much harm to ignore the kind of variability we have illustrated, provided the discrepancies are small. In the world of join-cardinality estimation, the difference between 1/4 and 1/5 is hardly noticeable.

On the other hand, the sample database in Figure 2.1 could easily have been constructed in such a way as to create a more dramatic discrepancy. Keeping the **Part** relation as is, and holding the selectivities of L_PARTNO = P_PARTNO and P_PARTNO = S_PARTNO at 1/2, the selectivity of their conjunction could have been driven as low as 0 (by making the L_PARTNO and S_PARTNO values disjoint), or as high as 1/2 (by making the L_PARTNO and S_PARTNO values all equal to, say, 7007). Such constructions are somewhat pathological, but they illustrate the potential for error when predicates are assumed independent.

Redundant Predicates Even without pathologies in the stored data in a database, under some circumstances the assumption of predicate independence is not even approximately valid, but is just plain wrong. A case in point is when a query contains *redundant* predicates.

Consider the predicate L_PARTNO = S_PARTNO. We can determine its selectivity by examining the relation in Figure 2.3, which incorporates both the L_PARTNO and S_PARTNO attributes. These two attributes are equal in just 12 of the relation's 30 tuples, and so when considered in isolation from other predicates, the predicate L_PARTNO = S_PARTNO is seen to have a selectivity of $12/30 = 2/5$.

Now consider again the selection result illustrated in Figure 2.4(d). Let E denote the selection in question; i.e.,

$$E = \sigma_{\text{L\_PARTNO=P\_PARTNO} \wedge \text{P\_PARTNO=S\_PARTNO}}(\textbf{LineItem} \times \textbf{Part} \times \textbf{Source}). \tag{2.50}$$

Suppose we were to define a new relation E' as the following selection on E:

$$E' = \sigma_{\text{L\_PARTNO=S\_PARTNO}}(E). \tag{2.51}$$

If we attempted to estimate the cardinality of E' by blindly applying formula (2.40), we would multiply the selectivity 2/5 of the selection predicate by the cardinality 6 of the selection argument, and so obtain an estimated result cardinality of $(2/5) \cdot 6 = 2.4$.

But the true result cardinality is 6; E' will be exactly the same as E. For as we can see from Figure 2.4(d), the L_PARTNO and S_PARTNO attributes of each tuple of E are already equal, and so the selection in (2.51) has no effect. Indeed the predicate L_PARTNO = S_PARTNO *must* be satisfied by the tuples of E, because it is *logically implied*, through transitivity, by the predicates L_PARTNO = P_PARTNO and P_PARTNO = S_PARTNO that were applied in the construction of E.

In this example, the redundant predicate L_PARTNO = S_PARTNO throws off our cardinality estimate by a factor of 5/2. It should be underscored that the only reason the effect is not *worse* is that our examples have dealt with small relations. Our sample database consists of relations with cardinalities of 2, 3, and 5, and not coincidentally, the selectivity values we have encountered have been close to the reciprocals of these figures. If our sample relations had instead had cardinalities on the order of 1000, we could have expected to encounter selectivities on the order of 1/1000. Erroneously incorporating such selectivity values into our cardinality calculations would have caused us to underestimate result cardinalities by several orders of magnitude.

One might think to eliminate the problem of redundant predicates by identifying and removing the redundancies. But redundant predicates actually serve a useful purpose. Consider the following variation on (2.50), in which the Cartesian product has been reordered so that the **LineItem** and **Source** relations are combined first:

$$E_1 = \sigma_{\text{L\_PARTNO=P\_PARTNO}\wedge\text{P\_PARTNO=S\_PARTNO}}((\textbf{LineItem} \times \textbf{Source}) \times \textbf{Part}). \tag{2.52}$$

Pushing down the predicates as far as possible, we obtain

$$(\textbf{LineItem} \times \textbf{Source}) \bowtie_{\substack{\text{L\_PARTNO=P\_PARTNO}\wedge \\ \text{P\_PARTNO=S\_PARTNO}}} \textbf{Part}. \tag{2.53}$$

Even with the predicates pushed down, this expression contains a Cartesian product between **LineItem** and **Source**. But by applying and pushing down the redundant predicate L_PARTNO = S_PARTNO, we can transform (2.53) into

$$E_1' = (\textbf{LineItem} \bowtie_{\text{L\_PARTNO=S\_PARTNO}} \textbf{Source}) \bowtie_{\substack{\text{L\_PARTNO=P\_PARTNO}\wedge \\ \text{P\_PARTNO=S\_PARTNO}}} \textbf{Part}, \tag{2.54}$$

in which the operator between **LineItem** and **Source** has now become a join. As a result, (2.54) can be expected to yield a lower cost estimate than (2.53). In the event that the cost of (2.54) should turn out to be *optimal*, the redundant predicate will have played a vital role—for without this predicate, the optimizer would have produced an inferior "optimum."

The potential benefits of redundant predicates are significant enough that many query processing systems, far from eliminating redundancies, actively seek them out [7, 16]. These systems analyze the predicates that appear explicitly in a user query, and then construct *additional* predicates that can be inferred from the explicit ones. Because of such policies, it is essential that an optimizer be prepared to make adjustments for predicate redundancies when carrying out cardinality estimation.

Foreign-Key Predicates The technical problems noted above reflect a deeper conceptual difficulty with the notion of selectivity. One can appreciate this conceptual difficulty

by taking a closer look at the behavior of the foreign-key predicates we have been using in our examples.

Consider once again the join

$$\textbf{LineItem} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} \textbf{Part}, \tag{2.55}$$

whose result is depicted in Figure 2.4(a). In light of the fact that L_PARTNO is a foreign key referencing the P_PARTNO attribute of the **Part** relation, one can deduce that the cardinality of the join result *must* be the same as that of **LineItem**: for each tuple in **LineItem** (*cf.* Figure 2.1), there is *exactly one* tuple in **Part** whose P_PARTNO attribute agrees with the foreign key L_PARTNO. The join between **LineItem** and **Part** may be conceived of as "filling out" the **LineItem** relation with additional information about each part it mentions—in other words, the intent of this join is to *widen* the **LineItem** relation without adding or subtracting any tuples.

However, the notion of selectivity disregards such semantic considerations. It uniformly imposes the view that a join result represents some *fraction* of the tuples of a Cartesian product. Formula (2.42) would have us compute the cardinality of (2.55) as

$$s \cdot |\textbf{LineItem}| \cdot |\textbf{Part}|, \tag{2.56}$$

where s is the selectivity of the predicate L_PARTNO = P_PARTNO. For this formula to work here, s must necessarily be the reciprocal of $|\textbf{Part}|$, since we have already established that the join in (2.55) preserves the cardinality of the **LineItem** relation. In this situation, the value of the selectivity s has nothing to do with the join predicate *per se*; instead, the role of the selectivity is to *undo* the effect of multiplying $|\textbf{LineItem}|$ by $|\textbf{Part}|$.

There is nothing harmful about computing selectivities by working backwards from a known result cardinality, provided that it is possible to do so. But the motivation for introducing selectivities in the first place was that estimating result cardinalities is exceedingly difficult—so working backwards from result cardinalities plainly cannot be a good general strategy. The concept of selectivity attempts to grapple with the estimation problem by treating selection as quasi-stochastic; unfortunately, the semantic content of selection predicates often makes their behavior highly non-stochastic.

2.4.3 Discussion and Resolution

In their classic paper on the System R optimizer, Selinger et al. [50] make an intriguing observation regarding errors in cost estimation. Although the System R optimizer proved to be rather poor at estimating the costs of plans, Selinger et al. found that the *rankings* of plans by their estimated costs tended to coincide with rankings based on true costs. Consequently the optimizer's inability to judge costs accurately usually did not impair its ability to find an optimal plan.

However, there are certainly situations in which inaccurate cost estimates have adverse effects, and research since the time of System R has sought to improve the quality of cardinality estimates (and hence cost estimates) through a variety of sophisticated techniques [5, 14, 31, 35, 47]. Antoshenkov [1] goes further, and cites instability in cardinality computations as grounds for rejecting point-valued estimates altogether. Noting that roughly half the problem reports regarding query processing in DEC's Rdb system were related to errors in cost estimation, Antoshenkov mentions cases in which the optimizer chose query plans that were suboptimal by several orders of magnitude. Ultimately the responsibility for these poor plan choices lay with wildly inaccurate cardinality computations.

Antoshenkov's solution to the problem is complex, and involves cooperation between the optimizer and the plan-execution component of a query-processing system. As such, his solution is incompatible with the conventional decomposition of query processing into three independent phases. In the present work, we will stick to the traditional style of estimation based on selectivities, despite its inadequacies. Selectivity-based estimation remains standard in the join-optimization research literature.

In the following, we will generally assume predicate selectivities to be fixed, as is done traditionally. However, we make an exception for the case of redundant predicates, whose practical importance cannot be ignored. We shall assume that any identification or construction of redundant predicates has already taken place before invocation of our optimization algorithms. These algorithms must be informed of redundancies by the caller; they will then take measures to ensure that selectivities are appropriately adjusted during the calculation of join-result cardinalities.

2.5 Join Graphs

Next we turn our attention to *join graphs*, which provide a way of representing the predicate relationships among the different relations in a query.

2.5.1 Concept of Join Graphs

Consider the query

$$\begin{aligned}&\textbf{Customer} \bowtie_{\text{C\_CUSTNO=O\_CUSTNO}} \textbf{Order} \bowtie_{\text{O\_ORDERNO=L\_ORDERNO}} \\ &\quad \textbf{LineItem} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} \textbf{Part} \bowtie_{\text{P\_PARTNO=S\_PARTNO}} \textbf{Source.}\end{aligned} \tag{2.57}$$

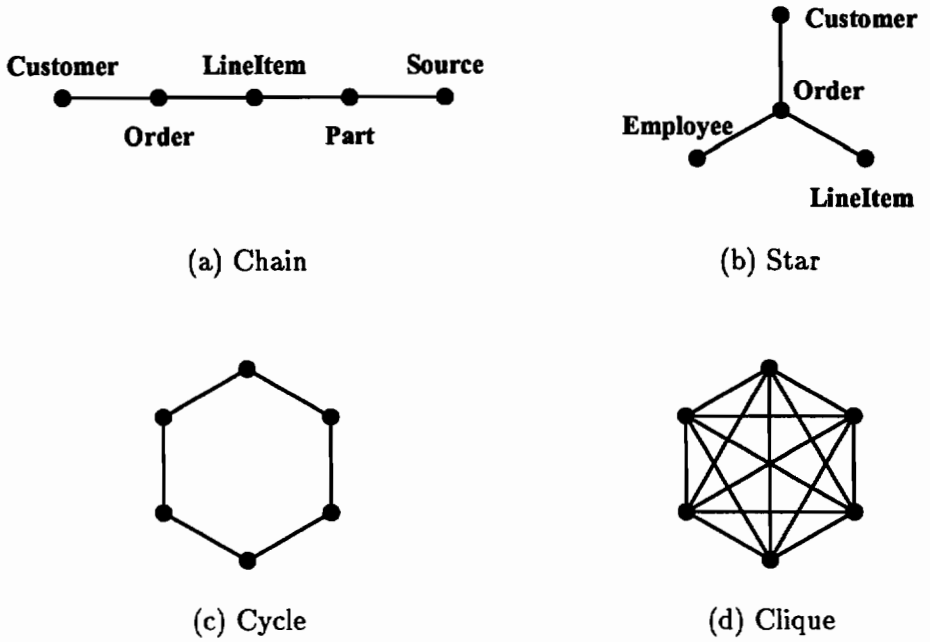
A *join graph* (or *query graph*) for this query appears in Figure 2.7(a). The nodes of the graph represent relations, and the edges represent join predicates. For example, the predicate C_CUSTNO = O_CUSTNO appears as an edge connecting the **Customer** and **Order** nodes because this predicate mentions attributes in those two relations. When a query's graph has a simple linear form, as in this instance, the query is called a *chain query.*

By contrast, a query such as

$$\begin{aligned}&(\textbf{Customer} \bowtie_{\text{C\_CUSTNO=O\_CUSTNO}} \textbf{Order} \bowtie_{\text{O\_SOLDBY=E\_EMPNO}} \\ &\quad \textbf{Employee}) \bowtie_{\text{O\_ORDERNO=L\_ORDERNO}} \textbf{LineItem},\end{aligned} \tag{2.58}$$

whose graph (Figure 2.7(b)) has a single relation at its hub, with spokes radiating outward to the remaining relations, is said to be a *star query.* Other join graphs whose topologies are sufficiently distinctive to earn them special titles are the *cycle* and the *clique* (Figures 2.7(c) and (d)). In a clique, every pair of relations is related by some predicate.

It should be noted that the relations at the nodes of a join graph need not all be distinct, since a query may involve multiple instances of the same relation under different aliases. However, it is always the case that an n-way join maps to an n-node join graph. To reduce confusion, we shall assume in the following that the n relations in a query are in fact distinct; but our results in no way depend on this assumption.



(a) Chain (b) Star

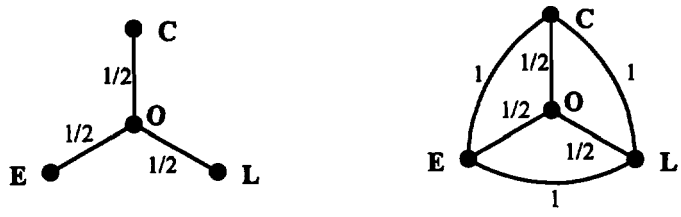
(c) Cycle (d) Clique

Figure 2.7: Join graphs

2.5.2 Edge-labeled Join Graphs

It is sometimes convenient to label the edges of a join graph with the corresponding predicate selectivities. For example, Figure 2.8(a) shows such a labeling of the join graph for (2.58). (The relation names have been abbreviated to reduce clutter.) With the addition of the selectivity labels, we are better equipped to understand the relationship between different kinds of join graphs. Indeed, we shall see shortly that an edge labeled with selectivity 1 is equivalent to no edge at all. Figure 2.8(b) uses this fact to transform the graph of (2.58) into an equivalent clique with three dummy edges. But what is the justification for these claims of equivalence?

The answer is that each dummy edge may be thought of as representing a predicate such as $(\text{C\_NAME} = \text{C\_NAME}) \wedge (\text{E\_NAME} = \text{E\_NAME})$, which induces a connection between **Customer** and **Employee** since it mentions an attribute from each. This predicate, and others of the same flavor, have selectivity 1 because they are vacuously true—hence the labels of 1 on the dummy edges. It should now be evident that since the meaning of a query is unchanged by the addition of vacuously true predicates, Figure 2.8(b) is (in some



(a) Query (2.58) (b) Query (2.58) as a clique

Figure 2.8: Join graphs labeled with selectivities

sense) just as good a graph for query (2.58) as Figure 2.8(a).

A corollary of this observation is that the notion of a join graph is somewhat ill-defined, inasmuch as different but algebraically equivalent formulations of the same query may have different join graphs. Still, the concept of join graphs has its uses, which we touch on next.

2.5.3 Role of the Join Graph

Join graphs are of interest for at least two reasons. First, a labeled join graph concisely captures much of the information needed to specify a given join-optimization problem. In fact, Steinbrunn [54] characterizes the input to a join optimizer as being exactly the join graph. In most of what follows, that characterization works well for us, too, provided that we think of our graphs as being labeled with relation cardinalities in addition to relation names. But at certain points—to handle predicate redundancies, for example—we will need to supplement the join graph with further information (or use more elaborate labels).

The second reason why join graphs are of interest is that the speed or effectiveness of a join optimizer often varies with the topology of the join graph. Graphs of large *diameter*, such as chains and cycles, are easier to cope with than small-diameter graphs such as stars. Graphs of low density (i.e., with few predicates) also tend to be easier to cope with than high-density graphs.

The worst-case join graph by any measure is the clique, for it is both minimal in diameter and maximal in density. Just how important cliques are in practice is not entirely clear. In queries involving many relations, cliques can arise only in the presence of an

absurd number of predicates—for example, a 15-way join does not become a clique until it has $\binom{15}{2} = 105$ predicates. But if a query-processing system automatically generates new predicates by exploring the implications of a query's explicit predicates, cliques could come about routinely. Furthermore, as illustrated above in Section 2.5.2, any query at all can be turned into a clique through the addition of vacuous predicates. So whether or not cliques are truly important in their own right, an optimizer's ability to accommodate them serves as an indication of its resiliency in the face of arbitrary join graphs.

2.5.4 Complex Predicates and Hyperedges

It was an unstated assumption of the previous sections that each join predicate in a query mentions attributes from precisely two relations; and further, that given a pair of relations, there may be at most one predicate relating them. If these assumptions were not satisfied, the correspondence between predicates and join-graph edges would fall apart.

One can easily imagine reasonable collections of predicates that do not conform to these assumptions, although often in such situations conformance can be achieved through minor rearrangements. To take a very simple example, consider the pair of predicates A_CITY = B_CITY and A_STATE = B_STATE, where the attributes A_CITY and A_STATE belong to some relation A, and B_CITY and B_STATE to B. This pair is not permissible, since both predicates refer to the two relations A and B. But by conjoining these two predicates as (A_CITY = B_CITY) $\wedge$ (A_STATE = B_STATE), one obtains a single predicate, which then conforms to our requirements. Under the assumption of predicate independence, the selectivity of this conjunction is just the product of the selectivities of the original two predicates.

Conversely, a predicate of the form A_VALUE < B_VALUE < C_VALUE, which is equivalent to (A_VALUE < B_VALUE) $\wedge$ (B_VALUE < C_VALUE), violates the restriction that a predicate may refer to only two relations. But this time conformance can be achieved by splitting the conjunction into two separate predicates A_VALUE < B_VALUE and B_VALUE < C_VALUE. (Determining the selectivities appropriate to these predicates is nontrivial, but that cannot be helped.)

These examples show that our assumptions about the forms that predicates may take

are perhaps not excessively restrictive. Nevertheless, they are restrictive to some degree, and preclude predicates such as A_VALUE + B_VALUE < C_VALUE, which is inherently ternary and cannot be split in two. In the framework of join graphs, ternary predicates correspond to *hyperedges*—edges with possibly more than two "endpoints"; graphs containing such edges are called *hypergraphs* [38]. So to support ternary, and more generally, n-ary predicates, an optimizer would have to accept join hypergraphs as its input.

Hypergraphs have a variety of applications in query processing besides the representation of n-ary predicates. For this reason, one does encounter mention of hypergraphs in the query-optimization research literature—but rarely in connection with n-ary predicates. Research on join-order optimization techniques appears to make no provisions for hypergraphs or hyperedges [54]. The present work will likewise focus on conventional join graphs with binary edges.

2.6 Cost Models and Physical Properties

2.6.1 Cost Models

It was observed above that one may think of the input for a join-optimization problem as being a labeled join graph. However, quite apart from such complications as redundant predicates, which we leave aside for the present, the labeled join graph does not by itself determine a least-cost plan for the join. The best plan also depends on what *cost model* is used to estimate the expected CPU and disk time consumption of alternative plans. As we have noted, there are many join-processing algorithms [43], and as many models (or more) for estimating their costs. Consequently, two optimizers, given the same join graph as input, could generate two completely different join-evaluation strategies as output, and yet each might be able to claim (legitimately!) that it had found the unique least-cost plan—according to its own cost model.

Fair comparison of different optimizers requires use of the same cost model in each. In a sense, one would like to treat the cost model as an optimizer input along with the join graph. It may not be realistic (or even desirable) to expect an optimizer to accept this input in the form of a run-time parameter of the optimizer invocation. But it is desirable

for an optimization technique to be adaptable to a variety of cost models, even if the adaptation entails changing a portion of the optimizer implementation. Such flexibility not only facilitates comparisons with other optimization techniques, but also simplifies optimizer maintenance in a changing environment.

Join-optimization techniques permit flexibility in the cost model to varying degrees. We shall touch further on this matter in Section 2.7 below. For now, we introduce the approach to cost modeling that we will rely on in the present work.

2.6.2 A Generic Cost Model

Following Steinbrunn [54, 55], the present work will allow for flexibility in the cost model by making use of a single *generic* cost model. This generic model is parameterized by a *cost function*, as described below.

Premises and Cost Equations Our generic cost model depends on the following three premises:

- Each join operator in a query plan entails a cost that is independent of the costs of the other operators in the plan, and that can be determined by a *cost function* $\kappa(R_{out}, R_{lhs}, R_{rhs})$. The arguments of the cost function are, respectively, the result of the join, its left-hand input, and its right-hand input.

- The cost of fetching each base relation from the database is independent of the join order—the data from each relation must be fetched regardless of how the relations are combined. Therefore, the cost of accessing the base relations is immaterial, and may be taken to be zero. (By taking the cost of accessing base relations to be zero, we will underestimate the costs of all plans, but always by the same amount—so the outcome of optimization will be unaffected.)

- The cost of a plan is simply the sum of the costs of the operators in the plan.

Note that none of these premises is strictly valid. Because of considerations such as memory contention, operator pipelining, and interactions of join algorithms with data-fetching mechanisms, our somewhat simplistic framework will not be able to capture all

the fine points of cost modeling. But through parameterization of the cost function κ, it affords enough generality to subsume most of the cost models that appear in the literature.

Our premises yield the following recursive definition of query-plan cost:

$$cost(R) = 0 \tag{2.59}$$

$$cost(E_0 \bowtie E_1) = cost(E_0) + cost(E_1) + \kappa([\![E_0 \bowtie E_1]\!], [\![E_0]\!], [\![E_1]\!]), \tag{2.60}$$

where R names a base relation, E_0 and E_1 are subplans, and $[\![E]\!]$ is the *denotation* of E (i.e., the relation to which E evaluates). Equations (2.59) and (2.60) shall provide the basis for all our discussions of cost in the remainder of this work.

Application of the Cost Equations As an illustration of the application of (2.59) and (2.60), let us introduce a naive cost function κ_0 that makes the cost of evaluating a given join equal to the cardinality of the result. That is,

$$\kappa_0(R_{out}, R_{lhs}, R_{rhs}) = |R_{out}|. \tag{2.61}$$

Given this cost function, let A and B be base relations, and consider the cost of the plan $A \bowtie B$ whose result is some relation that we shall refer to as R_{ab}. Then by (2.59) and (2.60), we obtain

$$cost(A \bowtie B) = cost(A) + cost(B) + \kappa_0([\![A \bowtie B]\!], [\![A]\!], [\![B]\!]) \tag{2.62}$$

$$= 0 + 0 + \kappa_0(R_{ab}, A, B) \tag{2.63}$$

$$= |R_{ab}|. \tag{2.64}$$

That is, the cost of the plan is equal to the cardinality of the result. Going a step further, let C be another base relation, and let us then consider the cost of the plan $(A \bowtie B) \bowtie C$, whose final result we shall call R_{abc}. Now we have

$$cost((A \bowtie B) \bowtie C) = cost(A \bowtie B) + cost(C) + \kappa_0([\![(A \bowtie B) \bowtie C]\!], [\![A \bowtie B]\!], [\![C]\!]) \tag{2.65}$$

$$= |R_{ab}| + 0 + \kappa_0(R_{abc}, R_{ab}, C) \tag{2.66}$$

$$= |R_{ab}| + |R_{abc}|. \tag{2.67}$$

The cost of the compound plan proves to be the sum of the costs of its two join operations; these costs are equal, respectively, to the cardinality of the intermediate result R_{ab}, and to the cardinality of the final result R_{abc}.

Multiple Physical Join Operators On the surface, it may appear that our "generic cost model" supports only those cost models that assume a single physical join operator. However, one can support multiple physical join operators under the generic cost model by recognizing that the applicable physical operator for a given join is always the *least costly* of the available alternatives (*cf.* Section 2.3.2).

For example, suppose a query-processing system supports two physical join operators, $\bowtie^{hash}$ and $\bowtie^{nl}$ (where *nl* stands for *nested-loops*). Suppose that the costs of these operators are modeled respectively by the cost functions κ_{hash} and κ_{nl}. Then we may define a *new* cost function

$$\kappa_{best}(R_{out}, R_{lhs}, R_{rhs}) = \min(\kappa_{hash}(R_{out}, R_{lhs}, R_{rhs}), \kappa_{nl}(R_{out}, R_{lhs}, R_{rhs})) \tag{2.68}$$

that represents the cost of the best physical instantiation of a given generic join operator. By extending this technique, one can accommodate an arbitrary number of alternative physical operators. At the conclusion of optimization, each generic join operator in the optimal query plan can be replaced by the physical operator that yields the lowest cost for the join in question. Thus, neither the assumption of a single, generic join operator, nor that of a single, generic cost function, necessarily precludes the modeling of multiple physical join operators.

Abstract Interpretation of the Cost-Function Arguments Now it may be a matter of some concern that if we interpret (2.60) literally, it seems to require full evaluation of the query plan whose cost we are trying to estimate. That is, the arguments of κ are *relations*, and the only way to obtain those relations is to evaluate each join in the plan.

On the other hand, we saw above that under our naive cost model with cost function κ_0, the cost of the plan $(A \bowtie B) \bowtie C$ was simply $|R_{ab}| + |R_{abc}|$. Hence, to estimate the plan cost in this instance, it would suffice just to compute (or estimate) the *cardinalities*

of R_{ab} and R_{abc}. If we went to the trouble of computing the relations themselves, and not just the cardinalities, we would waste a tremendous amount of effort.

It is possible to interpret (2.60) in a way that does not necessitate full evaluation of the relations. We may regard the denotation brackets $[\![\cdot]\!]$ as designating an *abstract interpretation* [9] of the expression they surround, and we may take the arguments of the cost function κ to be not relations, but *partial representations of relations.* In the example at hand, the partial representation of a relation might consist of its cardinality, together with additional information to assist in estimating the cardinalities of subsequent join results. This representation presumably would *not* contain any tuples of the relation. The notation $|R_{out}|$ in (2.61) should then be understood as the extraction of the cardinality from the representation R_{out}, and not as a tuple-counting operation.

Because cardinalities are so important to cost estimation, they will play a role in any imaginable cost model. From the standpoint of code reuse, it therefore makes sense for an optimizer to provide a cardinality-estimation mechanism separate from the code that implements individual cost models. The algorithms that we shall present in this work will incorporate cardinality-estimation mechanisms independent of the cost model.

But from the standpoint of program decomposition, the estimation of cardinalities ought by all rights to fall within the purview of the cost model. Our optimization techniques are in no way concerned with cardinalities for their own sake; we compute the cardinalities simply so that we can make them available to the cost function, which may then do with them as it pleases.

One can conceive of cost models that take into account not only the *cardinalities* of join inputs and join results, but also their *widths*, or indeed other properties that might affect costs. There is nothing in our framework that precludes such models. Our code to compute cardinalities should be seen as a replaceable module whose purpose is to meet the needs of the cost model, whatever those might happen to be. In this sense, the code to compute cardinalities is not really a part of our optimization algorithms, but serves as an illustration of one way in which these algorithms may be parameterized.

2.6.3 Physical Properties

Properties such as cardinality are known as *logical properties* of relations because they are independent of implementation considerations; one can speak of the cardinality of a relation in the abstract even if the relation is never stored in a computer. But it makes no sense to speak of the *sort order* or *location* of a relation when the relation is viewed as a mathematical abstraction. Such properties make sense only with reference to a relation's physical representation in memory, or on disk, or on a print-out. Consequently these properties are called *physical properties.*

If one ignores physical properties, then optimality in a join plan implies optimality in its subplans as well, exactly as one would expect. Consider the case where the best plan for joining A, B, C, and D turns out to be $(B \bowtie (A \bowtie C)) \bowtie D$. In this case one may deduce that the best plan for joining A, B, and C must necessarily be $B \bowtie (A \bowtie C)$. For suppose there were a better plan—say $C \bowtie (B \bowtie A)$. Then $(C \bowtie (B \bowtie A)) \bowtie D$ would have to be a better plan than $(B \bowtie (A \bowtie C)) \bowtie D$, since the only difference between the two is that the new plan computes its left-hand input more efficiently. Thus, in supposing $B \bowtie (A \bowtie C)$ to be suboptimal, we have arrived at a contradiction; it follows that $B \bowtie (A \bowtie C)$ is optimal after all.

The necessity of optimality in all subplans of an optimal plan is referred to as the *principle of optimality* [15, 24]. Unfortunately, the principle of optimality breaks down when costs are influenced by physical properties. Imagine that the join of the previous paragraph were being performed on a distributed system consisting of two processing units named t and k (for *Tokyo* and *Kyoto*).[1] In this scenario, one of the physical properties of each relation is its *location.*

Let us say that A and B reside on processing unit t in Tokyo, while C and D reside on k in Kyoto; as a mnemonic device we will now refer to these relations as A_t, B_t, C_k, and D_k. Part of the join optimizer's job is then to determine at what location each join should be performed. A join at processing unit t will be denoted $\overset{t}{\bowtie}$, and at k, $\overset{k}{\bowtie}$. The inputs to a given join operation need not be computed (or reside) at the location where

[1] This example is adapted from Lanzelotte et al. [34].

the join will take place, but remote inputs will incur a communication penalty that will drive up the estimated cost of the join. Under these circumstances, it is imaginable that

$$C_k \overset{t}{\bowtie} (B_t \overset{t}{\bowtie} A_t) \tag{2.69}$$

could be a better plan than

$$B_t \overset{k}{\bowtie} (A_t \overset{k}{\bowtie} C_k), \tag{2.70}$$

and yet be less well suited to a subsequent join with D_k: for the final join result in (2.70) is computed in Kyoto, where D_k already resides, while the final result in (2.69) would have to be transferred from Tokyo to Kyoto (or else D_k would have to be transferred from Kyoto to Tokyo). Consequently, it is possible for $(B_t \overset{k}{\bowtie} (A_t \overset{k}{\bowtie} C_k)) \overset{k}{\bowtie} D_k$ to be optimal while $B_t \overset{k}{\bowtie} (A_t \overset{k}{\bowtie} C_k)$ is suboptimal.

This problem is not special to distributed join processing. One encounters similar phenomena in connection with sort orders, or with any other physical property.

Some of the join-optimization techniques described in the literature make provisions for physical properties, while others do not. The present work does not address the issue of physical properties, although in Chapter 10 we do comment on what might be involved in adapting our techniques to accommodate them.

2.7 Approaches to Join-order Optimization

We now briefly discuss the different approaches that have been taken to join-order optimization. Some of the techniques we describe are specific to join-order optimization; others are more general, and were designed to optimize queries involving arbitrary operators, not just join operators. However, as join-order optimization remains a central concern for all query optimizers, it makes sense to include general-purpose query-optimization techniques in our discussion.

2.7.1 Dynamic Programming

System R The System R optimizer [50] was the first to apply *dynamic programming* to join-order optimization. This optimizer constructed only left-deep plans, and excluded

Cartesian products except where they could not be avoided.

In Chapter 3, we shall discuss in detail our own approach to optimization by way of dynamic programming. Here we give only a brief sketch of the principle involved. System R and other dynamic programming optimizers proceed in *phases*. In the early phases, they build candidate subplans involving just a few relations; in later phases, they build larger candidate subplans involving many relations. The larger subplans are built up from the smaller ones in such a way that the subplans retained at the end of each phase are optimal. The final phase constructs a plan involving all the relations in the query; this plan is then optimal for the query. For obvious reasons, this approach to optimization is often referred to as *bottom-up* optimization.

For example, suppose System R were to optimize the join of relations A, B, C, and D. In Phase 1, it would find the optimal manner of accessing the data in each of the individual relations A, B, C, and D. In Phase 2, it would find optimal subplans for joins involving *two* relations; i.e., for the join of A and B, for the join of B and C, for the join of C and D, and so on. In Phase 3, it would find optimal subplans for joins of *three* relations; i.e., for the join of A, B, and C, for the join of B, C, and D, and so on. In Phase 4, it would find an optimal plan for the join of A, B, C, and D. This final plan would contain subplans that had been constructed in the earlier phases of optimization.

Starburst and the Complexity of Dynamic Programming The Starburst optimizer [45] extended the techniques used in System R and provided greater generality and flexibility. When requested to do so, the Starburst optimizer can produce bushy plans, as well as plans that contain arbitrary Cartesian products. But the mechanisms used in the Starburst optimizer are conceptually no different from those used in the System R optimizer.

Ono and Lohman [44, 45] analyze the time complexity of optimization in Starburst under a variety of circumstances. For chain join graphs, the time complexity is $O(n^2)$ (where n is the number of relations in the query) to find optimal left-deep plans without Cartesian products, and $O(n^3)$ to find optimal bushy plans without Cartesian products. These cases are noteworthy because the complexity is polynomial. For star join graphs, the

time complexity is $O(n2^n)$ to find optimal plans without Cartesian products. Interestingly, the complexity in the case of a star is independent of whether the search is confined to left-deep plans. When arbitrary Cartesian products are considered, the time complexity of optimization becomes $O(n2^n)$ for left-deep search, and $O(3^n)$ for bushy search, regardless of the join graph. The complexity figures that apply when arbitrary Cartesian products are considered also apply when the join graph is a clique. Thus $O(n2^n)$ represents the worst-case complexity for a left-deep search, and $O(3^n)$ the worst case for a bushy search.

Ono and Lohman do not discuss the space complexity of Starburst, but it is almost certainly $O(2^n)$ in the worst case. We base this conjecture on the similarity between the information stored by Starburst and that stored by the dynamic programming algorithm we shall present and analyze in Chapter 3 below.

Problem with the Starburst Complexity Figures Unfortunately, the Starburst time-complexity figures given by Ono and Lohman do not tell the whole story. Ono and Lohman base their time-complexity figures on the number of "feasible joins" for combining subplans to obtain larger plans. A "feasible join" is a join expression that meets the criteria of the optimization problem at hand—for example, in a left-deep search, only a left-deep join expression would be considered "feasible," and in a search that excluded Cartesian products, a "feasible" join expression would have to be free of Cartesian products. Their premise is that in the course of optimization, the optimizer will examine each feasible join, and that each feasible join can be examined in constant time. Hence the time required for optimization should be proportional to the number of feasible joins.

However, before the feasible joins can be examined, they must be *constructed,* or *enumerated.* Ono and Lohman present pseudo-code for enumerating the feasible joins [44], but disregard the time contribution of this code. Analysis by the present author reveals that in general, enumerating the feasible joins can take more time than examining them; the worst-case time complexity of feasible-join enumeration in Starburst is $O(4^n)$. Intuitively, the reason for this phenomenon is that Starburst's algorithm for enumerating feasible joins uses a generate-and-filter mechanism. It generates all the joins that *might* be feasible, discarding those that are not. In the worst case, it discards many more joins

than it retains. Appendix A gives the details of the author's analysis.

2.7.2 Rule-based Optimization

Principle of Rule-based Optimization and Application to Join Optimization *Rule-based* optimizers such as Exodus [19] and Volcano [20, 40] strive for extensibility through a general-purpose search mechanism that can easily be reconfigured to accommodate new operators and new transformation rules. Configuration is accomplished through a special rule-definition file, or through a collection of functions that define the operators and transformation rules of the query algebra. Given an input expression over the operators of the algebra, the optimizer applies the specified transformation rules until it has explored the space of all expressions equivalent to the input. The optimizer attempts to find a low-cost plan corresponding to each expression encountered, and chooses the best such plan as the optimum for the given input expression.

The standard approach to configuring a rule-based optimizer for join-order optimization is to include the join commutativity and associativity laws, (2.17) and (2.28), as transformation rules. As we remarked earlier, join associativity can be stated in several ways; the formulation given in (2.28),

$$(A \bowtie_p B) \bowtie_q C = A \bowtie_{p'} (B \bowtie_{q'} C), \tag{2.71}$$

is the one that corresponds most closely to the formulation ordinarily used in rule-based optimizers. Recall, however, that this formulation of join associativity had a well-formedness condition attached to it, as well as a constraint relating p and q to p' and q'. Rule-based optimizers in the style of Exodus and Volcano require that segments of code (e.g., C code) be supplied along with the transformation rules to ensure satisfaction of any preconditions on rule applicability, and to compute any portions of the transformed expression (such as the predicates p' and q') that cannot be obtained automatically through application of the rule.

Complexity of Volcano with Standard Rule Sets One of Volcano's principal innovations was the use of *memoization* to improve the performance of rule-based query

optimization. In Volcano (unlike Exodus), representations of subexpressions are shared across all expressions in which they occur; in addition, all possible transformations of a given subexpression are stored with the subexpression, so that when this subexpression is next encountered, none of these transformations need to be repeated.

Volcano's use of memoization has the interesting consequence that the *space* complexity of join-order optimization in Volcano corresponds to the *time* complexity figures given by Ono and Lohman, as noted above. In effect, Volcano's memo structures store representations of the "feasible" joins in the sense of Ono and Lohman, and each such join is represented exactly once. It follows that space complexity for bushy join-order optimization in Volcano is $O(3^n)$ in the worst case.

McKenna [20, 40] carried out extensive empirical studies of Volcano performance, which proved to be roughly comparable to that of Starburst. But the first *analytical* treatment of the subject was given by Pellenkoft, Galindo-Legaria, and Kersten [46], who have shown Volcano's worst-case time complexity in bushy join-order optimization to be $O(4^n)$. (The present author [61] had previously conjectured, incorrectly, that Volcano's time complexity was just a constant multiple of its space complexity—hence $O(3^n)$.) We thus see that Starburst and Volcano yield a worst-case time complexity of the same order—$O(4^n)$—despite their use of unrelated search algorithms. There is no evident structural explanation for this similarity in performance; it appears to be purely coincidental.

Duplicate-free Rule Sets In the case of Volcano, the fact that the time complexity grows faster than the number of feasible joins is due to the existence of multiple distinct paths of transformation steps from a given expression to another, equivalent expression. For example, suppose relation A is connected to B through predicate p, and B to C through predicate q. Then the expression $(A \bowtie_p B) \bowtie_q C$ can be transformed into its mirror-image, $C \bowtie_q (B \bowtie_p A)$, through any one of the following paths of transformation steps using the join commutativity rule (2.17) and the join associativity rule (2.28):

1. $(A \bowtie_p B) \bowtie_q C \quad \rightsquigarrow \quad (B \bowtie_p A) \bowtie_q C \quad \rightsquigarrow \quad C \bowtie_q (B \bowtie_p A).$

2. $(A \bowtie_p B) \bowtie_q C \quad \rightsquigarrow \quad C \bowtie_q (A \bowtie_p B) \quad \rightsquigarrow \quad C \bowtie_q (B \bowtie_p A).$

$$3.\ (A \bowtie_p B) \bowtie_q C \quad \rightsquigarrow \quad A \bowtie_p (B \bowtie_q C) \quad \rightsquigarrow \quad A \bowtie_p (C \bowtie_q B) \quad \rightsquigarrow$$
$$(C \bowtie_q B) \bowtie_p A \quad \rightsquigarrow \quad C \bowtie_q (B \bowtie_p A).$$

(Additional paths are also possible.) Note that paths 1 and 2 are essentially the same. One may think of $(A \bowtie_p B) \bowtie_q C$ as being the join $E \bowtie_q C$, where E in turn is the join $A \bowtie_p B$. What happens in path 1 is that first A and B are commuted inside E to give a new subexpression E', and then E' is commuted with C. In path 2, by contrast, first E is commuted with C, and then A and B are commuted inside E to give E'. In other words, both paths involve the same pair of *independent* transformation steps; these independent steps are just carried out in different orders. Most rule-based optimizers work in such a way that no duplication of effort is entailed by the existence of distinct paths such as paths 1 and 2, in which the only difference is in the ordering of independent transformation steps.

However, path 3 differs more fundamentally from paths 1 and 2, and entails duplication of effort. To circumvent this kind of duplication of effort, the present author [60] and Galindo-Legaria and his colleagues [13, 46] have independently devised formulations of join commutativity and associativity under which transformation paths obey the following property: The path from an expression E to another expression E', if one exists, is *unique* up to reordering of independent transformation steps. Pellenkoft et al. refer to a formulation that yields this property as a *duplicate-free* rule set.

When Volcano is configured with a duplicate-free rule set for join commutativity and join associativity, its worst-case time complexity drops to $O(3^n)$. But there is a catch. Under the conventional formulation of join commutativity and associativity, if E and E' are equivalent join expressions, and neither one contains Cartesian products, then there is guaranteed to exist a path from E to E' such that no expression along the path contains Cartesian products.[2] This guarantee is lost under duplicate-free formulations. Consequently, for queries that yield a low optimization complexity when Cartesian products are disallowed (e.g., chain queries), optimization effort could be much greater under duplicate-free rule sets than under conventional rule sets.

[2]This guarantee is taken for granted in the literature, though to the author's knowledge, it is nowhere proved; nor is the author aware of any simple proof of its validity.

At this writing, it is not known whether it is possible to construct a rule-based join optimizer, or indeed *any* join optimizer, such that with Cartesian products excluded, optimization effort is bounded by a constant multiple of the number of feasible joins in an arbitrary join query.

The following questions related to duplicate-free rule sets also remain open:

1. Under what conditions do duplicate-free rule sets exist? Is there an algorithm for determining whether an arbitrary, given rule set can be reformulated as a duplicate-free rule set?

2. In those cases where a duplicate-free formulation of a given rule set exists, is there an algorithm for constructing the duplicate-free formulation mechanically?

The duplicate-free rule sets that have been devised to date have been *ad hoc* constructions. It appears difficult to extend them with additional transformation rules without compromising the duplicate-free property.

2.7.3 Heuristic and Sequencing Techniques

The approaches to join-order optimization discussed above differ in their search techniques, but both perform an *exhaustive search.* It is natural to wonder whether it is possible somehow to *calculate* or *construct* an optimal plan (or nearly optimal plan), without having to go to the trouble of searching for it.

Heuristic join-optimization techniques aim to obtain satisfactory query plans by direct construction. Typically they use cardinalities and predicate selectivities as guidelines in deciding how to proceed, but without performing any explicit cost estimation. A consequence of the lack of cost estimation is that heuristic join optimizers are somewhat inflexible: they cannot be adapted to different cost models, and hence cannot take into account the performance of new join algorithms that might become available.

Steinbrunn [54, 55] surveys a variety of heuristic techniques, explaining their mechanisms and examining their performance. As Steinbrunn shows, these techniques are extremely fast—which one would expect, since they have no searching to do—but generally deliver query plans of very poor quality. When cost estimates are applied to the plans

obtained, the estimates are often orders of magnitude higher than those of the optimal plans for the same queries. Because these techniques yield such poor results, we shall discuss them no further.

Remarkably, however, there is an optimization technique that, with essentially no search, can directly construct *optimal* join plans—though only under special conditions. We shall refer to this technique and its derivatives as *sequencing techniques*. The original application of sequencing to join-order optimization, which is due to Ibaraki and Kameda [25], imposes the following restrictions:

- The join graph must be acyclic.
- Only left-deep plans are considered.
- Cartesian products are excluded.
- The cost function κ must be expressible as $\kappa(R_{out}, R_{lhs}, R_{rhs}) = |R_{lhs}| \cdot g_p(R_{rhs})$ for some family of functions g_p, where p designates the predicate that qualifies the join of R_{lhs} and R_{rhs}. (Since the plans must be left-deep, R_{rhs} will always designate an individual relation; and since the join graph must be acyclic, only a single predicate will belong to each join in any plan that is free of Cartesian products.)

Ibaraki and Kameda observed that under these assumptions, the problem of join-order optimization *almost* reduces to a sequencing problem that appears in the operations-research literature. The sequencing problem in question can be solved by an algorithm of time complexity $O(n \log n)$, where n corresponds to the number of nodes in the join graph. But to map the join-optimization problem to the sequencing problem, one must designate an "initial" node in the join graph. Since there is no way to determine the best "initial" node *a priori*, the technique of Ibaraki and Kameda applies the sequencing algorithm for each possible choice of "initial" nodes, and takes the best result. Thus, the sequencing algorithm is applied n times, and so the net time complexity of join-order optimization by this technique is $O(n^2 \log n)$.

Krishnamurthy, Boral, and Zaniolo [33] subsequently noticed that a portion of the computation in each of the n applications of the sequencing algorithm was redundant. By

eliminating this redundancy, they obtained an $O(n^2)$ algorithm for join-order optimization, given the same restrictions as in the method of Ibaraki and Kameda. Krishnamurthy et al. also sought to extend the technique by relaxing these restrictions; for example, they propose a mechanism for accommodating arbitrary join graphs, and not just acyclic ones. But in the process, they give up the guarantee of optimality, and thus their technique becomes a heuristic one. Steinbrunn's measurements [54] show that this heuristic extension generates plans of mediocre quality.

More recently, Swami and Iyer [59] have proposed another technique based on sequencing. Their approach begins by applying the technique of Krishnamurthy et al., and then seeks to improve the resulting plan by perturbing it in small ways. In its use of perturbations to improve a plan, the technique of Swami and Iyer borrows from *stochastic* join-optimization techniques, which we discuss next.

2.7.4 Stochastic Techniques

Stochastic techniques for join-order optimization operate on the principle that if one constructs a large number of distinct plans for a join query, then just by blind luck, the best of these plans is likely to be of high quality.

Most stochastic techniques incorporate a strategy of "improvement." Given an arbitrary plan, they make small exploratory changes to it; changes that turn out to make the plan better are considered desirable, and are generally retained, while changes that make the plan worse are generally not retained. (However, these policies are not absolute; in some cases, a change that makes the plan worse will be retained in the expectation—or at least the hope—that it will lead to a subsequent improvement.) In this manner an optimizer can gradually evolve a mediocre plan into a much better one.

Stochastic optimization of join orders was first investigated by Ioannidis and Wong [27], and many variations on the theme have been proposed since. Stochastic techniques cannot guarantee optimality, but often they can generate high-quality plans for moderate effort, and with few restrictions. For joins of very large numbers of relations, the stochastic techniques may provide the best hope of offering a reasonable balance between plan quality and affordability of optimization. We shall discuss several stochastic techniques in more

detail in Chapter 8, in the context of presenting our own stochastic technique.

2.7.5 Hybrids and Frameworks

Not all join-optimization techniques fit neatly into one of the classifications listed above. Swami [56] investigated a variety of hybrids built from combinations of optimization techniques; Swami's hybrids all incorporated a stochastic component. The technique of Swami and Iyer mentioned in Section 2.7.3 above, though primarily based on sequencing, might also be considered a hybrid on the basis of its use of stochastic perturbation to improve its result.

Much of the more recent effort in query optimization research has focused on reducing the difficulty of constructing, extending, and modifying query optimizers. Production optimizers require ongoing maintenance because the other components of query-processing systems tend to be enhanced over time. New features in SQL sometimes require optimizer support if these features are to deliver adequate performance; improvements to a system's execution engine, such as addition of new algorithms or support for parallelism, likewise call for optimizer support.

In principle, rule-based optimizers ought to be easy to build and maintain, but in practice they often are not. To construct a rule-based optimizer using a tool such as Volcano, one must supply not only the transformation rules, but also support functions written in a language such as C. Typically these support functions run to many thousands of lines of code, sometimes with the result that the custom-built portion of the optimizer outweighs the portion supplied by the optimizer-generator tool. Maintenance of optimizers that include such large amounts of custom code can be a burden. At the same time, optimizer generators in the style of Volcano lack flexibility inasmuch as they impose a fixed, transformational search strategy.

Several newer optimizer frameworks seek to overcome some of the limitations of optimizer generators in the style of Volcano. Cascades [18], OPT++ [28], and EROC [41] are three such frameworks that have several characteristics in common, as well as many individual differences. Among the characteristics they share are the following:

- All are constructed out of C++ classes. This application of object-oriented software

engineering appears to improve modularity and flexibility of optimizers.

- All support class hierarchies for representing arbitrary query expressions. In Volcano, join and selection predicates were opaque to the built-in mechanisms, and had to be handled entirely through custom support functions. But in Cascades, OPT++, and EROC, such predicates are represented as subexpressions that can be manipulated in the same ways as top-level query expressions.

- All are designed with a view to providing power and flexibility in their search mechanisms.

The kinds of power and flexibility these frameworks provide varies from one framework to the next.

Cascades emphasizes greater *control* over a search strategy that is essentially the transformative strategy of Volcano. By applying heuristics to the order in which different portions of the search space are explored, an optimizer built with Cascades can reduce its exploration effort. Moreover, unlike Volcano, Cascades can interleave the application of transformation rules on *logical* expressions with the construction of *physical* plans. In doing so, Cascades opens up the possibility of pruning away some of the logical transformations that can be applied to a query—a possibility that Volcano did not offer. Shapiro et al. [51] present a modified version of Cascades called *Columbia* that uses a variety of pruning techniques to improve on the performance of previous rule-based optimizers—but does so without sacrificing the optimality of the generated plans.

The design of OPT++ emphasizes an optimizer's *adaptability* to changing requirements. OPT++ can be configured to carry out its search in the manner of Starburst *or* in the manner of Volcano; relatively few changes are required to switch between these two search paradigms, despite their rather different character. OPT++ also easily supports other search strategies, such as stochastic search strategies; and regardless of the search strategy, OPT++ can be made to restrict the search to left-deep plans, or to plans without Cartesian products. In addition, the modular design of OPT++ facilitates support of new logical or physical operators as a query-processing system evolves.

EROC takes a pragmatic, eclectic approach to supporting optimizer construction. The

EROC class library may be thought of as a *toolkit* for building optimizers from ready-made components. The toolkit includes classes for general-purpose data structures that tend to arise in optimization, as well as more specialized classes that deal with searching and other domain-specific operations. McKenna et al. describe how this toolkit was used to build an unconventional, hybrid optimizer that combines Starburst-style enumeration of join orders with Volcano-style plan generation and pruning. The hybrid was easy to construct and performs well in empirical tests.

As previously noted, the primary goals of these frameworks are to provide flexibility and adaptability. The ability to optimize joins of very large numbers of relations does not appear to have been a design goal in any of the frameworks. However, these frameworks *have* been designed so as to provide competitive performance in the optimization of joins of *moderate* numbers of relations. For example, Kabra and DeWitt [28] present join-optimization timings that show that when emulating Volcano's search strategy, OPT++ performs nearly identically to Volcano itself. No such direct comparisons are available for Cascades and EROC, but both have performed well when applied to the TPC/D benchmark queries [4, 41].

2.7.6 Summary

In this section we have discussed the approaches to join-order optimization that appear in the literature: dynamic programming, rule-based optimization, heuristic and sequencing techniques, stochastic techniques, and hybrids. We mentioned representative systems and algorithms that exemplify each of these approaches, and we also discussed frameworks that assist in the development of optimizers whose approach to join-order optimization can be adapted to changing needs.

Dynamic programming remains the predominant approach used by commercial optimizers, though some database vendors prefer to use rule-based optimization. When a system encounters queries too complex to optimize exhaustively, heuristic techniques are the preferred fallback. To the author's knowledge, stochastic techniques are not yet used in commercial optimizers, perhaps because they entail a greater optimization effort than simple heuristic techniques, without a guaranteed payback.

2.8 Summary and Discussion

In this chapter we have introduced some of the vocabulary, concepts, conventions, and stumbling blocks associated with join-order optimization. We reviewed selected fundamentals of relational databases and the relational algebra, and presented a context for query optimization by describing the phases of query processing. We introduced the notion of predicate selectivity, and showed how it is used in the estimation of intermediate-result cardinalities. We described the representation of a query as a join graph, and noted that conventional join graphs preclude predicates involving more than two relations. We presented a *generic cost model* that can be parameterized to predict join-processing costs under a variety of assumptions; and we discussed the effect of physical properties on join-processing costs, observing that the *principle of optimality* is lost in the presence of physical properties. Finally, we discussed the approaches to join-order optimization that have appeared in the literature, of which a subset are in use in commercial optimizers as well.

We have seen that some of the trickiest aspects of join-order optimization are related to predicates. The conditional nature of join associativity, for example, comes as a consequence of join predicates. The problems with the notion of selectivity, and more generally, the difficulty of cardinality estimation, also arise because of predicates. Join graphs have utility only because they shed light on predicate relationships; in the absence of predicates, they could be dispensed with.

These observations lead to the following idea for decomposing the problem of join-order optimization: First solve the problem in the absence of predicates, and then try to add predicates back to the solution to the simpler problem. In the following chapters we explore such a decomposition of join-order optimization, and so obtain the results outlined in the introduction.

Chapter 3

Cartesian Product Optimization

In this chapter, we address the question of how to optimize the computation of a Cartesian product. We begin by presenting the solution steps for a sample optimization problem, and then give an algorithmic generalization of this solution. We go on to analyze the algorithm's complexity, and to discuss the ramifications of our analysis.

Although optimization of Cartesian product computations is of little practical interest in itself, we will see later that our Cartesian product optimizer can serve as the backbone of a join optimizer. By deferring such distractions as predicates until later we may concentrate all our attention on the essential structure of our algorithm.

3.1 Preliminaries

Suppose we wish to find the optimal expression for computing the Cartesian product $A \times B \times C \times D$. (We assume that only a dyadic $\times$ operator is available.) Before proceeding, we need a cost model and some information about A, B, C, and D (e.g., their cardinalities).

Let us say that A, B, C, and D have cardinalities 10, 20, 30, and 40, respectively, and let us assume for the present an extremely straightforward cost model—again deferring consideration of more complicated alternatives until later. For now we define the cost of evaluating a Cartesian product operator to be the cardinality of the result, in keeping with the cost function κ_0 defined in equation (2.61). Thus, since A has cardinality 10 and B has cardinality 20, the cost of the operation $A \times B$ is $10 \cdot 20 = 200$.

To determine the cost of evaluating a *compound* expression such as $(A \times B) \times C$, we shall sum together the cost of each $\times$ operator in the expression, in accordance with

equations (2.59) and (2.60). In the present situation, those cost equations simplify to

$$cost(R) = 0 \tag{3.1}$$

$$cost(E_0 \times E_1) = cost(E_0) + cost(E_1) + |E_0 \times E_1|. \tag{3.2}$$

For example, with $|A| = 10$, $|B| = 20$, and $|C| = 30$, we have $cost((A \times B) \times C) = cost(A \times B) + cost(C) + |A \times B \times C| = 200 + 0 + 10 \cdot 20 \cdot 30 = 200 + 6000 = 6200$.

Note that the cost model given by κ_0 is *symmetric* in that $cost(E_0 \times E_1) \equiv cost(E_1 \times E_0)$. In general a cost model need not be symmetric, and in most of what we do below we will make no assumption of symmetry. (We will point out the situations in which symmetry makes a difference.)

3.2 Solution using Dynamic Programming

Table 3.1 illustrates how dynamic programming can be exploited to find the cheapest way of computing our four-way Cartesian product. The idea is to build a table that records the best strategy for computing each possible *subproduct* of the four-way product in question. Each entry (i.e., each row) of the table corresponds to one subproduct.

The table is constructed in such a way that we can extract the optimal expression for the four-way product as follows. First, we consult the table's final entry, Entry 15, which says that the "Best Split" for $\{A, B, C, D\}$ is $\{A, D\}, \{B, C\}$; that is, the product of $\{A, B, C, D\}$ is best computed as $E_0 \times E_1$, where E_0 computes the product of $\{A, D\}$ and E_1 computes the product of $\{B, C\}$.

Next observe that we wish E_0 to be *optimal* for computing the product of $\{A, D\}$; such an optimum is furnished by Entry 7 in the table. An optimal E_1 is furnished by Entry 8. By recursively consulting the table in this manner we can derive an optimal expression for the four-way product—in this case, $(A \times D) \times (B \times C)$.

A bit of notation will prove convenient. Let $\mathcal{S}^\star$ denote the optimal expression for computing the product of $\mathcal{S}$. Then Entry 15 of our table may be read as saying, in part, that $\{A, B, C, D\}^\star \equiv (\{A, D\}^\star) \times (\{B, C\}^\star)$; similarly for the other entries. Singletons are special: the optimal expression for any individual relation R is evidently just R; thus, $\{R\}^\star \equiv R$.

Entry	Relation Set	Cardinality	Best Split	Cost
1	$\{A\}$	10		0
2	$\{B\}$	20		0
3	$\{C\}$	30		0
4	$\{D\}$	40		0
5	$\{A, B\}$	200	$\{A\}, \{B\}$	200
6	$\{A, C\}$	300	$\{A\}, \{C\}$	300
7	$\{A, D\}$	400	$\{A\}, \{D\}$	400
8	$\{B, C\}$	600	$\{B\}, \{C\}$	600
9	$\{B, D\}$	800	$\{B\}, \{D\}$	800
10	$\{C, D\}$	1200	$\{C\}, \{D\}$	1200
11	$\{A, B, C\}$	6000	$\{A, B\}, \{C\}$	6200
12	$\{A, B, D\}$	8000	$\{A, B\}, \{D\}$	8200
13	$\{A, C, D\}$	12000	$\{A, C\}, \{D\}$	12300
14	$\{B, C, D\}$	24000	$\{B, C\}, \{D\}$	24600
15	$\{A, B, C, D\}$	240000	$\{A, D\}, \{B, C\}$	241000

Table 3.1: Dynamic programming table

Now that we have seen how the table will be used, let us consider how its entries are constructed.

3.2.1 Initialization and Two-way Products

In any application of dynamic programming, the dynamic programming table is filled in entry by entry until the table is completely filled. Entries for "smaller" subproblems are always filled in before entries for "larger" subproblems; in this way, the solutions for the smaller subproblems can be used to assist in solving the larger subproblems.

The first four entries of our sample table describe degenerate "products" involving just a single relation in each instance. Since the "product" of a single relation R is just R itself, there is no Best Split to record for the singleton sets of relations. Thus, the role of Entries 1–4 is to record the cardinalities of the relations A, B, C, D—these are assumed given—together with cost values of 0 reflecting cost equation (3.1).

Entries 5–10 record information about two-way products. Consider Entry 5 as an example. Here the Cardinality field refers to the cardinality of $A \times B$—hence the value $10 \cdot 20 = 200$. The Best Split field gives an encoding of the best (i.e., cheapest) way

to compute the product of A and B. We know that there are just two expressions that compute this product—$A \times B$ and $B \times A$—and because our cost model is symmetric, neither expression is better than the other. But in the general case that will not be true, and so we must distinguish between the two. In the present instance we arbitrarily designate $A \times B$ as the best choice. We encode this choice as the pair of sets $\{A\}$,$\{B\}$, representing, respectively, the relations that appear to the left of the $\times$ operator, and those that appear to the right. Finally, the Cost field gives the cost of the expression designated by the Best Split field; in this case the cost simply equals the cardinality, since both the left- and right-hand inputs of $A \times B$ are just relation names and not complex subexpressions.

3.2.2 Three-way Products

The benefits of dynamic programming begin to be felt, if only in a small way, when we come to the three-way products. Again, let us consider a particular table entry, Entry 11, as an example. Here the objective is to find the best way to compute the product of A, B, and C. There are actually twelve possible alternatives, but we need not examine each of these alternatives individually. Instead, we reason as follows.

Any expression that computes our three-way product must have the form $E_0 \times E_1$ for some E_0 and E_1. Our strategy will be to ignore the substructure of E_0 and E_1, and to think only about which relations participate in each of these subexpressions. Observe that each of A, B, and C must appear exactly once in the whole expression $E_0 \times E_1$. Therefore, the set of relations appearing in E_0 must be some nonempty, proper subset of $\{A, B, C\}$, and the relations in E_1 must be the complement (with respect to $\{A, B, C\}$) of those in E_0.

For example, suppose that the set of relations appearing in E_0 is $\{A, C\}$—in other words, suppose that E_0 computes the product of A and C. Then the set of relations appearing in E_1 is constrained to be exactly $\{B\}$. Now the cost of $E_0 \times E_1$ is $cost(E_0) + cost(E_1) + 10 \cdot 30 \cdot 20$ by equation (3.2). The cost of E_0, in turn, *cannot be less* than the cost of the best expression for the product of $\{A, C\}$, which was computed in Entry 6 to be 300. The cost of E_1 is zero, since E_1 is just B, which involves no computation. It follows

that with E_0 computing the product of $\{A, C\}$ and with E_1 computing the product of $\{B\}$, the cost of $E_0 \times E_1$ is *at best* $300 + 0 + 10 \cdot 30 \cdot 20 = 300 + 6000 = 6300$.

Now recall that we defined the shorthand $\{A, C\}^*$ to mean the *best* expression for the product of $\{A, C\}$, whatever that expression might be; recall also that $\{B\}^*$ should be understood as a synonym for B (the idea being that B is the only expression for the product of the singleton $\{B\}$, and hence the *best* such expression). Thus, we may read Entry 6 of Table 3.1 as saying that $cost(\{A, C\}^*) = 300$, and Entry 2 as saying that $cost(\{B\}^*) = 0$; and we may now restate the conclusion of the preceding paragraph in the concise form of an equation: $cost(\{A, C\}^* \times \{B\}^*) = 300 + 0 + 10 \cdot 30 \cdot 20 = 6300$.

As it happens, $\{A, C\}^* \times \{B\}^*$ is not the best expression for the product of $\{A, B, C\}$; a superior alternative is found in $\{A, B\}^* \times \{C\}^*$. The cost of the latter is given by $cost(\{A, B\}^*) + cost(\{C\}^*) + 10 \cdot 20 \cdot 30$, which works out to $200 + 0 + 6000 = 6200$, as can be seen by consulting Entries 5 and 3 of the table. If we were to consider the other four possible splits of $\{A, B, C\}$ into pairs of nonempty subsets, we would find that none does better than the pair $\{A, B\}, \{C\}$. Accordingly, Entry 11 of the table shows this pair as the Best Split for $\{A, B, C\}$. The corresponding cost of 6200 is entered alongside.

Note that although the table nowhere explicitly records the *best expression* for the product of $\{A, B, C\}$, that expression can be inferred from the Best Split field for $\{A, B, C\}$ together with the Best Split field for $\{A, B\}$.

3.2.3 Final Result

The handling of the four-way product in Entry 15 is conceptually identical to that of the three-way products. The only difference is that the four-way product involves more work—in the general asymmetric case, a total of fourteen splits of $\{A, B, C, D\}$ must be examined to determine which one is best. On the other hand, a naive exhaustive search without dynamic programming would entail the examination of all the 120 different expressions that compute the four-way product. Moreover, determining the cost of each of those 120 compound expressions would involve more work than determining the cost associated with a given split. These facts, taken together, justify going to the trouble of constructing a table such as Table 3.1 when seeking the best expression for a four-way product.

The benefits of dynamic programming become glaringly apparent only when one considers products over somewhat larger sets. But this small example at least illustrates the principles involved.

3.3 The *Blitzsplit* Algorithm

Naturally enough, the procedure we used above to fill in Table 3.1 by hand can also be carried out automatically by a computer program. Figures 3.1 and 3.2 give abstract pseudo-code for such a program, which we call the Blitzsplit algorithm. This pseudo-code is abstract in the sense that it uses features that are not generally supported in computer languages, such as the **choose-such-that** and **for-each-such-that** statements. The use of *sets* as array indexes also departs from the more conventional use of integers in this role. Later we will see how to map these abstract pseudo-operations onto efficient, concrete implementations. But for the present let us focus on higher-level matters.

3.3.1 Declarations

Figure 3.1 contains two declarations. The first of these introduces a new type *rel_data* that describes the information we need to know about the relations whose product is to be optimized. With our simple cost model, we just need to know their cardinalities. Therefore *rel_data* is declared as an array indexed by relation names that associates a cardinality with each such name. This declaration assumes that the type *relation_name*

```
type  rel_data = array indexed by relation_name of
      record
          cardinality : real
      end

var   table : array indexed by set[relation_name] of
      record
          cardinality : real
          best_lhs    : set[relation_name]
          cost        : real
      end
```

Figure 3.1: Declarations for the Blitzsplit algorithm

```
procedure blitzsplit(R : set[relation_name], rel_data : rel_data)
    for each R ∈ R do
        init_singleton(R, rel_data)
    end for
    for m := 2 to |R| do
        for each S ⊆ R such that |S| = m do
            compute_properties(S)
            find_best_split(S)
        end for
    end for
end procedure

procedure init_singleton(R : relation_name, rel_data : rel_data)
    table[{R}].cardinality := rel_data[R].cardinality
    table[{R}].best_lhs := ∅
    table[{R}].cost := 0.0
end procedure

procedure compute_properties(S : set[relation_name])
    choose U such that ∅ ⊊ U ⊊ S
    V := S − U
    table[S].cardinality := table[U].cardinality * table[V].cardinality
end procedure

procedure find_best_split(S : set[relation_name])
    best_cost_so_far := ∞
    for each S_lhs such that ∅ ⊊ S_lhs ⊊ S do
        S_rhs := S − S_lhs
        operand_cost := table[S_lhs].cost + table[S_rhs].cost
        dependent_cost := operand_cost + κ^split(S, S_lhs, S_rhs)
        if dependent_cost < best_cost_so_far then
            best_cost_so_far := dependent_cost
            table[S].best_lhs := S_lhs
        end if
    end for
    table[S].cost := best_cost_so_far + κ^out(S)
end procedure
```

Figure 3.2: The Blitzsplit algorithm

has been defined previously. The precise nature of the type *relation_name* is unimportant, so long as it can be used as an array index.

The second declaration in Figure 3.1 allocates a global variable *table*—a table very much like Table 3.1, with several minor differences:

- Table 3.1 was annotated with Entry numbers, which were convenient to have in Section 3.2 above for the purposes of discussion. But the Blitzsplit algorithm will have no use for these numbers, and so they are omitted from *table*.

- Lacking any notion of Entry numbers, the Blitzsplit algorithm instead accesses table entries on the basis of the Relation Set for which they provide information. Accordingly, *table* is represented as an array *indexed by sets of relation names*; and since these sets of relation names must be known before the corresponding table entries can be accessed, it would be redundant to store these sets inside the table. Hence *table* contains no field that corresponds to the Relation Set column of Table 3.1.

- We are left with the Cardinality, Best Split, and Cost columns of Table 3.1, and these are faithfully reflected in the *cardinality*, *best_lhs*, and *cost* fields of the *table* elements. However, the *best_lhs* field records just the left-hand component of the best split, leaving the right-hand component implicit. Nothing is lost in this way, since the left-hand component fully determines the right-hand component in the context of a given table entry.

Thus, although *table* differs from Table 3.1 in a few details, its information content is essentially the same.

3.3.2 Procedure *blitzsplit*

Let us now turn to the first procedure in Figure 3.2, procedure *blitzsplit*. The arguments to this procedure are a set $\mathcal{R}$ of relation names and an array *rel_data* containing information about the relations named in $\mathcal{R}$. The objective of *blitzsplit* is to find the least costly way of computing the Cartesian product of those relations.

The body of procedure *blitzsplit* consists of two **for**-loops. The first **for**-loop fills in the table entries for the singleton subsets of $\mathcal{R}$; these entries correspond to Entries 1–4

of Table 3.1. The second **for**-loop (which has yet another **for**-loop nested inside of it) successively fills in the table entries for subsets of $\mathcal{R}$ consisting of 2 relation names, then for subsets consisting of 3 relation names, and so on. At the completion of these two loops, *table* has been entirely filled in. Embedded within the completed table lies an encoding of the best expression for the product of $\mathcal{R}$. (Extraction of this expression is discussed in Section 3.3.6 below.)

The real work in filling in the table is done by the subprocedures *init_singleton*, *compute_properties*, and *find_best_split*, discussed below. But the correctness of those subprocedures hinges on a dynamic programming assumption that concerns procedure *blitzsplit*: The subprocedures assume, when filling in the table entry for a given subset $\mathcal{S}$ of $\mathcal{R}$, that the entries for all nonempty, proper subsets of $\mathcal{S}$ have already been completed. We shall refer to this assumption as the *subsets-first assumption.* It is satisfied in the *blitzsplit* procedure of Figure 3.2, because smaller sets of relation names are always dealt with before larger ones.

In Chapter 4 we will see that there are also other good ways to satisfy this assumption. For simplicity, the pseudo-code of Figure 3.2 specifies a deterministic order in which the subsets of $\mathcal{R}$ are to be processed. But in prescribing a particular order, the pseudo-code is actually *overspecifying* the processing of the subsets. Slavish adherence to this prescription is not necessary; what is important is that all nonempty subsets of $\mathcal{R}$ be processed, and that the validity of the subsets-first assumption be upheld.

3.3.3 Procedure *init_singleton*

The role of procedure *init_singleton* is to consult the input information given in *rel_data*, and using that information, to create entries in *table* for singleton sets:

- The *cardinality* field for a given singleton product $\{R\}$ is simply copied from the *cardinality* of R as furnished by *rel_data*.
- The *best_lhs* for $\{R\}$ is assigned the empty set for lack of a better value—the *best_lhs* field actually has no meaning for singleton sets. (Recall that in Table 3.1 the Best Split field was left blank in the entries for singletons.)

- The *cost* of $\{R\}$ is set to 0.0 in accordance with equation (2.59) of our generic cost model.

3.3.4 Procedure *compute_properties*

Table entries for non-singleton sets of relation names are filled in in two steps. First, procedure *compute_properties* computes the *cardinality* field; second, *find_best_split* computes both the *best_lhs* and the *cost* fields. Here we consider *compute_properties*.

In Section 3.2 above we glossed over the computation of the Cardinality values in Table 3.1 because those computations were so straightforward. That the cardinality for $\{A, B, C, D\}$ should be $10 \cdot 20 \cdot 30 \cdot 40 = 240000$, for example, requires no explanation. However, this naive cardinality computation involves 3 multiplications, and more generally, for a set of m relation names, $m - 1$ multiplications. It is preferable to obtain the result cardinality with a single multiplication. Observe that the set $\{A, B, C, D\}$ may be arbitrarily split into two nonempty subsets—for example, $\{A, C\}$ and $\{B, D\}$—and the cardinalities associated with these subsets, 300 and 800 respectively, may be multiplied together to obtain the cardinality 240000 for $\{A, B, C, D\}$.

Procedure *compute_properties* takes advantage of this observation, together with the subsets-first assumption, to compute the *cardinality* field for a given set $\mathcal{S}$. Thus, $\mathcal{S}$ is split arbitrarily into two nonempty subsets $\mathcal{U}$ and $\mathcal{V}$, and the *cardinality* entries for those two subsets are multiplied together. In this way all product cardinalities are obtained without any loop structure inside of procedure *compute_properties*.

3.3.5 Procedure *find_best_split*

Procedure *find_best_split* examines all splits of a given set $\mathcal{S}$ into pairs of nonempty subsets and selects as the best split the pair that yields the lowest total cost. The left-hand component of that pair is recorded in the *best_lhs* field for $\mathcal{S}$, and the corresponding cost is placed in the *cost* field.

That is the gist of *find_best_split*, but it is not the whole story. Because *find_best_split* is called from inside a loop (actually, a nested pair of loops) in procedure *blitzsplit*, the code within *find_best_split*'s own loop is the most speed-critical part of the whole algorithm. It

proves to be worth going to some trouble to pare this code down to its bare essentials.

Thus, to reduce the effort needed to compute costs, we permit the cost function κ to be broken apart into a *split-independent* component κ^{out} and a *split-dependent* component κ^{split}, such that

$$\kappa(R_{out}, R_{lhs}, R_{rhs}) = \kappa^{out}(R_{out}) + \kappa^{split}(R_{out}, R_{lhs}, R_{rhs}). \tag{3.3}$$

(We shall assume that both components are non-negative.) For example, the cost function $\kappa_0(R_{out}, R_{lhs}, R_{rhs}) = |R_{out}|$ can be decomposed into

$$\kappa_0^{out}(R_{out}) = |R_{out}| \tag{3.4}$$

and

$$\kappa_0^{split}(R_{out}, R_{lhs}, R_{rhs}) = 0. \tag{3.5}$$

Using this decomposition, *find_best_split* avoids computing the total cost for the expression associated with each split. Instead it computes a total cost just once, *outside* the loop, after the best split has already been determined. This shortcut has the following straightforward justification.

Let $\mathcal{S}_{lhs}$ and $\mathcal{S}_{rhs}$ denote, respectively, the left- and right-hand sides of some split of $\mathcal{S}$. Then by equation (2.60) the total cost of the expression $\mathcal{S}^\star_{lhs} \times \mathcal{S}^\star_{rhs}$ is

$$cost(\mathcal{S}^\star_{lhs}) + cost(\mathcal{S}^\star_{rhs}) + \kappa(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs}).$$

(Here we let the sets $\mathcal{S}$, $\mathcal{S}_{lhs}$, and $\mathcal{S}_{rhs}$ act as representations for the Cartesian products of the sets they contain.) Using the decomposition of κ into κ^{out} and κ^{split}, this total cost may be rewritten as

$$cost(\mathcal{S}^\star_{lhs}) + cost(\mathcal{S}^\star_{rhs}) + \kappa^{split}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs}) + \kappa^{out}(\mathcal{S}).$$

To minimize this cost over all possible choices of $\mathcal{S}_{lhs}$ and $\mathcal{S}_{rhs}$, it suffices to minimize

$$cost(\mathcal{S}^\star_{lhs}) + cost(\mathcal{S}^\star_{rhs}) + \kappa^{split}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs}),$$

since the term $\kappa^{out}(\mathcal{S})$ is independent of $\mathcal{S}_{lhs}$ and $\mathcal{S}_{rhs}$. Hence, the latter term may be disregarded during the search for the minimum.

The first two terms of $cost(\mathcal{S}^{\star}_{lhs}) + cost(\mathcal{S}^{\star}_{rhs}) + \kappa^{split}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs})$ are very easy to compute. By the subsets-first assumption, the values $cost(\mathcal{S}^{\star}_{lhs})$ and $cost(\mathcal{S}^{\star}_{rhs})$ may simply be fetched from the *cost* fields of the table entries for $\mathcal{S}_{lhs}$ and $\mathcal{S}_{rhs}$, respectively. Consequently, the "operand cost" $cost(\mathcal{S}^{\star}_{lhs}) + cost(\mathcal{S}^{\star}_{rhs})$ can be obtained with a single addition. The difficulty of computing the term $\kappa^{split}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs})$ will depend on the function κ; but for our sample cost function κ_0, this computation is trivial, since $\kappa_0^{split}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs}) = 0$. In this instance, then, the entire partial cost $cost(\mathcal{S}^{\star}_{lhs}) + cost(\mathcal{S}^{\star}_{rhs}) + \kappa^{split}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs})$ can be computed with a single addition.

Procedure *find_best_split* seeks to minimize this partial cost by keeping a running best value for it in the variable *best_cost_so_far* as the loop scans through all possible combinations of $\mathcal{S}_{lhs}$ and $\mathcal{S}_{rhs}$. The running best value is revised downward each time a better partial cost is encountered; when that occurs, the left-hand component of the split that engendered the new minimum is recorded as the *best_lhs* for $\mathcal{S}$. Thus, the *best_lhs* field may be overwritten many times, and might be more aptly named *best_lhs_so_far*. But by the end of the scan, whatever remains in *best_lhs* will in fact be the left-hand side of the absolute best split (or one such in the case of ties); and *best_cost_so_far* will contain the corresponding partial cost. The total cost is then obtained by adding on the term $\kappa^{out}(\mathcal{S})$, and the result is placed in the *cost* field.

The shortcut of computing only partial costs inside the loop in *find_best_split* helps to tighten the loop, but unfortunately the effectiveness of this shortcut is very much dependent on the cost function κ. However, the cost function need not be as trivial as κ_0 to make the shortcut beneficial. Even with more complex cost models, the amount of computation required inside the loop in *find_best_split* can often be kept small on average. Later we shall discuss techniques for mitigating the expense of computing κ^{split} inside the loop.

3.3.6 Extracting the Best Expression

After the *table* array has been filled in, a small amount of extra work is required to extract the expression that represents the optimal computation of the given Cartesian product. Depending on context, one may wish to extract that expression as a tree, or as a string,

```
procedure print_expression(S : set[relation_name])
    if S = {R} for some R then
        print_relation_name(R)
    else
        print "("
        print_expression(table[S].best_lhs)
        print "×"
        print_expression(S − table[S].best_lhs)
        print ")"
    end if
end procedure
```

Figure 3.3: Printing an optimal expression

or in some other form.

As an example of such an extraction, Figure 3.3 gives pseudo-code that *prints* the expression. If procedure *blitzsplit* was used to construct a table for some collection $\mathcal{R}$ of relation names, then *print_expression* may be invoked on any nonempty subset $\mathcal{S}$ of $\mathcal{R}$—including $\mathcal{R}$ itself. The output will be an optimal expression for computing the product of the relations named in $\mathcal{S}$. We assume the existence of a previously defined procedure *print_relation_name* to print the relation names at the leaves of the expression tree.

3.4 Complexity of the Algorithm

We shall now examine the complexity of the Blitzsplit algorithm. We begin with a rough assessment of its space complexity, and then proceed with a somewhat more detailed analysis of its time complexity. We then make several observations that follow from the complexity analysis.

3.4.1 Space Complexity

Let n be the number of relation names in the set $\mathcal{R}$. Then $\mathcal{R}$ has 2^n subsets, and there is an entry in *table* for each of these (except the empty set). The Blitzsplit algorithm uses no other data structure of any significant size. Hence the space complexity of the algorithm is $O(2^n)$. (We will give a more precise estimate when we discuss concrete representations for our data types.)

3.4.2 Time Complexity

For the time complexity we shall give more than just a big-O analysis, since we are interested in the detailed performance characteristics of the algorithm. We assume that sets of relation names are of bounded size, and hence that primitives on them are constant-time operations; we also assume that per-iteration loop overheads are constant. One way to satisfy these assumptions is shown in Chapter 4 below.

Observe first that procedure *blitzsplit* makes n calls to *init_singleton*, and approximately 2^n calls to each of *compute_properties* and *find_best_split* (because there is one call for each nonempty subset of $\mathcal{R}$, and $\mathcal{R}$, being an n-element set, has 2^n subsets). The net contribution of *init_singleton* is plainly insignificant, while the net contribution of the straight-line code in *compute_properties* and *find_best_split* is $2^n T_{subset}$ for some constant T_{subset}.

There is, in addition, a loop in *find_best_split*. Consider the execution of *find_best_split* for some particular argument $\mathcal{S}$, and suppose that $|\mathcal{S}| = m$; i.e., suppose that $\mathcal{S}$ is a set containing m relation names. Then $\mathcal{S}$ has altogether 2^m subsets, hence $2^m - 2$ nonempty, proper subsets. The loop inside *find_best_split* iterates once for each such subset, hence approximately 2^m times for the given argument $\mathcal{S}$.

We have just counted the number of loop iterations in *one* call to *find_best_split*, but now we must count the number of iterations over *all* such calls. Observe the point where *find_best_split* is called, inside the nested pair of loops in procedure *blitzsplit*. Inside this pair of loops, $\mathcal{S}$ is successively bound to different subsets of $\mathcal{R}$, and *find_best_split* is called for each such binding. Indeed, eventually *every* subset of $\mathcal{R}$ (except the singleton subsets and the empty set) will have been bound to $\mathcal{S}$ and passed as an argument to *find_best_split*. Unfortunately, the number of iterations of the loop in *find_best_split* varies dramatically with the number of relations in $\mathcal{S}$. So we must separately consider the 2-relation subsets, the 3-relation subsets, and in general, the m-relation subsets of $\mathcal{R}$.

The number of m-relation subsets of the n-relation set $\mathcal{R}$ is $\binom{n}{m}$. Hence *find_best_split* will be called $\binom{n}{m}$ times with an m-relation argument. As the number of loop iterations for each such call is about 2^m, the aggregate number of loop iterations for all the m-relation

subsets of $\mathcal{R}$, for a given m, is about $\binom{n}{m}2^m$. This formula must be summed over all values of m from 2 to n to take account of the 2-relation subsets of $\mathcal{R}$, the 3-relation subsets, and so on up to the sole n-relation subset of $\mathcal{R}$ (namely $\mathcal{R}$ itself). Therefore the total number of iterations for *all* calls to *find_best_split* is $\sum_{m=2}^{n}\binom{n}{m}2^m$; but it will be convenient to work with the simpler sum $loop\_count = \sum_{m=0}^{n}\binom{n}{m}2^m$, which overestimates the true total by a negligible amount.

To obtain a closed form for *loop_count*, recall the binomial expansion $(x+y)^n = \sum_{m=0}^{n}\binom{n}{m}x^m y^{n-m}$. Taking $x=2$ and $y=1$, we obtain $(2+1)^n = \sum_{m=0}^{n}\binom{n}{m}2^m \cdot 1^{n-m} = \sum_{m=0}^{n}\binom{n}{m}2^m = loop\_count$. That is, $loop\_count = (2+1)^n = 3^n$. Thus, in aggregate, the time contributed by the loop in *find_best_split* is $3^n T_{loop}$ for some constant T_{loop}.

(Another way to see that there are altogether approximately 3^n iterations through the loop in *find_best_split* is this: In any split of a subset $\mathcal{S}$ of n relation names, a given name among the n will appear in the left-hand component of the split, in the right-hand component, or *neither*. So each of the n names falls in one of three conceptual *buckets* labeled "left," "right," and "discard." Conversely, distributing the n names among three such buckets determines a subset of the names (namely, those not "discarded") and a split of that subset. Thus, splits of subsets of n names are in one-to-one correspondence with assignments of those names to three buckets. There are 3^n such assignments, hence 3^n splits of subsets. All these splits will be examined in the course of the Blitzsplit algorithm's execution, except the handful that would leave the "left" or "right" bucket empty.)

To assess the contribution of the conditionally executed code within the loop body, we use a statistical argument. Consider a particular execution of *find_best_split*. On each iteration through the loop, the **if** condition is satisfied only when the split under consideration improves upon the best so far. Assume the splits are examined in random order. Then the probability that the split considered on the ith iteration is better than the first $i-1$ is $1/i$, since any of the first i splits is equally likely to be the best among the i. Hence the expected number of executions of the conditionally executed code is given approximately by the harmonic series $H_{2^m} = \sum_{i=1}^{2^m} 1/i$, where again $m = |\mathcal{S}|$. In

aggregate, the number of executions of this code across all calls to *find_best_split* is

$$cond\_count = \sum_{m=2}^{n} \binom{n}{m} H_{2^m}. \tag{3.6}$$

Once again, to simplify we change the bounds of the summation, giving

$$cond\_count \approx \sum_{m=0}^{n} \binom{n}{m} H_{2^m}. \tag{3.7}$$

Using the fact $H_k \approx \ln k + \gamma$, where $\gamma = 0.57721\ldots$ [29], we obtain

$$cond\_count \approx \sum_{m=0}^{n} \binom{n}{m} (\ln 2^m + \gamma), \tag{3.8}$$

which expands to

$$\sum_{m=0}^{n} \binom{n}{m} m \ln 2 + \sum_{m=0}^{n} \binom{n}{m} \gamma \tag{3.9}$$

or

$$\ln 2 \sum_{m=0}^{n} \binom{n}{m} m + \gamma \sum_{m=0}^{n} \binom{n}{m}. \tag{3.10}$$

Now recall that $\sum_{m=0}^{n} \binom{n}{m} = 2^n$, and that $\sum_{m=0}^{n} \binom{n}{m} m = (n/2)2^n$. (For an explanation and interpretation of the former identity, see the text by Liu [36]; the latter identity can be derived from the former through straightforward manipulation.) Applying these identities to (3.10) finally yields

$$cond\_count \approx (\ln 2/2) n 2^n + \gamma 2^n. \tag{3.11}$$

Let us disregard the $\gamma 2^n$ term; it is relatively small, and in any event, its effect on execution time can be absorbed by the term $2^n T_{subset}$ discussed above. We may thus view the net contribution of the conditionally executed code as $(\ln 2/2) n 2^n T_{cond}$ for some constant T_{cond}.

Then for some T_{loop}, T_{cond}, and T_{subset}, the execution time of the Blitzsplit algorithm is closely approximated by

$$3^n T_{loop} + (\ln 2/2) n 2^n T_{cond} + 2^n T_{subset}. \tag{3.12}$$

3.4.3 A Small Algorithmic Improvement

From the foregoing analysis we can also deduce the number of executions of the cost-function components κ^{out} and κ^{split}. Since the split-independent component κ^{out} appears *outside* the loop in *find_best_split*, we may conclude that κ^{out} executes approximately 2^n times (i.e., the same number of times as *find_best_split* itself). On the other hand, the split-dependent component κ^{split} appears *inside* the loop, and hence, according to our analysis, executes approximately 3^n times.

If κ^{split} should turn out to involve nontrivial computation, its execution count of 3^n could prove to be a bottleneck. To reduce that bottleneck, we may alter the body of the loop in *find_best_split* as illustrated in Figure 3.4. Instead of using a single **if** statement, the revised loop body of Figure 3.4 contains two *nested* **if** statements, and predicates the computation of κ^{split} on the condition $operand\_cost < best\_cost\_so\_far$. Since failure of this condition also implies $dependent\_cost \not< best\_cost\_so\_far$, the net effect of the pair of **if** statements in Figure 3.4 is the same as that of the single **if** statement in Figure 3.2.

However, the number of executions of κ^{split} may be reduced considerably. From our analysis above, we may infer that the statements outside the outer **if** are executed 3^n times, while those inside the inner **if** are executed about $(\ln 2/2)n2^n$ times. Then the execution count of κ^{split} in Figure 3.4 must lie somewhere between those two quantities.

It is difficult to give a more precise *analytical* characterization of the execution count of

```
for each S_lhs such that ∅ ⊊ S_lhs ⊊ S do
    S_rhs := S − S_lhs
    operand_cost := table[S_lhs].cost + table[S_rhs].cost
    if operand_cost < best_cost_so_far then
        dependent_cost := operand_cost + κ^split(S, S_lhs, S_rhs)
        if dependent_cost < best_cost_so_far then
            best_cost_so_far := dependent_cost
            table[S].best_lhs := S_lhs
        end if
    end if
end for
```

Figure 3.4: Making execution of κ^{split} conditional

κ^{split}. But later we shall study this quantity empirically, and we shall see that at least for some cost functions, the execution count of κ^{split} runs closer to $(\ln 2/2)n2^n$ than to 3^n. In such instances, the restructuring of the loop body as illustrated in Figure 3.4 significantly reduces the algorithm's sensitivity to the expense of computing κ^{split}.

3.4.4 Discussion

In Section 2.7 we noted that the exhaustive-search join-optimization algorithms used by Volcano and Starburst have a worst-case time complexity of $O(4^n)$. It is encouraging that the Blitzsplit algorithm's $O(3^n)$ complexity is lower, but this complexity comparison by itself does not tell us a great deal. Constant factors must also be taken into account; in addition, one must bear in mind that in typical cases, the Volcano and Starburst algorithms perform much better than they do in the worst case—and so the $O(4^n)$ complexity is not necessarily very meaningful.

It is therefore interesting to look at actual join-optimization timings for systems that use the Volcano and Starburst algorithms as reported in some of the recent literature. The measurements for Volcano [17] and OPT++ [28] show optimization timings in the range of *seconds* for a join of eight relations. Optimization time for a ten-way join runs to *tens of seconds*, as reported by Kabra and DeWitt [28], and as can be inferred by extrapolation from the performance graph given by Graefe and McKenna [17]. Kabra and DeWitt also graph timings for bushy join optimization allowing Cartesian products; from the standpoint of complexity, these timings *do* reflect the worst case, and run to *hundreds of seconds* for a ten-way join. The EROC timings [41] appear to be consistent with the Volcano and OPT++ timings, though direct comparison is problematical because of differences in the kinds of queries that were tested.

We do not yet know the values of the time constants T_{loop}, T_{cond}, and T_{subset} in formula (3.12) for the execution time of the Blitzsplit algorithm. Even so, by examining the figures in Table 3.2 we can begin to get a feeling for what the algorithm can and cannot do, given an execution time of $3^nT_{loop} + (\ln 2/2)n2^nT_{cond} + 2^nT_{subset}$ for *some* values of T_{loop}, T_{cond}, and T_{subset}. In the table, the variable c_n is used as shorthand for $(\ln 2/2)n2^n$. Thus, successive columns of Table 3.2 give, for a range of values of n, the execution count

n	3^n	$c_n = (\ln 2/2)n2^n$	2^n	$3^n/c_n$	$c_n/2^n$
2	9	2.8	4	3.2	0.7
4	81	22.2	16	3.7	1.4
6	729	133.1	64	5.5	2.1
8	6 561	709.8	256	9.2	2.8
10	59 049	3 548.9	1 024	16.6	3.5
12	531 441	17 034.8	4 096	31.2	4.2
14	4 782 969	79 495.7	16 384	60.2	4.9
16	43 046 721	363 408.7	65 536	118.5	5.5
18	387 420 489	1 635 339.4	262 144	236.9	6.2
20	3 486 784 401	7 268 175.0	1 048 576	479.7	6.9
22	31 381 059 609	31 979 969.9	4 194 304	981.3	7.6

Table 3.2: Quantities relevant to time complexity of Cartesian product optimization

for the unconditional code inside the loop in *find_best_split*, for the conditional code inside the loop, and for the code outside the loop.

The first observation to draw from these numbers concerns what the Blitzsplit algorithm *cannot* do. On today's hardware, this algorithm cannot hope, even under the best of circumstances, to make short work of problems with n equal to 20 or more. If we assume a machine cycle time of 5 nsec, and optimistically suppose that an iteration of the inner loop in *find_best_split* can execute in, say, 4 cycles, then $T_{loop} = 20 \cdot 10^{-9}$ sec. Under these highly optimistic assumptions, the algorithm's execution time with $n = 18$ will already have reached $3^{18} \cdot T_{loop} \approx (0.387 \cdot 10^9) \cdot (20 \cdot 10^{-9}\ \text{sec}) = 7.74$ sec. With $n = 19$, the time will be 3 times that—about 23 seconds—and with $n = 20$, the time will exceed one minute. Clearly, then, for expeditious optimization of 20-way joins, we must look to a new generation of processors or a faster algorithm.

On the other hand, one cannot fail to be struck by the fact that for smaller n, the values of 3^n are not really very large at all. When $n = 8$, 3^n is only a few thousand. To perform competitively with the algorithms mentioned above, the Blitzsplit algorithm would have to optimize an 8-way Cartesian product in a matter of seconds, and hence, the time constants T_{loop}, T_{cond}, and T_{subset} would have to be on the order of a *millisecond.* But it is imaginable that these constants could be made far smaller than that, and we

shall see that indeed they can be.

It is the constant T_{loop}, above all, that we will want to make small, and in seeking to do so, we will have an advantage over transformation-based algorithms such as Volcano's. For in contrast to those algorithms, ours need not concern itself with complex predicate manipulations each time it constructs a new expression (i.e., each time it considers a new pairing of sets of relations). Later, when we address join optimization, and not just Cartesian product optimization, our algorithm will need to manipulate predicates, too. But the structure of the algorithm is such that these predicate manipulations will never need to enter the inner loop where the alternative pairings of sets of relations are examined.

There are other observations to be drawn from Table 3.2. The ratios in the last two columns of the table shed light on the relative contributions of T_{loop}, T_{cond}, and T_{subset} to the algorithm's total execution time. We see that as n rises to the mid-teens, only a hundredth of the inner-loop iterations execute the conditional code inside the loop. Thus, to the extent that the improvement suggested in Section 3.4.3 above succeeds in bringing the κ^{split}-execution count close to $(\ln 2/2)n2^n$, it significantly reduces the criticality of the κ^{split} function.

At the same time, perhaps the most striking feature of the ratios in the table's last two columns is how *small* they are overall. The smallness of these ratios means that while the constant T_{loop} is certainly the most critical of the three, none of them can be ignored: profligate computation in any part of the algorithm will make itself felt.

3.5 Summary

We began this chapter by illustrating the technique of dynamic programming as applied to Cartesian product optimization. We then formalized the technique in abstract pseudo-code for the Blitzsplit algorithm, and analyzed the algorithm's complexity. From that analysis it became apparent that the algorithm ought to perform competitively with the Volcano and Starburst join-optimization algorithms, provided the abstract pseudo-code can be mapped onto an efficient concrete realization. Finding such a realization will be our next concern.

Chapter 4

Lightweight Implementation of Cartesian Product Optimization

The analysis of the previous chapter suggested that despite the exponential complexity of the Blitzsplit algorithm, it ought to have no difficulty in optimizing Cartesian products of moderate numbers of relations. However, to make a more precise assessment of its capabilities, one must consider the algorithm's implementation.

In the present chapter we undertake to translate the abstract code given in the previous chapter into a more concrete form. Our objective will be to devise an implementation that achieves low values for the time constants T_{loop}, T_{cond}, and T_{subset}; we shall also seek to achieve efficiency in space usage.

We shall continue to express the algorithm in a kind of pseudo-code, but this time around the pseudo-code will be more concrete, and will avoid using sets as array indexes, or instructions such as **choose-such-that** and **for-each-such-that**. At the end of the section we will discuss the mapping of this concrete pseudo-code into an even more concrete form: C code. The discussion will conclude with observations about empirical performance measurements taken on C code.

4.1 Representation of Data Types

Let us refer to the relation names in $\mathcal{R}$ as R_0, R_1, ..., R_{n-1}. Then in our concrete pseudo-code we will identify these names by their integer indexes; inside the code, R_0 will be just 0, R_1 just 1, and so on up to $n-1$. What these numerical names lack in mnemonic value they will make up for in programming convenience.

Undoubtedly the implementation detail that most affects the performance of the Blitzsplit algorithm is the representation of *sets* of relation names. *Bit vectors* are a natural representation for sets over finite domains—especially over *small* domains. The domain of interest in the present instance is very small indeed: it consists of the n relation names in $\mathcal{R}$. Since we do not expect n to run as high as 20—and certainly no higher than 32—we may encode a set of relation names as a single 32-bit integer. This representation is not only compact, but also provides for extremely rapid execution of the set manipulations we require in the algorithm.

Specifically, we will assign sets of relations names to integers in the following manner. We will represent the singleton $\{R_0\}$ by the integer 2^0; $\{R_1\}$ by 2^1; and so on, so that in general $\{R_i\}$ becomes 2^i. Representations for larger sets will be obtained by summation; for example, $\{R_0, R_2\}$ becomes $2^0 + 2^2 = 1 + 4 = 5$ (or 101 in binary), and $\{R_0, R_1, R_5\}$ becomes $2^0 + 2^1 + 2^5 = 1 + 2 + 32 = 35$ (or 10011 in binary). This encoding of sets is called the *characteristic vector*.

Note that a given small integer can have two completely different interpretations as a relation name on the one hand, and as a set of relation names on the other hand. The integer 5 representing the relation name R_5 should not be confused with the integer 5 representing the set $\{R_0, R_2\}$. To minimize such confusion, our pseudo-code will use separate type names for integers in these two roles.

The code in Figure 4.1 is adapted from Figure 3.1; the portions that are new or changed are highlighted in shadowboxes. The limit we have placed on the problem size, $max\_n = 18$, reflects the observations of Section 3.4.4. The type *relation_name* has now become a subrange of the integers, as discussed above. The new type *setrep*, also an integer subrange, will now be used where formerly we specified **set**[*relation_name*].

We have been forced to introduce the constant max_n mainly because our new, concrete declaration for *table* reserves space for an array and has to say how much space is needed. In an actual implementation, one might prefer to allocate space for *table* dynamically to avoid waste. Indeed, one of the useful properties of the integer representation of sets is that it is *dense* in the sense that every integer in the range $[0, 2^n - 1]$ is the representation for some subset of $\mathcal{R} = \{R_0, R_1, \ldots, R_{n-1}\}$. Because of this property, a dynamically

```
const max_n = 18
type relation_name = 0..(max_n − 1)
type setrep = 0..(2^max_n − 1)

type rel_data = array indexed by relation_name of
     record
          cardinality : real
     end

var  table : array indexed by [setrep] of
     record
          cardinality : real
          best_lhs    : [setrep]
          cost        : real
     end
```

Figure 4.1: Concrete declarations

allocated table with index range $[0, 2^n - 1]$ wastes no space on table entries for sets that do not exist. (It does waste space by reserving an entry for the empty set of relations, which "exists" but plays no part in the Blitzsplit algorithm. However, the entry for the empty set is the only one that is wasteful in this way.)

Moreover, the compactness of the integer representation for sets has the consequence that each entry of the table need occupy only 20 bytes: 8 bytes for each of the reals *cardinality* and *cost*, and 4 bytes for the bit-vector *best_lhs*.[1] The $O(2^n)$ space complexity estimate given previously may now be refined to $20 \cdot 2^n$ bytes. For $n = 16$, the space requirement comes to about 1.3MB, a relatively modest amount by current standards; for $n = 18$, the requirement is 5.2MB—more substantial, but still not outrageous. (Space usage will rise by a small factor—ordinarily between 1.2 and 4, depending on the cost model and whatnot—when we extend the table entries in later chapters.)

[1] We assume IEEE double-precision floating point except where otherwise noted, not because the problem demands high precision, but because we may encounter a wide range of exponents.

4.2 Set Operations using Integer Arithmetic

We will assume that the available operations on integers include the usual two's complement addition, subtraction, and negation, as well as several bit-oriented operations: bit-wise *and*, here denoted &; bit-wise negation (or one's complement negation), here denoted ~; and binary left-shift of the constant 1, here denoted 2^k, where k is the shift distance. When its operands represent sets, & may be thought of as the set-intersection operator; ~ may be thought of as set complement (with respect to the domain). Note that when we use the operation 2^k to promote a relation name into a singleton set, the exponent k should be thought of as having type *relation_name*, while the result should be thought of as having type *setrep*.

Our code will take advantage of the fact that integer subtraction, as in $x-y$, represents set subtraction when the set represented by y is contained in the set represented by x; and that addition, as in $x+y$, represents set union when x and y represent disjoint sets.

4.3 The Auxiliary Function *least_subset*

At several points we will make use of the function *least_subset* illustrated in Figure 4.2. Given the input $2^7+2^4+2^3$, to take one example, this function returns 2^3; given $2^{15}+2^5$, it returns 2^5. It always returns the low-order bit of its input, so in some sense it is returning the "least" nonempty subset of the set represented by the input. It achieves this result through a well-worn bit-manipulation trick that can be explained as follows.

The two's complement of a binary number is obtained by inverting all bits, then adding one. For example, if we assume 8-bit words, 10011000, when inverted, becomes 01100111, and then adding one yields 01101000. But observe that if the original binary number had exactly k trailing zero-bits (in our example, $k = 3$), these become one-bits when inverted, and then adding one to them has the effect of restoring them to zero-bits through carry

```
function least_subset(s : setrep) : setrep
    return s & -s
end function
```

Figure 4.2: Least-subset function

propagation. The carry finally propagates into the $(k+1)$st-lowest bit, which, having previously been inverted from one to zero, is restored to one. At that point the carry propagation stops, and all higher-order bits remain inverted. In short, an alternative description of the two's complement operation is that it inverts all bits *to the left of the lowermost one-bit.* In our example, the high-order bits 1001 changed to 0110, but the lowermost one-bit and the trailing zero-bits were unaffected. Hence the bit-wise *and* of a binary number with its two's complement erases its high-order bits and preserves just the lowermost one-bit.

4.4 Procedure *blitzsplit*

Armed with *least_subset*, let us now consider the concrete realization of the procedures of the Blitzsplit algorithm, beginning in Figure 4.3 with procedure *blitzsplit*. The first argument to this procedure is now simply an integer specifying the number n of relations over which we wish to compute a Cartesian product. Since we are assuming the relations will be identified by index values in the range $[0, n-1]$, there is no need to pass a set of relation names to *blitzsplit*.

In the first loop of *blitzsplit*, R now is an integer rather than a relation name, and the iteration over set elements in the pseudo-code is realized by simply counting from 0 to $n-1$. The second loop has also turned into a simple counting loop and is no longer a nested pair of loops. Here s is a *setrep*, i.e., an integer corresponding to a set $\mathcal{S}$ in the abstract code; to bind s to the representations of all nonempty subsets of $\{R_0, \ldots, R_{n-1}\}$, we simply step through the integers from 1 to $2^n - 1$. There are two details that must be observed to ensure the soundness of this implementation.

First, the procedures *compute_properties* and *find_best_split* should be called only with arguments representing sets of at least two relation names; but sequentially stepping through the integers from 1 to $2^n - 1$ yields the singleton sets intermixed with all the others. To get around this difficulty, the test "**if** *least_subset*$(s) \neq s \ldots$ " bypasses the singleton sets: a bit pattern is equal to its low-order bit just when there is exactly one bit in the pattern to begin with.

```
procedure blitzsplit(n : integer, rel_data : rel_data)
    for R := 0 to n − 1 do
        init_singleton(R, rel_data)
    end for

    for s := 1 to 2^n − 1 do
        if least_subset(s) ≠ s then
            compute_properties(s)
            find_best_split(s)
        end if
    end for
end procedure
```

Figure 4.3: Concrete *blitzsplit*

Second, recall that *compute_properties* and *find_best_split* rely on the *subsets-first assumption*. That is, to be sure they will work correctly, we must apply them to all subsets of a set $\mathcal{T}$ before applying them to $\mathcal{T}$ itself. But this requirement will indeed be satisfied if we step through the integers sequentially, for the following reason: If $\mathcal{S} \subsetneq \mathcal{T}$, and s and t are their respective integer representations, then $s < t$, since s is just t with some of its bits zeroed out. (Note, though, that the converse does not hold; for example, $2^0 < 2^1$ but $\{R_0\} \not\subseteq \{R_1\}$.) Consequently, s will be encountered before t in sequential iteration.

4.5 Procedures *init_singleton* and *compute_properties*

The realizations of *init_singleton* and *compute_properties* (Figure 4.4) contain no surprises. Aside from the fact that the set variables $\mathcal{S}$, $\mathcal{U}$, and $\mathcal{V}$, which appeared in script in the abstract code, have been replaced with the *setrep* variables s, u, and v, there are just three small changes.

- The empty-set constant has been replaced by its integer representation, 0.
- Instead of $\{R\}$ we now write 2^R to create a singleton set.
- The nondeterministic **choose-such-that** statement in *compute_properties* has been replaced by a deterministic assignment statement (the highlighted statement in Figure 4.4).

procedure *init_singleton*(R : *relation_name*, *rel_data* : *rel_data*)
    table[2^R].*cardinality* := *rel_data*[R].*cardinality*
    table[2^R].*best_lhs* := 0
    table[2^R].*cost* := 0.0
end procedure

procedure *compute_properties*(s : *setrep*)
    u := *least_subset*(s)
    $v := s - u$
    table[s].*cardinality* := *table*[u].*cardinality* * *table*[v].*cardinality*
end procedure

Figure 4.4: Concrete *init_singleton* and *compute_properties*

The use of the highlighted assignment statement as a replacement for the original **choose-such-that** statement is justified as follows. Since the argument s of *compute_properties* is a set of at least two relation names, any singleton subset of it is a nonempty, proper subset. So the *least_subset* in particular, being a singleton, satisfies the conditions of the choice; and since the choice was not otherwise restricted, this choice is as good as any other.

4.6 The Auxiliary Function *next_subset*

In *find_best_split* we face a new problem when we try to make concrete the **for-each-such-that** loop that scans through all possible splits of a given set $\mathcal{S}$. In the abstract code, the loop body executes once for each possible binding of $\mathcal{S}_{lhs}$ to a nonempty, proper subset of $\mathcal{S}$. For a loop in the concrete code to achieve the same effect, its loop variable must be bound successively to each *integer* representing a nonempty, proper subset of $\mathcal{S}$. In other words, given a *setrep* s representing a set $\mathcal{S}$, the concrete code must somehow step through all the integers whose *bits* are a nonempty, proper subset of the *bits* in s.

If the bits in s are contiguous—for example, if $s = 2^4+2^3+2^2$—then stepping through the subsets is easy. In that case one may just start with 2^2, and on each iteration add 2^2 until one reaches $2^4 + 2^3 + 2^2$. But if s has "holes" in it, as in $2^{10} + 2^7 + 2^6 + 2^2$, the stepping cannot be accomplished by iterated addition of the lowermost bit, because

during some of those additions carry bits will spill into the holes and will not propagate as they should. However, this problem has a remarkably straightforward remedy.

4.6.1 Conception

The auxiliary function *next_subset* in Figure 4.5 provides the means for stepping through all the subsets of set s without tripping over the holes in its bit pattern. The conception behind *next_subset* is to build "bridges" over the holes, thereby linking together the islands of one-bits in s. Then during addition, carry bits will propagate from one island to the next by crossing the bridges. Once the addition is completed, the bridges may be removed.

The particulars of this process are illustrated in Figure 4.6. In the first line of the figure, our example set $s = 2^{10} + 2^7 + 2^6 + 2^2$ is represented as a 12-bit binary number; the labels across the top of the figure reflect the positions of the bits. Just below the binary representation of s is a depiction of the *islands* of one-bits that lie within s. In this depiction, the zero-bits appear as blanks so as to create a visual contrast between the islands and the holes. (Inasmuch as *islands* has a numerical value, it is the same as that of s, namely $2^{10} + 2^7 + 2^6 + 2^2$. Throughout the figure, blanks are to be interpreted as having zero value.) The next line depicts *bridges* of one-bits that span the holes between the islands. Numerically, *bridges* is just the one's complement of s.

Subsequent lines in the figure illustrate the process of iterating through the subsets of s. Starting with four zero-bits in the positions that lie within the islands ("start"), we first fill in the holes between these bits with bridges ("build bridges"). Then we add 1 at the low-order bit ("increment"); this addition ripples through bit-values 2^0 and 2^1 and winds up turning on the bit for 2^2. After the bridges have been stripped away ("burn bridges"), we are left with the bit pattern 0,0,0,1 in the island bit positions.

The figure shows three more iterations as the island bits run through the values 0,0,1,0, then 0,0,1,1, and finally 0,1,0,0. In effect we are counting in binary within the island bits,

```
function next_subset(subset : setrep, s : setrep) : setrep
    return s &(subset − s)
end function
```

Figure 4.5: Next-subset function

	11	10	9 8	7 6	5 4 3	2	1 0
s	0	1	0 0	1 1	0 0 0	1	0 0
islands		1		1 1		1	
bridges	1		1 1		1 1 1		1 1
start		0		0 0		0	
build bridges	1	0	1 1	0 0	1 1 1	0	1 1
increment	1	0	1 1	0 0	1 1 1	1	0 0
burn bridges		0		0 0		1	
build bridges	1	0	1 1	0 0	1 1 1	1	1 1
increment	1	0	1 1	0 1	0 0 0	0	0 0
burn bridges		0		0 1		0	
build bridges	1	0	1 1	0 1	1 1 1	0	1 1
increment	1	0	1 1	0 1	1 1 1	1	0 0
burn bridges		0		0 1		1	
build bridges	1	0	1 1	0 1	1 1 1	1	1 1
increment	1	0	1 1	1 0	0 0 0	0	0 0
burn bridges		0		1 0		0	

Figure 4.6: Counting inside of a bit pattern

just as if there were no holes, and the islands were contiguous.

4.6.2 Implementation

We can now explain the expression $s\,\&(subset - s)$ in the definition of *next_subset*.

We are given a value *subset* that lies within the islands defined by s, and we are to find the next binary value that lies within those islands. The first step is to build bridges, which we achieve by adding the binary number *bridges* to *subset*; but since *bridges* is just the one's complement of s, the result of this first step is $subset + {\sim}s$. (Observe that the bits in *subset* and ${\sim}s$ are necessarily disjoint.)

The next step is to add 1, which yields $subset + {\sim}s + 1$. But ${\sim}s + 1$ is just the two's

complement of s, i.e., $-s$, so at the end of the second step we have $subset - s$.

Finally, in the third step, a bitwise *and* with s destroys bridges and preserves only island bits. Thus we have $s\,\&(subset - s)$.

The foregoing explanation notwithstanding, the appearance of subtraction in the expression $s\,\&(subset - s)$ may seem counterintuitive. But it makes perfect sense if one considers particular special cases. One important case is seen when s is the maximal set consisting of all one-bits. Since the pattern of all one-bits represents -1 in two's complement arithmetic, $s\,\&(subset-s) = -1\,\&(subset-(-1)) = -1\,\&(subset+1) = subset+1$. In other words, when the set s is all-inclusive, *next_subset* simply steps sequentially through the integers. The cases where s is the two's complement representation of -2, -4, etc., are similar—*next_subset* then steps by 2 through the even integers, or by 4 through the multiples of four, and so on.

Note also that in the special case where *subset* equals 0, *next_subset* returns $s\,\&(0-s) = s\,\&-s$. Thus, $next\_subset(0, s)$ is the same as $least\_subset(s)$. There is nothing surprising about these facts, but if nothing else they are reassuring, for they show that *next_subset* behaves as it ought in the cases that are easiest to understand.

4.6.3 Generalization

The function *next_subset* presented above counts *upward* in binary inside the island bits of a bit pattern. But counting *downward* inside a bit pattern is also easily achieved by means of the function $prev\_subset(subset, s) = s\,\&(subset - 1)$. The asymmetry between this definition and that for *next_subset* stems from the fact that the zero-bits between the islands in s already form effective bridges for the propagation of *borrows*, and hence do not need to be altered prior to the subtraction of 1.

More generally, the subtrahend 1 in *prev_subset* can be replaced by any value that lies within the island bits of s, and subtraction will continue to work correctly inside the island bits, just as if these bits were contiguous. Thus, if k lies within the island bits of s, one can define $prev\_subset(subset, s) = s\,\&(subset - k)$ to step through the subsets by some stride determined by k. Note that any odd stride (i.e., odd with respect to the island bits) may be used to cycle through all the subsets before any subset is repeated.

Although we use only *next_subset* in the present work, it is conceivable that one could profit by choosing strides other than +1 for stepping through the subsets in alternative orders. In particular, the assumption in Section 3.4.2 that the subsets are visited in a *random* order might come closer to being satisfied (at least in appearance) with some different stride. Use of a large stride with frequent wrap-around can ensure that none of the bits within s remains in the same state for a long time.

4.7 Procedure *find_best_split*

With the function *next_subset* in hand, we obtain a concrete realization of procedure *find_best_split* without difficulty (Figure 4.7). For simplicity, in this realization of the code we assume that the cost function κ is the naive cost function κ_0. In particular, we assume that $\kappa^{split}(\ldots) = 0$, and so *dependent_cost* and *operand_cost* are one and the same.

For the most part this concrete code is a fairly direct transcription of the abstract procedure *find_best_split* in Figure 3.2, but (aside from the omission of *dependent_cost*) there is one detail of the concrete code in which we have cheated a bit. Rather than use *next_subset* to iterate through all the subpatterns of s, as one might expect, we have instead used *next_subset* to iterate through just *half* of those patterns. Suppose s is $2^7 + 2^6 + 2^2$. Then the new variable *high_part* acquires the value $2^7 + 2^6$, i.e., all bits of s save the least bit. It is *high_part*, not s, that we iterate through, so on successive iterations of the loop, *lhs* takes on the values 2^6, then 2^7, and finally $2^7 + 2^6$. The effect of leaving the bit 2^2 out of the iteration is that we consider only those splits that place R_2 on the *right*-hand side of the Cartesian product expression. Symmetry assures that if there is a least-cost plan with R_2 on the left, then there is a plan of equally low cost with R_2 on the right—because the right- and left-hand sides of the expression are interchangeable. So we do not lose anything by consigning some particular relation (R_2 in our example) to the right-hand side. We do however gain approximately a factor of two in speed.

That gain *appears* to rely on the symmetry of our cost model. But even when the cost model is not symmetric, it is beneficial to structure the iteration as we have done in Figure 4.7. Imagine a cost model such that κ, and hence κ^{split} too, is asymmetric with

```
procedure find_best_split(s : setrep)
    best_cost_so_far := ∞
    high_part := s − least_subset(s)
    lhs := 0
    while lhs < high_part do
        lhs := next_subset(lhs, high_part)
        rhs := s − lhs
        operand_cost := table[lhs].cost + table[rhs].cost
        if operand_cost < best_cost_so_far then
            best_cost_so_far := operand_cost
            table[s].best_lhs := lhs
        end if
    end while
    table[s].cost := best_cost_so_far + table[s].cardinality
end procedure
```

Figure 4.7: Concrete *find_best_split*

```
if operand_cost < best_cost_so_far then
    dependent_cost := operand_cost + κ^split(s, lhs, rhs)
    if dependent_cost < best_cost_so_far then
        best_cost_so_far := dependent_cost
        table[s].best_lhs := lhs
    end if
    dependent_cost := operand_cost + κ^split(s, rhs, lhs)
    if dependent_cost < best_cost_so_far then
        best_cost_so_far := dependent_cost
        table[s].best_lhs := rhs
    end if
end if
```

Figure 4.8: Use of an asymmetric cost function

respect to the left-hand and right-hand inputs. Then the **if** statement inside the **while** loop of Figure 4.7 must be replaced by the more complicated **if** statement of Figure 4.8. By computing *dependent_cost* twice, with the roles of *lhs* and *rhs* reversed the second time, Figure 4.8 compensates for the fact that *lhs* takes on only half of the candidate patterns: the other half are all taken on by *rhs*. The functionality is therefore the same as if *lhs* iterated through all the subpatterns of s, but the loop overhead is only half as great.

In fact, the savings are not confined to loop overhead. As noted in our discussion of

complexity, the frequency with which the test *operand_cost* < *best_cost_so_far* evaluates to *true* tends to be low, and so the entire body of the **if** block of Figure 4.8 is often skipped over. Because of this effect, asymmetric cost models need not entail significantly greater optimization effort than symmetric ones.

4.8 Implementation of Concrete Code in C

Although we have translated the abstract code of Figure 3.2 into a more concrete form in the figures of this chapter, the result is still pseudo-code. To execute this code and measure its performance, one must take the further step of translating the concrete pseudo-code into an actual programming language such as C. Appendix B (page 281) shows the result of such a translation, in which numerous small improvements have been made along the way. The most significant differences between our concrete pseudo-code and the C implementation are the following:

- The table *table*, which in the pseudo-code has three fields per entry, has been decomposed vertically in the C code. That is, it has been made into three separate arrays, named `t_cardinality`, `t_best_lhs`, and `t_cost`, each with a scalar element type. This decomposition appears to improve processor-cache performance.

- The C code has two versions of *find_best_split*, and so to scan a given subset of relations, one of the two versions is chosen on the basis of the number of relations in the subset.

  - The function `find_best_split2`, for sets of three or more relations, begins with three lines of code (labeled with the comment "heuristic to reduce initial best_cost_so_far") that could just as well be omitted as far as functionality is concerned. The idea behind the heuristic is to examine, before any others, two splits of s in which the relations of the left-hand or right-hand side are known to "go well together"—in the sense that these sets of relations participated in an optimal split for some subset of s. One thereby hopes to lower the frequency with which the condition *operand_cost* < *best_cost_so_far* is satisfied.

(The author now believes that this heuristic has little or no value in join-order optimization.)

- In addition, `find_best_split2` unrolls its loop four times. A further economy is made possible by the fact that successive values produced by *next_subset* fall into a cyclic pattern. By precomputing the pattern that *next_subset* would produce on four successive calls, one cuts the number of *next_subset* computations by a factor of four.
- Both of the C versions of *find_best_split* avoid computation of the floating-point sum $table[lhs].cost + table[rhs].cost$ if it can be ascertained that the sum is not needed.

- In representing real numbers, C distinguishes between 4-byte "floats" and 8-byte "doubles." In our application, we may encounter extremely large exponents that can be accommodated only by doubles;[2] floats are inadequate in general. However, we reason that any Cartesian product with cost exceeding 10^{35} is of no practical interest. On this basis we use floats for cost values.

  As a consequence, if the least-cost solution to a given Cartesian product optimization problem does exceed 10^{35} (`COST_LIMIT` in the code), our C implementation of *blitzsplit* will return empty-handed. But if there exists *any* solution with a cost of 10^{35} or less, we are still assured of finding it, and indeed of finding the cheapest such solution.

- To achieve further savings in floating-point manipulations, we sometimes compare floats using integer operations. That is, we take the raw 32-bit encodings of floats, and pretend that these 32-bit values are integers. It is a property of many floating-point representations, including those of the IEEE standard, that if only non-negative, finite values are considered, smaller integers encode smaller floating-point

[2]Most newer machines conform to the IEEE floating-point standard, in which the range for double-precision values exceeds 600 orders of magnitude. However, some older architectures, such as the VAX, use exponent fields of the same width for doubles as for floats. On such architectures one needs to simulate wide-range floating-point arithmetic—for example, using (integer, float) pairs—to avoid overflow on join problems involving many large relations and many small selectivities.

values, and larger integers encode larger floating-point values. This property justifies our use of integer < to compare floating costs.

The improvements listed above have proved to be beneficial to the performance of the algorithm in tests run on a Sun SPARCstation 2.[3] Whether these benefits carry over to other platforms has not been investigated; presumably at least some of them do. In all likelihood many other improvements of a similar nature are possible, some platform-dependent, and some not.

4.9 Empirical Observations

Later, in Chapter 6, we will present a detailed analysis of our algorithm's performance on join-optimization problems. But there is merit in taking a preliminary look at performance now. Patterns seen in the measurements here will provide intuition behind decisions to be made when we generalize the algorithm beyond Cartesian products. They will also give us clues about how to attack the much more difficult problem of measuring performance subsequently, when we must worry about the join-graph topology, selectivity values, and the cost model in addition to the variables we will consider here.

4.9.1 Selection of Sample Points

To measure our optimizer's performance, we must select a sample Cartesian product query, or a set of such queries, to use as input. How should we select those samples? Even at this stage we are dealing with a query space that has many degrees of freedom; it is not obvious how to find sample points in that space that will be representative of the remainder of the space. However, we have nothing to lose by surveying a cross section of the query space in the hope of discovering regularities. As a first step let us therefore examine the subspace in which the number of relations n is fixed at 10.

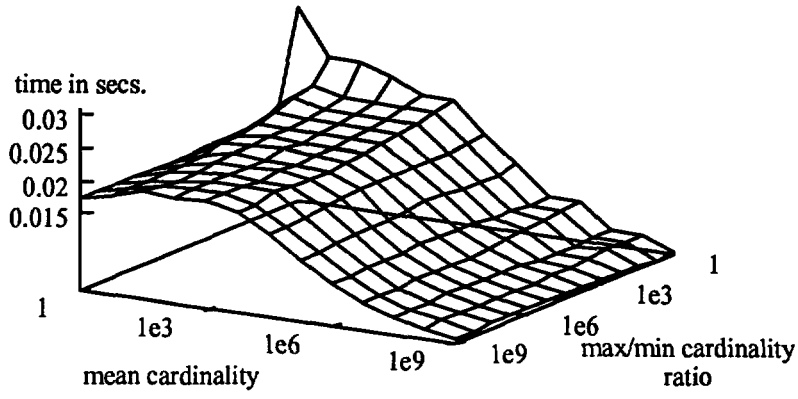
[3]The performance judgments reported here, and the Sun timings reported in this chapter, are based on measurements taken on a lightly loaded Sun 4/75 running under SunOS 4.1.3_U1 Version B at circa 40MHz with a 64KB unified instruction + data cache; the C compiler was gcc version 2.5.8 invoked as gcc -O2.

Thus, we are assuming that the optimization problem involves $n = 10$ cardinalities to be given as input. Each cardinality may vary independently, so we are still faced with the challenge of exploring a 10-dimensional problem space. But common sense suggests that it may not be necessary to vary each cardinality independently; instead it may be more profitable to identify important *features* of the input configurations, and to consider variations in those features. One interesting feature is the *geometric mean* of the 10 cardinalities. A second feature is the *spread* of the cardinalities—the ratio between the maximum and minimum among them. Let us see what we can learn by varying just these two features.

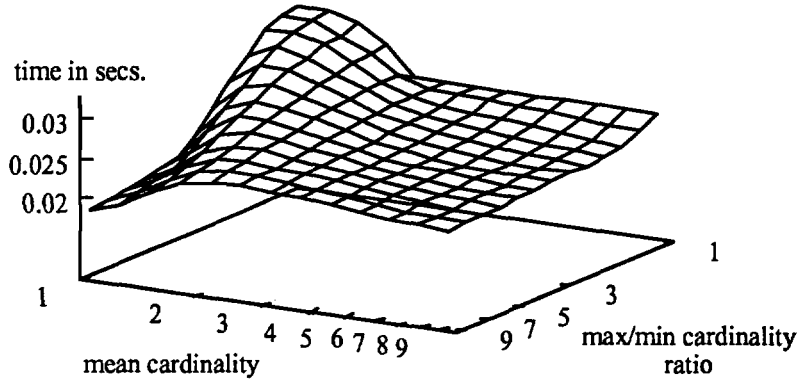
(Note that what we call the *spread* of the cardinalities is closely related to the statistical concept of *variance* [23]. In varying the *geometric mean* and *spread* of the cardinalities, we are effectively varying the first two *moments* of the distribution of the cardinalities' logarithms.)

Since we have just replaced 10 dimensions with merely 2 features, there are still 8 degrees of freedom left over that we must resolve in some manner. We do so as follows. Upon choosing a mean cardinality μ and a spread S, we assign cardinalities to R_0 through R_9 such that R_0 is smallest and R_9 largest, with the cardinalities of the remaining relations spaced evenly between these extremes on a logarithmic scale. Hence $|R_0| \cdot |R_9| = \mu^2$ and $|R_9|/|R_0| = S$, and there is a constant ratio $\rho = S^{1/9}$ such that $|R_{i+1}|/|R_i| = \rho$ for i from 0 to 8. For example, if $\mu = 10^5$ and $S = 10^9$, then $|R_0| = 10^{0.5} \approx 3.162$, while $|R_9| = 10^{9.5} \approx 3.16 \cdot 10^9$, so that $|R_0| \cdot |R_9| = 10^{0.5} \cdot 10^{9.5} = 10^{10} = \mu^2$; moreover $\rho = S^{1/9} = (10^9)^{1/9} = 10$, and hence the R_i form the sequence $3.162, 31.62, 316.2, \ldots, 3.162 \cdot 10^9$. We use this policy to assign cardinalities to the input relations in all the measurements reported in this chapter.

Figure 4.9 shows how optimization time varies with our two selected features. The overall shape of Figure 4.9(a) is attributable to the fact that the total number of iterations of the loop in *find_best_split* is constant *except* when the Cartesian product under consideration entails costs exceeding `COST_LIMIT` (10^{35}). Such costs arise when mean cardinality reaches about $10^{3.5}$, for then the cardinality of the result, a 10-way Cartesian product, becomes $10^{3.5 \cdot 10} = 10^{35}$—hence the total cost of the product cannot be less than 10^{35}.



(a) The big picture



(b) Close-up view of rear corner of big picture

Figure 4.9: Cartesian product optimization time for 10 relations, as a function of mean cardinality and ratio of maximum to minimum cardinality

As mean cardinality rises *above* $10^{3.5}$, one starts to encounter products of *fewer* than 10 relations whose cardinality exceeds 10^{35}, and to avoid overflowing its cost variables, our implementation will refrain from optimizing such products. Consequently, optimization time falls off steeply beyond this point. But for mean cardinalities below this threshold, the picture is basically a large plateau.

At the rear corner of the plateau, however, there is a sharp upward aberration. This aberration is brought into focus in the close-up view of Figure 4.9(b). The shape of this corner surface is attributable to variations in the frequency with which the **if** conditions

are satisfied in the loop in *find_best_split*. When the costs of alternative splits considered in *find_best_split* are spaced far apart, it is often possible to dismiss an uncompetitive split with a cursory glance at its left- or right-hand component alone. More effort is needed to decide whether alternatives are competitive when the costs are close together.

The spacing of costs of alternative splits depends both on the mean cardinality and on the spread of the cardinalities. When mean cardinality is 1 and the spread factor is 1, all intermediate results for all subsets also have cardinality 1; in this situation, there is no variation at all in the costs of alternative splits.[4] If mean cardinality rises but the spread remains at 1, there will be variation due to the fact that products of larger numbers of relations have greater cardinality than products of smaller numbers of relations. On the other hand, if the cardinality remains at 1 but the spread increases, there will be variation simply because different products will be made from different mixes of relations.

As the figure shows, the interplay between mean cardinality and spread is complex, but well-behaved. There is no evidence of discontinuities anywhere in the surface; it seems a good bet that the worst time that appears in the figure is not far from the worst time possible. It is also notable how quickly the optimization time drops off from the corner mound; mean cardinalities as low as 5 yield times that are not far different from the average over the whole plateau.

When mean cardinality is 5 or greater, the spread of the cardinalities no longer seems to make a great deal of difference. From this observation we leap to the conjecture that the 8 degrees of freedom that we threw away earlier probably would not make a great deal of difference either. The measurements in the figure are based on cardinalities equally spaced (logarithmically) between the minimum and the maximum. What would happen if the cardinalities were spaced irregularly? One cannot say for certain without making innumerable measurements, but it seems most implausible that the individual spacings should have a big effect when the aggregate spacing—the ratio between the maximum and minimum—evidently does not.

[4] However, somewhat paradoxically, this case is not the very worst one. With no variation in costs, the *best_lhs* for a given set will be fixed on the first iteration of the loop in *find_best_split*, and will never be bested; the innermost **if** will fail on all subsequent iterations. A harder case is when there is *almost* no variation, but just enough to leave room for updates to *best_lhs*.

The 8 degrees of freedom we discarded determine not only the *spacing* between the cardinalities, but also the *order* of the cardinalities. The measurements shown in the figure are based on R_0 being smallest, R_1 being next smallest, and so on. Might one obtain different measurements with a different ordering? In fact, the ordering of the cardinalities *does* affect performance; the ascending order we have used here is actually not an especially favorable one. At present we do not know how to predict which orderings will do better and which will do worse (though a simple *descending* order often gives noticeably better results than an ascending order). Our ignorance on this point represents an opportunity for further improvement in the implementation—for nothing stands in the way of adding a preprocessing step that rearranges the input cardinalities (in effect by renaming the relations) to obtain a more favorable ordering. But lacking a solid understanding of the issue, for the present we shall stick with a simple ascending order. The performance figures given here should be regarded as *conservative* with respect to order; it is possible to do better.

Our cross-sectional measurements and the conclusions and conjectures we drew from them were based on a fixed number of relations $n = 10$. With different n one obtains numerically different results, but *qualitatively* the effects described here and displayed in Figure 4.9 are entirely typical.

The smooth and generally flat shape of the optimization time function makes it possible to give a fairly informative characterization of the implementation's performance at a given n with just two measurements:

- A "typical" performance figure, meant to be representative of the plateau, and taken at the sample point with mean cardinality 50 and spread 10. These parameters take us a safe distance from the rear corner mound, and at the same time, even with $n = 18$, keep clear of the drop-off that results when costs exceed 10^{35}. (Note that with $n = 18$, a mean cardinality of only 100 falls off the edge of the plateau, since $100^{18} = 10^{2 \cdot 18} = 10^{36}$.)

- A "near-worst" figure, taken at mean cardinality 1.01 and spread 1.01. We know of no sample point that yields higher timing figures.

4.9.2 Timings

Based on the "typical" and "near-worst" sample points discussed above, we now present optimization times for n ranging from 3 to 18. Table 4.1 gives both typical and near-worst times measured on a Sun SPARCstation 2 and on a Hewlett-Packard Series 9000/755.[5] Figure 4.10 gives only the typical times, in the form of a graph.

One will note, referring to the graph, a sudden jump in the Sun timings at $n = 15$ and in the HP timings at $n = 17$. We conjecture that these jumps reflect the points at which the array of costs no longer can be held in processor cache. Because of these jumps, it is unreasonable to try to fit all the Sun measurements or all the HP measurements to a function of the form $3^n T_{loop} + (\ln 2/2) n 2^n T_{cond} + 2^n T_{subset}$, i.e., to the execution times predicted by formula (3.12). However, it is an interesting exercise to attempt such a fit by excluding the points beyond the jump—thus, excluding $n \geq 15$ on the Sun and $n \geq 17$ on the HP. The dashed curves in the graph illustrate such a fit, and show that formula (3.12) indeed fits the measured timings quite well for the points that precede the jumps. From these curve fittings, we infer that T_{loop} is about 180 nsec on the Sun, and about 50 nsec on the HP.

It is notable that if the $(\ln 2/2) n 2^n T_{cond}$ term of formula (3.12) is dropped, it is not possible to obtain curves that fit the measured data anywhere near as well as the curves seen here. Thus, our statistical argument regarding the execution frequency of the conditional code in *find_best_split* appears to be borne out empirically, despite the fact that our implementation of the algorithm visits the subsets in an order that is anything but random. The success of the model may just be a lucky accident, but in any event, it does appear that the execution count of the conditional code is roughly proportional to $n2^n$.

These last remarks apply only to optimization problems whose cardinality configurations lie in the plateau of Figure 4.9(a). When the cardinalities are all very small, the execution frequency of the conditional code will be higher. On the other hand, when the cardinalities are greater than those in the plateau, not only will the conditional code

[5] All Hewlett-Packard timings were measured on a lightly loaded HP 9000/755 running under HP-UX 09.03 at 97MHz with 256KB each of instruction and data cache; the C compiler bundled with HP-UX was invoked as `cc -Aa +03`. The code in Appendix B differs slightly from the version used for these timings.

n	*time in seconds*			
	Sun SPARCstation 2		HP 9000/755	
	typical	*near-worst*	*typical*	*near-worst*
3	0.000 033	0.000 033	0.000 015	0.000 015
4	0.000 089	0.000 090	0.000 041	0.000 041
5	0.000 229	0.000 236	0.000 099	0.000 100
6	0.000 566	0.000 607	0.000 228	0.000 235
7	0.001 37	0.001 58	0.000 519	0.000 556
8	0.003 34	0.004 21	0.001 20	0.001 34
9	0.008 2	0.011 5	0.002 80	0.003 36
10	0.020 8	0.032 3	0.006 77	0.008 67
11	0.054 2	0.092 4	0.016 9	0.023 2
12	0.144	0.269	0.043 5	0.064 0
13	0.397	0.792	0.116	0.181
14	1.15	2.39	0.320	0.521
15	5.36	9.37	0.90	1.52
16	18.8	32.1	2.60	4.49
17	60	104	16.5	23.0
18	187	326	62	87

Table 4.1: Cartesian product optimization time for a given number of relations n

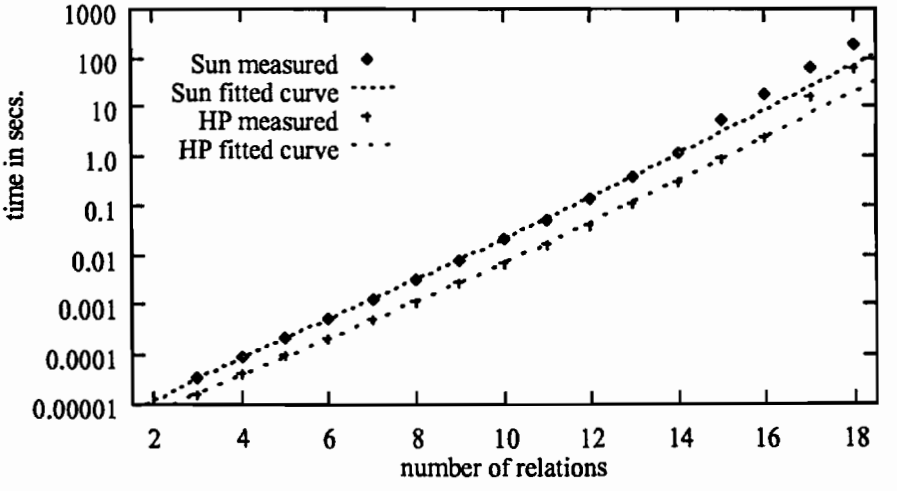


Figure 4.10: Cartesian product optimization time for a given number of relations n

be executed less frequently, but the function *find_best_split* will be skipped altogether for many sets of relations. When join predicates are added to the picture, we will be able to capitalize on the latter effect in ways that are not possible in Cartesian product optimization.

4.10 Summary

In this chapter we have descended from high-level algorithmic matters into a detailed investigation of implementation considerations. We drew on a combination of lightweight data representations and coding tricks in the pursuit of high performance.

Our empirical results validate our approach, at least in the limited context of Cartesian product optimization. They also reveal performance properties of our algorithm that would have been difficult to predict from first principles. Especially interesting is the algorithm's insensitivity to variations in the spread of the input cardinalities except at very low mean cardinalities—and the consequent flatness of the optimization time function over a wide range of possible inputs. That flatness allows us to say with some confidence that the performance trends seen in Table 4.1 and Figure 4.10 are not a fluke of our chosen sample points, but are representative.

Having obtained what we needed from this venture into implementation details, we shall revert in the remaining chapters to a more abstract treatment of the subject, and say no more about bit manipulations, floating-point representations, or cache effects.

Chapter 5

Support for Join Predicates

In the last two chapters we focused on Cartesian product optimization. Because there were no join predicates to contend with, the cardinality of the product over a set $\{A, B, C\}$ was simply $|A| \cdot |B| \cdot |C|$; consequently, computation of cardinalities was a trivial part of the problem. But now, when we take join predicates into account, we have a little more work to do to compute cardinalities.

Suppose the join of $\{A, B, C\}$ can be computed by the expression $(A \bowtie_p B) \bowtie_{q \wedge r} C$, where p, q, and r are predicates. Then the join-result cardinality (henceforth, just *join cardinality*) is $|A| \cdot |B| \cdot |C| \cdot \mathit{selectivity}(p) \cdot \mathit{selectivity}(q) \cdot \mathit{selectivity}(r)$. So if we can identify such an expression for joining $\{A, B, C\}$, it will be straightforward to obtain the cardinality of the result. But given just $\{A, B, C\}$, how can we deduce p, q, and r? Worse, what if there is another expression for the join of $\{A, B, C\}$ that uses *different* predicates—say $(C \bowtie_s A) \bowtie_t B$? Which expression should we take as the basis for our cardinality computation?

As we shall see shortly, the worries lurking behind these questions are unfounded. All sensible expressions that join $\{A, B, C\}$ must involve exactly the same predicates, and these predicates can be deduced without actually constructing *any* of the join expressions in which they participate. In fact, although the computation of cardinalities in the presence of predicates is more complicated than in the Cartesian product case, the extra effort required is surprisingly slight.

In this chapter we present techniques for accommodating join predicates under two sets of assumptions. First, we consider the case where the predicates are *independent*, and second, we develop a mechanism that compensates for *redundant* predicates. But before

getting deeply into either of these topics, let us start off with a few general observations about join graphs and join-result cardinalities.

5.1 Join Graphs, Subgraphs, Predicates, and Cardinalities

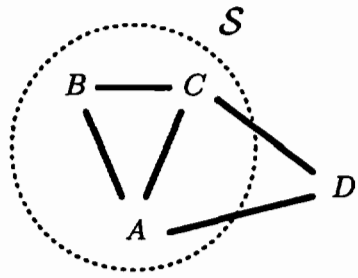
Consider the join graph in Figure 5.1(a). Its nodes are labeled with the relation names A, B, C, and D; we will identify its edges by the names $\widehat{AB}$, $\widehat{AC}$, $\widehat{BC}$, $\widehat{AD}$, and $\widehat{CD}$, and these names will also serve to identify the corresponding predicates. Following graph-theoretic convention, we may characterize the graph as an ordered pair $\mathbf{G} = (\mathcal{R}, \mathcal{P})$, where $\mathcal{R}$ is the node set $\{A, B, C, D\}$, and the edge set $\mathcal{P}$ is the set of predicate names $\{\widehat{AB}, \widehat{AC}, \widehat{BC}, \widehat{AD}, \widehat{CD}\}$.

5.1.1 Induced Subgraphs

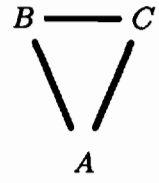
Now suppose we are interested in the cardinality that results from a join over the subset $\mathcal{S} = \{A, B, C\}$. Let $\mathcal{Q}$ be the set of edges *wholly contained* in $\mathcal{S}$ (i.e., those with both endpoints in $\mathcal{S}$)—namely $\{\widehat{AB}, \widehat{AC}, \widehat{BC}\}$. Then the subgraph $(\mathcal{S}, \mathcal{Q})$ of $\mathbf{G}$, illustrated in Figure 5.1(b), is called the subgraph of $\mathbf{G}$ *induced by* $\mathcal{S}$. One can see that in the course of a join of the relations named in $\mathcal{S}$, the predicates that will be applied will be exactly those in the subgraph $(\mathcal{S}, \mathcal{Q})$—no more and no fewer. No more, because predicates not in $\mathcal{Q}$ refer to relations not in $\mathcal{S}$, so these predicates cannot possibly be evaluated when only the relations in $\mathcal{S}$ are available. No fewer, because there is no benefit in deferring the application of a predicate once its referent relations have become available.

It follows that the join cardinality of $\mathcal{S}$ can be computed by multiplying together the cardinalities of all the relations, and the selectivities of all the predicates, that are represented in the induced subgraph $(\mathcal{S}, \mathcal{Q})$ shown in Figure 5.1(b).

A word of caution should be added to the assertion above that there is no benefit in deferring the application of predicates. This assertion rests on the assumption that predicate evaluation is cheap. Other optimizers make the same assumption, and rule out deferral of predicate application in multiway joins. That is, if $(A \bowtie_{p \wedge q} B) \bowtie_r C$ is a valid join expression, most optimizers will not consider the alternatives $(A \bowtie_p B) \bowtie_{r \wedge q} C$



(a) Sets of relation names in a join graph



(b) A subgraph

Figure 5.1: Subsets and subgraphs in a graph

and $(A \bowtie_q B) \bowtie_{r \wedge p} C$. However, those alternatives *could* be preferable to the original expression if p or q involve expensive computations. Hellerstein and Stonebraker [22] describe a cost-based predicate-placement technique that achieves huge gains when expensive predicates are deferred.

We have not investigated the applicability of their technique in the context of the Blitzsplit algorithm. In the present work, we assume that predicates are cheap to evaluate, and that they should therefore be evaluated as early as possible. In other words, they should be *pushed down* as far as possible.

5.1.2 Subgraphs and Join Expressions

The observations of the foregoing paragraphs tell us which predicates should be applied in the course of a join over $\mathcal{S}$, but they do not directly tell us how to construct actual join expressions that apply those predicates. They do however give us the latter information by implication.

Suppose we split $\mathcal{S}$ into two disjoint subsets $\mathcal{U}$ and $\mathcal{V}$, as illustrated in Figure 5.2. Then what was true for $\mathcal{S}$ must also be true for $\mathcal{U}$ and $\mathcal{V}$: The predicates applied in the course of a join over $\mathcal{U}$ will be just those in the subgraph induced by $\mathcal{U}$, and the predicates applied in a join over $\mathcal{V}$ will be those in the subgraph induced by $\mathcal{V}$.

Now let $\mathcal{U}^\star$ denote the best expression for joining $\mathcal{U}$, and $\mathcal{V}^\star$ the best for $\mathcal{V}$ (*cf.* page 71), and consider a join of $\mathcal{U}^\star$ and $\mathcal{V}^\star$. Since all relations in $\mathcal{S}$ participate in this join, all

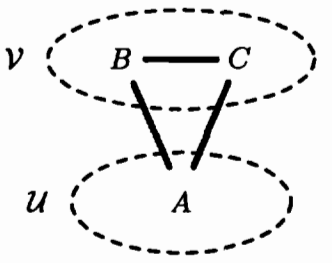


Figure 5.2: Subsets and subgraphs in the graph for $\mathcal{S} = \{A, B, C\}$

predicates wholly contained in $\mathcal{S}$ should also participate. But some of those predicates may also be wholly contained in $\mathcal{U}$ and therefore already participate in $\mathcal{U}^*$, and similarly for $\mathcal{V}$. Then it is the predicates that are left over—those that *span* $\mathcal{U}$ and $\mathcal{V}$—that must qualify the join of $\mathcal{U}^*$ and $\mathcal{V}^*$. In our example, the predicates spanning $\mathcal{U}$ and $\mathcal{V}$ are $\widehat{AB}$ and $\widehat{AC}$, so the correct expression for joining $\mathcal{U}^*$ and $\mathcal{V}^*$ is $\mathcal{U}^* \bowtie_{\widehat{AB} \wedge \widehat{AC}} \mathcal{V}^*$.

Because the predicates qualifying a join are completely determined by the join graph and by the relation names on the left- and right-hand sides, we may omit predicate annotations without ambiguity: $\mathcal{U}^* \bowtie \mathcal{V}^*$, given the join graph $\mathbf{G}$ above, means $\mathcal{U}^* \bowtie_{\widehat{AB} \wedge \widehat{AC}} \mathcal{V}^*$, and cannot mean anything else. In the general case,

$$\begin{aligned} \mathcal{U}^* \bowtie \mathcal{V}^* \quad &\equiv \quad \mathcal{U}^* \bowtie_{conj} \mathcal{V}^* \\ &\text{where} \quad conj = \bigwedge \left\{ p \;\middle|\; p \text{ spans } \mathcal{U} \text{ and } \mathcal{V} \right\} \end{aligned} \tag{5.1}$$

whenever $\mathcal{U}$ and $\mathcal{V}$ are disjoint sets of relation names. The predicates that span $\mathcal{U}$ and $\mathcal{V}$ are precisely those that *belong to* the join of $\mathcal{U}^*$ and $\mathcal{V}^*$ in the sense described previously in Section 2.2.3.

5.1.3 Cardinality Recurrence

Now to compute the join cardinality of $\mathcal{S}$, we may multiply together the join cardinality of $\mathcal{U}$, the join cardinality of $\mathcal{V}$, and the selectivities of all predicates spanning $\mathcal{U}$ and $\mathcal{V}$:

$$\begin{aligned} \textit{cardinality}(\mathcal{S}) \;=\; & \textit{cardinality}(\mathcal{U}) \cdot \textit{cardinality}(\mathcal{V}) \cdot \\ & \prod \Big\{ \textit{selectivity}(p) \;\Big|\; p \text{ spans } \mathcal{U} \text{ and } \mathcal{V} \Big\} \\ \text{where} \quad & \mathcal{U} \cap \mathcal{V} = \emptyset \\ \text{and} \quad & \mathcal{U} \cup \mathcal{V} = \mathcal{S}. \end{aligned} \tag{5.2}$$

The validity of (5.2) follows immediately from the fact that the join of $\mathcal{S}$ can be computed by the join expression of (5.1) (given that $\mathcal{U} \cap \mathcal{V} = \emptyset$ and $\mathcal{U} \cup \mathcal{V} = \mathcal{S}$).

But this explanation may not satisfy completely. Is it not possible that the right-hand side of (5.2) could depend on the choice of $\mathcal{U}$ and $\mathcal{V}$? That no such danger exists, and that the right-hand side of (5.2) gives the correct cardinality regardless of the particular choice of $\mathcal{U}$ and $\mathcal{V}$, can be seen by considering separately the cardinalities and the selectivities that must participate in this product.

The cardinality of each relation in $\mathcal{S}$ must show up in the product exactly once, and indeed it will; for a given relation in $\mathcal{S}$ is either in $\mathcal{U}$, in which case its cardinality appears as a factor in *cardinality*($\mathcal{U}$), or it is in $\mathcal{V}$ and contributes a factor to *cardinality*($\mathcal{V}$). The given relation cannot be in both $\mathcal{U}$ and $\mathcal{V}$, and hence never contributes more than one factor to the product. These observations are valid whenever $\mathcal{U}$ and $\mathcal{V}$ are disjoint and their union is $\mathcal{S}$.

As for the selectivities, recall that the join cardinality of $\mathcal{S}$ must include a selectivity factor for each predicate in the subgraph induced by $\mathcal{S}$. These predicates, as noted above, fall in three camps: those wholly contained in $\mathcal{U}$, those wholly contained in $\mathcal{V}$, and those that span $\mathcal{U}$ and $\mathcal{V}$. Those wholly contained in $\mathcal{U}$ contribute their selectivities as factors in *cardinality*($\mathcal{U}$), and similarly for $\mathcal{V}$; so the predicates that remain are just those that span $\mathcal{U}$ and $\mathcal{V}$. But since we explicitly include the selectivities of these predicates in formula (5.2), these, too, are accounted for. Moreover, these three sets of predicates are disjoint, so no predicate is counted more than once.

5.1.4 Summary

We now have some basic facts about join predicates and join cardinalities at our disposal, and we will proceed to apply them in the context of the Blitzsplit algorithm. All the cardinality computations in the sections that follow are based, directly or indirectly, on equation (5.2).

Refer to equation (5.2) once again. If $\mathcal{U}$ and $\mathcal{V}$ are both nonempty, then by the subsets-first assumption, the values *cardinality*($\mathcal{U}$) and *cardinality*($\mathcal{V}$) will be readily available when it comes time to compute *cardinality*($\mathcal{S}$) in the Blitzsplit algorithm. Consequently the problem of computing cardinalities reduces to the problem of computing $\prod\{\textit{selectivity}(p) \mid p \text{ spans } \mathcal{U} \text{ and } \mathcal{V}\}$; products of this form will command much of our attention as we proceed.

5.2 Cardinality in the Presence of Predicates

In this section we shall present a technique for computing cardinalities in the presence of simple, independent predicates. Our objective will be to enhance the Blitzsplit algorithm so that the cardinality associated with each set $\mathcal{S}$ in the dynamic programming table reflects the appropriate predicates (i.e., the predicates in the join subgraph induced by $\mathcal{S}$). Our technique will apply to arbitrary join graphs (excluding hyperedges).

We will use the observations of Section 5.1 above to incorporate predicate selectivities into our cardinality computations with just three multiplications per entry in the dynamic programming table. (Since we already required one multiplication in the Cartesian product case, the selectivity computations effectively require only *two* multiplications per table entry.) Achieving this efficiency in computing the cardinalities will depend in part on adding another field to each entry in the dynamic programming table. By extending the table in this way, we will be able to further capitalize on the sharing of computation that dynamic programming makes possible.

First we present the conception behind our approach, and then its realization in abstract pseudo-code.

5.2.1 Conception of Cardinality Computation

Our strategy takes advantage of the fact that an order may be imposed on the relation names in the input. Conceptually it does not matter what the order is, as long as it is well-defined and total; but for concreteness let us say that the ordering is $A < B < C < D$ in our example above. This ordering has nothing to do with cardinality or any other property of relations—it is just an arbitrary ordering on the names.

We develop the conception as follows. First we define two operations, *least_subset* and *fan_sels*, that rely on the ordering of the relation names. Then we show how to compute *fan_sels*, and finally we show that the cardinality computation is straightforwardly expressed in terms of *least_subset* and *fan_sels*.

Operation *least_subset* We have already seen an operation with this name in Chapter 4. Here the definition will be slightly different, but as it turns out, equivalent, which justifies our reuse of the name. What we need now is that *least_subset*($\mathcal{S}$) should be the singleton set $\{R\}$ such that $R \leq R'$ for all R' in $\mathcal{S}$. For example, *least_subset*($\{A, C, D\}$) = $\{A\}$, and *least_subset*($\{B, C\}$) = $\{B\}$. Our new definition matches the behavior of the *least_subset* function of Chapter 4 if one takes the ordering on $R_0, \ldots, R_{n-1}$ to be $R_0 < \ldots < R_{n-1}$.

Recall that our goal in Chapter 4 was to split a set $\mathcal{S}$ into two disjoint subsets $\mathcal{U}$ and $\mathcal{V}$; taking $\mathcal{U}$ = *least_subset*($\mathcal{S}$) proved to be a convenient route to that goal. That $\mathcal{U}$ ended up being a singleton, and moreover, a particular, well-defined singleton, was an accident of the implementation, and completely irrelevant from the standpoint of the specification.

Here the situation is entirely different: all details of the behavior of *least_subset* take on logical significance. At this stage *least_subset* should be thought of not as an implementation device, but as an abstract operation in its own right—albeit one with a very precise specification, and one for which we are lucky enough to have an efficient implementation waiting in the wings.

Operation *fan_sels* Before defining *fan_sels*, we must introduce the notion of a *fan* of predicates for a set $\mathcal{S}$. Again consider the set $\mathcal{S}$ of Figure 5.2. Since $\mathcal{U}$ is a singleton, the

edges emanating from $\mathcal{U}$ toward the relation names in $\mathcal{V}$ resemble the spokes of a folding fan. In this example there are just two spokes, $\widehat{AB}$ and $\widehat{AC}$, so it is not much of a fan; but one can well imagine if $\mathcal{V}$ were a larger set such as $\{B, C, E, J, K, L\}$, then there could be as many as six spokes, and the resemblance to a folding fan would emerge more strongly.

There may be many fans embedded in the subgraph induced by a set $\mathcal{S}$, but when we speak of *the* fan of $\mathcal{S}$, we shall mean specifically the fan of predicates reaching from *least_subset*($\mathcal{S}$) to the remaining relations in $\mathcal{S}$. Because A is least in $\{A, B, C, D\}$, the fan we used in our illustration above was in fact the fan of $\mathcal{S} = \{A, B, C\}$.

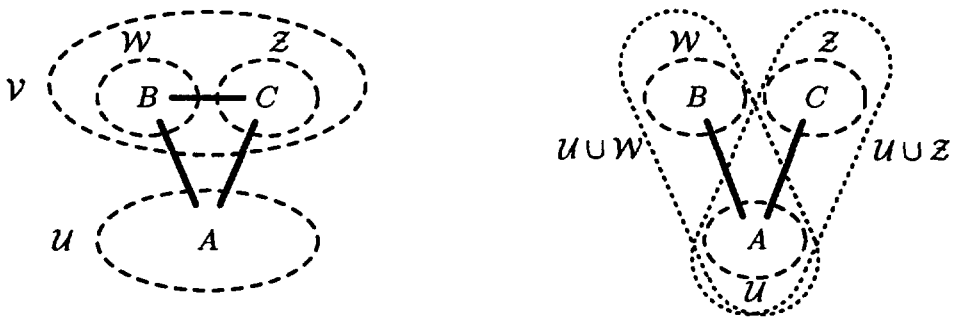
Now to the definition of *fan_sels*. In the example of Figure 5.2, *fan_sels*($\mathcal{S}$) is just the product of the selectivities of $\widehat{AB}$ and $\widehat{AC}$. More generally, for any $\mathcal{S}$, *fan_sels*($\mathcal{S}$) is the product of the selectivities of the predicates in the fan of $\mathcal{S}$:

$$\begin{aligned} \mathit{fan\_sels}(\mathcal{S}) \;&=\; \prod\Big\{ \mathit{selectivity}(p) \;\Big|\; p \text{ spans } \mathcal{U} \text{ and } \mathcal{V} \Big\} \\ &\text{where} \quad \mathcal{U} = \mathit{least\_subset}(\mathcal{S}) \\ &\text{and} \quad \mathcal{V} = \mathcal{S} - \mathcal{U}. \end{aligned} \tag{5.3}$$

One will note the similarity between this product and the $\prod$-expression that appears in equation (5.2). The only difference is that our present characterization of the product is more restrictive: here $\mathcal{U}$ *must* be *least_subset*($\mathcal{S}$). This restriction will prove crucial to computing *fan_sels* easily in the context of the Blitzsplit algorithm.

A recurrence for *fan_sels* A free-standing computation of *fan_sels* would presumably require a loop or recursion to iterate through the selectivities to be multiplied together. However, our use of *fan_sels* will not be free-standing; rather, it will occur in the context of a dynamic programming algorithm, and we will compute *fan_sels*($\mathcal{S}$) for every nonempty subset $\mathcal{S}$ of some set $\mathcal{R}$. If we memoize the result of each such computation in our dynamic programming table, then we have the option of expressing the results of the later computations in terms of the earlier results—relying, as usual, on the subsets-first assumption. By constructing a *recurrence relation* for *fan_sels*, we will be able to avoid looping or recursion in the computation of *fan_sels* for any particular set $\mathcal{S}$.

Suppose as before that $\mathcal{S} = \{A, B, C\}$ has been split into $\mathcal{U} = \{A\}$ and $\mathcal{V} = \{B, C\}$.



(a) A split of $\mathcal{V}$ (b) Subfans of $\mathcal{S} = \mathcal{U} \cup \mathcal{V}$

Figure 5.3: Carving up a fan

Figure 5.3(a) illustrates how $\mathcal{V}$ may be further subdivided into subsets $\mathcal{W}$ and $\mathcal{Z}$. Now consider the sets $\mathcal{U} \cup \mathcal{W}$ and $\mathcal{U} \cup \mathcal{Z}$ shown in Figure 5.3(b). Since A is least in $\mathcal{S}$, A must necessarily also be least in each of the sets $\mathcal{U} \cup \mathcal{W}$ and $\mathcal{U} \cup \mathcal{Z}$. It follows that the predicates spanning $\mathcal{U}$ and $\mathcal{W}$ constitute the fan of $\mathcal{U} \cup \mathcal{W}$, and the predicates spanning $\mathcal{U}$ and $\mathcal{Z}$ constitute the fan of $\mathcal{U} \cup \mathcal{Z}$. Moreover, these fans are disjoint (since $\mathcal{W}$ and $\mathcal{Z}$ are disjoint) and their union is the fan of $\mathcal{S}$ (since $\mathcal{W} \cup \mathcal{Z} = \mathcal{V}$ and $\mathcal{U} \cup \mathcal{V} = \mathcal{S}$). From these facts we deduce the recurrence

$$\begin{aligned} \mathit{fan\_sels}(\mathcal{S}) \;&=\; \mathit{fan\_sels}(\mathcal{U} \cup \mathcal{W}) \cdot \mathit{fan\_sels}(\mathcal{U} \cup \mathcal{Z}) \\ \text{where}\quad & \mathcal{U} = \mathit{least\_subset}(\mathcal{S}) \\ \text{and}\quad & \mathcal{W} \cap \mathcal{Z} = \emptyset \\ \text{and}\quad & \mathcal{W} \cup \mathcal{Z} = \mathcal{S} - \mathcal{U}. \end{aligned} \tag{5.4}$$

In the figure, both $\mathcal{W}$ and $\mathcal{Z}$ are singletons, but in general they need not be; (5.4) holds for any split of $\mathcal{V}$ into disjoint $\mathcal{W}$ and $\mathcal{Z}$. One may visualize the more general case by thinking of the solid lines in Figure 5.3(b) as representing not individual predicates, but *bundles* of predicates connecting $\mathcal{U}$ to $\mathcal{W}$ and $\mathcal{U}$ to $\mathcal{Z}$, respectively. (Nor does the correctness of (5.4) depend on $\mathcal{U}$ being a singleton. The necessity of $\mathcal{U} = \mathit{least\_subset}(\mathcal{S})$ being a singleton will be explained shortly in Section 5.2.2.)

Computing cardinality Combining equations (5.2) and (5.3), we immediately obtain

$$\begin{aligned} \textit{cardinality}(\mathcal{S}) &= \textit{cardinality}(\mathcal{U}) \cdot \textit{cardinality}(\mathcal{V}) \cdot \textit{fan\_sels}(\mathcal{S}) \\ \text{where } & \mathcal{U} = \textit{least\_subset}(\mathcal{S}) \\ \text{and } & \mathcal{V} = \mathcal{S} - \mathcal{U}. \end{aligned} \tag{5.5}$$

Now we are done, for by applying first equation (5.4) and then (5.5), we obtain a cardinality for $\mathcal{S}$ that takes the appropriate selectivities into account; and it does so with just three multiplications, as promised—provided only that we extend our dynamic programming table so that it can store $\textit{fan\_sels}(\mathcal{S})$ with each set of relations $\mathcal{S}$.

5.2.2 Realization of Cardinality Computation

It is straightforward to implement the technique just described as program code. But the foregoing discussion left some loose ends, and these need to be tied up in the code, for computers are famously obstinate about failing to infer one's intent even when it ought to be obvious.

One thing that ought to be obvious is that selectivities cannot be conjured up out of thin air. Yet in our just-concluded discussion of cardinality computations, in which we purportedly took selectivities into account, we never once referred to the predicate selectivities that presumably would be supplied by the caller as input to the join optimizer. Plainly we must have omitted something. Our omission was to give recurrence relations without addressing the *initial conditions*. The recurrence relations themselves were legitimate, but for some sets $\mathcal{S}$ the recurrences shed no light. Of particular interest are the cases of recurrence (5.4) when $\mathcal{S}$ is a singleton, or when $\mathcal{S}$ is a set of exactly *two* relation names.

Consider first the case where $\mathcal{S}$ is a singleton. What is the fan of a singleton? If we split singleton $\mathcal{S}$ into $\mathcal{U} = \textit{least\_subset}(\mathcal{S})$ and $\mathcal{V} = \mathcal{S} - \mathcal{U}$, then $\mathcal{U} = \mathcal{S}$ and $\mathcal{V} = \emptyset$. Then the fan of $\mathcal{S}$ is the set of predicates spanning $\mathcal{S}$ and the empty set, which is just an empty set of predicates. Therefore $\textit{fan\_sels}(\mathcal{S})$ is the empty product, i.e., 1. Note also that if the empty set $\mathcal{V}$ were to be split into $\mathcal{W}$ and $\mathcal{Z}$, then both $\mathcal{W}$ and $\mathcal{Z}$ would have to be the empty set themselves. It would then follow that $\mathcal{U} \cup \mathcal{W} = \mathcal{U} = \mathcal{S}$ and

$\mathcal{U} \cup \mathcal{Z} = \mathcal{U} = \mathcal{S}$. Then what recurrence (5.4) says about *fan_sels*($\mathcal{S}$) for singleton $\mathcal{S}$ is that *fan_sels*($\mathcal{S}$) = *fan_sels*($\mathcal{S}$) · *fan_sels*($\mathcal{S}$), which is consistent with *fan_sels*($\mathcal{S}$) = 1; so we are still respecting (5.4) even in the singleton case.[1]

When $\mathcal{S}$ has exactly *two* elements, then if $\mathcal{U}$ = *least_subset*($\mathcal{S}$) and $\mathcal{V} = \mathcal{S} - \mathcal{U}$, evidently $\mathcal{V}$ must be a singleton. Consequently if $\mathcal{V}$ is split into $\mathcal{W}$ and $\mathcal{Z}$, then either $\mathcal{W} = \mathcal{V}$ and $\mathcal{Z} = \emptyset$, in which case $\mathcal{U} \cup \mathcal{W} = \mathcal{S}$ and $\mathcal{U} \cup \mathcal{Z} = \mathcal{U}$, or just the reverse (i.e., $\mathcal{W} = \emptyset$ and $\mathcal{Z} = \mathcal{V}$, in which case $\mathcal{U} \cup \mathcal{W} = \mathcal{U}$ and $\mathcal{U} \cup \mathcal{Z} = \mathcal{S}$). Then recurrence (5.4) degenerates into the tautology *fan_sels*($\mathcal{S}$) = *fan_sels*($\mathcal{S}$)·1 (or, equally unhelpful, *fan_sels*($\mathcal{S}$) = 1·*fan_sels*($\mathcal{S}$)).

This tautological recurrence is the crux of the matter: the tautology tells us that when $\mathcal{S}$ is a two-element set, we are free to set *fan_sels*($\mathcal{S}$) to *whatever value we choose*. In choosing this value, we must respect the selectivity information that is to be supplied as input to the optimizer. Observe that the fan of a two-element set consists of the one predicate, if there is one, that connects the two elements of the set. Then for *fan_sels*($\mathcal{S}$) we may take the externally supplied selectivity of the predicate in question. On the other hand, if there is no predicate connecting the two elements of the set, we again have the empty product, or 1, as the appropriate value for *fan_sels*($\mathcal{S}$). Another way to explain the choice of 1 for *fan_sels*($\mathcal{S}$) in this situation is to imagine that the only join graphs we deal in are cliques. Then to transform a nonclique graph into a clique, it is necessary to add dummy edges; these dummy edges will have selectivity 1 (*cf.* Section 2.5.2).

The singleton sets and sets of two elements are the only anomalous cases. When $\mathcal{S}$ has three elements or more, then if $\mathcal{U}$ = *least_subset*($\mathcal{S}$) and $\mathcal{V} = \mathcal{S} - \mathcal{U}$, we can be certain that $\mathcal{V}$ has at least two elements. Consequently there exists at least one pair $\mathcal{W}, \mathcal{Z}$ of *nonempty*, proper subsets of $\mathcal{V}$ that qualify as a split of $\mathcal{V}$. In this case we are assured that recurrence (5.4) will give us a straightforward means of computing *fan_sels*($\mathcal{S}$) in terms of previously established results. But note that this assurance is obtained only by virtue of the fact that $\mathcal{U}$ = *least_subset*($\mathcal{S}$) is guaranteed to be a singleton; and it is for this reason, and no other, that we insisted that *least_subset*($\mathcal{S}$) be a singleton when we defined it in

[1]The equation *fan_sels*($\mathcal{S}$) = *fan_sels*($\mathcal{S}$) · *fan_sels*($\mathcal{S}$) has a second solution *fan_sels*($\mathcal{S}$) = 0. However, if *fan_sels*($\mathcal{S}$) = 0 for singleton $\mathcal{S}$, then one may deduce *fan_sels*($\mathcal{S}$) = 0 for *all* $\mathcal{S}$. Such an interpretation of the recurrence is mathematically consistent, but of no apparent utility.

Section 5.2.1 above.

Let us now go through the abstract code of the Blitzsplit algorithm piece by piece, revising where necessary to support selectivity computations, and taking care not to misuse the recurrences.

Declarations Figure 5.4 revises the declarations for the Blitzsplit algorithm. The vertical dots indicate that we still have a type *rel_data*, but since it is unchanged from Figure 3.1 (page 75), it is not shown here. However, we have declared a new type *predicate* (shadowboxed) that characterizes a predicate: a *predicate* record identifies the relation names mentioned by the predicate—these are the predicate's endpoints when it is viewed as an edge in the join graph—as well as the predicate's selectivity. As long as hyperedges are excluded, a predicate will have exactly two endpoints, hence the *endpoints* field will be a set of exactly two relation names.

As was true of the *rel_data* entries, one can imagine that there might be additional fields in the *predicate* entries that would be necessary to support some cost models. For example, for some cost models it might be helpful to have access to the *text* (or the abstract syntax tree) of the predicates.

Aside from the addition of a *predicate* type, we have made one other change to the declarations: the addition of the *fan_sels* field in the table entries.

Procedure *blitzsplit* Let us now turn to the procedures of the Blitzsplit algorithm, starting with the top-level procedure *blitzsplit*. The algorithm appears in Figure 5.5 with revisions to support predicates.

The top-level procedure *blitzsplit* has two differences from the version of Figure 3.2. First, *blitzsplit* now has an additional argument, which identifies the predicates and their properties. Second, two new loops have been added to *blitzsplit* to initialize the *fan_sels* fields of the table entries for sets of two relation names.

The first loop assigns a default *fan_sels* value of 1 to every set of two relation names (and also, incidentally, to every singleton set, since we did not specify that R and R' should be distinct). In effect, this loop generates a clique of dummy edges in the join

```
    ⋮

type predicate =
     record
          endpoints : set[relation_name]
          selectivity : real
     end

var  table : array indexed by set[relation_name] of
     record
          fan_sels    : real
          cardinality : real
          best_lhs    : set[relation_name]
          cost        : real
     end
```

Figure 5.4: Changes to declarations to support predicates

graph, and assigns a selectivity of 1 to each such edge. Where a pair of relation names has an actual edge connecting them, the dummy edge will be replaced with the actual one in the second loop.

The second loop iterates through each predicate name supplied in the input, looks up the set of two relation names that the predicate connects, and assigns to that set a *fan_sels* value equal to the predicate's selectivity. Thus, the dummy selectivities are replaced wherever actual selectivities are available.

All initialization of the *fan_sels* entries for the singleton and two-element sets is therefore completed before the start of the final loop in *blitzsplit*, where the entries for smaller sets will be consulted in the computation of entries for larger sets.

Procedure *init_singleton* Procedure *init_singleton* has been changed to initialize the *fan_sels* fields of singleton sets to 1 in accordance with the observations of the beginning of this section. This initialization is not really needed, since it is performed redundantly by the first of the new loops in *blitzsplit*, as noted above. On the other hand, *init_singleton* is the obvious place for this initialization, and there is no harm in performing it twice. (This initialization is cheap, and its execution count is only n.)

```
procedure blitzsplit( R : set[relation_name], rel_data : rel_data,
                                                  P : set[predicate] )
    for each R ∈ R do
        init_singleton(R, rel_data)
    end for

    for each R, R' ∈ R do
        table[{R, R'}].fan_sels := 1.0
    end for
    for each p ∈ P do
        table[p.endpoints].fan_sels := p.selectivity
    end for

    for m := 2 to |R| do
        ⋮
    end for
end procedure


procedure init_singleton(R : relation_name, rel_data : rel_data)
    table[{R}].fan_sels := 1.0
    table[{R}].cardinality := rel_data[R].cardinality
    table[{R}].best_lhs := ∅
    table[{R}].cost := 0.0
end procedure


procedure compute_properties(S : set[relation_name])
    U := least_subset(S)
    V := S − U
    W := least_subset(V)
    Z := V − W
    table[S].fan_sels := table[U ∪ W].fan_sels * table[U ∪ Z].fan_sels
    table[S].cardinality := table[U].cardinality * table[V].cardinality
                                                  * table[S].fan_sels
end procedure
    ⋮
```

Figure 5.5: Changes to Blitzsplit algorithm to support predicates

Procedure *compute_properties* The meat of the changes is in *compute_properties*, the present version of which is a fairly direct transcription of equations (5.4) and (5.5) for computing *fan_sels* and *cardinality*, respectively.

Several points about this code are worth noting. First, although *compute_properties* will never be called with a singleton argument $\mathcal{S}$, it *will* be called for two-element sets. Since we know two-element sets to be an anomalous case, we must pay special attention to how they are handled. If $\mathcal{S}$ has two elements, both $\mathcal{U}$ and $\mathcal{V}$ will be singletons, $\mathcal{W}$ will be the same as $\mathcal{V}$, and $\mathcal{Z}$ will be empty. Therefore, we will perform the assignment

$$table[\mathcal{S}].fan\_sels := table[\mathcal{S}].fan\_sels * table[\mathcal{U}].fan\_sels,$$

which will be useless but harmless: it will multiply the *fan_sels* value associated with $\mathcal{S}$ by 1.0. A possibly more pleasing though less compact alternative would be to check for empty $\mathcal{Z}$, and to skip the assignment in that case.

Second, although we have argued that it is essential that $\mathcal{U}$ be obtained via *least_subset*, it is *not* essential that $\mathcal{W}$ also be obtained in this manner. In defining $\mathcal{W}$ to be equal to $least\_subset(\mathcal{V})$, the revised code overspecifies. What is really intended is this: "$\mathcal{W}$ should be any nonempty, proper subset of $\mathcal{V}$, except in the case where $\mathcal{V}$ is a singleton, for in that case, $\mathcal{V}$ *has* no nonempty, proper subsets; so in that case, $\mathcal{W}$ should be any subset of $\mathcal{V}$ at all." That is what is intended, but it is easier just to say, "$\mathcal{W} := least\_subset(\mathcal{V})$," which is close enough—and which certainly conforms to the intent.

Finally, the interested reader is referred again to the C code in Appendix B. That implementation of the Blitzsplit algorithm does not support predicates; even so, the program *already* calculates $\mathcal{W}$ and $\mathcal{Z}$, and, in fact, already checks for empty $\mathcal{Z}$. Consequently, the changes required for the C code to support predicates and selectivity computations are very slight indeed.

Summary Support for predicates entails changes to each procedure in the Blitzsplit algorithm of Figure 3.2 except *find_best_split*—the vertical dots at the bottom of Figure 5.5 indicate that *find_best_split* remains as before.

The loops added to *blitzsplit* have $O(n^2)$ time complexity, and contribute little to overall execution time of the algorithm. Because the $O(3^n)$ and $O(n2^n)$ parts of the algorithm

are completely unaffected by the changes, and because the time needed by the $O(2^n)$ part has presumably no more than tripled (where there was one floating multiplication before, there are now three), one would expect the impact of the changes on the algorithm's speed to be small, especially for larger n. Measurements bear out this expectation; under the naive cost model (i.e., using the cost function κ_0 defined on page 53), the slowdown at larger n does not exceed a few percent.

The impact on space complexity is also small. We have added an 8-byte field to each entry of *table*. At the beginning of Chapter 4 we estimated the size of an entry at 20 bytes, but we subsequently brought the figure down to 16 bytes by representing costs in a 4-byte floating-point format. Now we are bringing it back up to 24, a 50% increase over 16. But total space usage for $n = 16$ remains just 1.6MB—a 20% increase over our original estimate.

A final issue that we have yet to address is extraction of the optimal plan from the completed table. The process is essentially the same as before, but to be thorough we ought now to annotate the join operators with the predicates attached to them. However, we have made no provision to store those annotations! But they can easily be reconstructed when the optimal plan is extracted from the dynamic programming table. The expense of this reconstruction is small, since the reconstructed annotations are needed only for the $n - 1$ operators of the optimal plan.

5.3 Accommodating Redundant Predicates

The issue of redundant predicates was first raised in Section 2.4.2. Now we consider the problem in more detail, and propose a mechanism for dealing with it.

Recall queries (2.50) and (2.51) from page 43, and consider the following reformulation of (2.51) as a multiway join query:

$$(\textbf{LineItem} \bowtie_{\text{L\_PARTNO=P\_PARTNO}} \textbf{Part}) \bowtie_{\substack{\text{P\_PARTNO=S\_PARTNO} \wedge \\ \text{L\_PARTNO=S\_PARTNO}}} \textbf{Source}. \tag{5.6}$$

The difficulty we encounter in a query of this kind is that the predicate L_PARTNO = S_PARTNO makes no difference to the result of the query, since this predicate is implied by the other two (L_PARTNO = P_PARTNO and P_PARTNO = S_PARTNO). Because the

predicate L_PARTNO = S_PARTNO could equally well be omitted from the query without affecting the result, its selectivity in the context of this query must be 1. But in other contexts the selectivity of this same predicate may be nowhere near 1. We must therefore allow L_PARTNO = S_PARTNO to have its own independent selectivity s, but we must then ignore s (or correct for it) in queries in which L_PARTNO = S_PARTNO turns out to be redundant, as it is here.

5.3.1 Transitive Chains

The situation just described, abstracted somewhat, is depicted graphically in Figure 5.6(a). The graph represents the three-way join of relations A, B, and C under predicates $\widehat{AB}$, $\widehat{BC}$, and $\widehat{AC}$, whose selectivities are taken to be 1/2, 1/2, and 2/5, respectively. We shall assume that these predicates have a particular form: $\widehat{AB}$ is A_$x \prec$ B_x, $\widehat{BC}$ is B_$x \prec$ C_x, and $\widehat{AC}$ is A_$x \prec$ C_x, where $\prec$ is some transitive relation (not necessarily an equivalence relation). Under these circumstances we shall say that $\widehat{AB}$, $\widehat{BC}$, and $\widehat{AC}$ belong to a *transitive chain.*

Formally, we define a transitive chain c to be a triple $c = (\mathcal{R}_c, \mathcal{P}_c, <_c)$ satisfying these conditions:

- $\mathcal{R}_c$ is a set of relation names; in our example, $\mathcal{R}_c = \{A, B, C\}$. (In general, $\mathcal{R}_c$ will be a subset, and typically a *proper* subset, of the set $\mathcal{R}$ of relation names supplied as input to the join optimizer.)

- $\mathcal{P}_c$ is a set of predicate names such that the graph $(\mathcal{R}_c, \mathcal{P}_c)$ is a clique; in our example, $\mathcal{P}_c = \{\widehat{AB}, \widehat{BC}, \widehat{AC}\}$.

- $<_c$ is a total order on the relation names in $\mathcal{R}_c$. (In general $<_c$ will not necessarily coincide with the relation-name ordering $<$ introduced in Section 5.2.) In our example, $A <_c B <_c C$.

- Whenever $R <_c R' <_c R''$ for some $R, R', R'' \in \mathcal{R}_c$, then the predicate named $\widehat{RR''}$ is *logically implied* by the conjunction of $\widehat{RR'}$ and $\widehat{R'R''}$. In our example, since $A <_c B <_c C$, the predicates $\widehat{AB}$, $\widehat{BC}$, and $\widehat{AC}$ must be such that $\widehat{AC}$ is logically

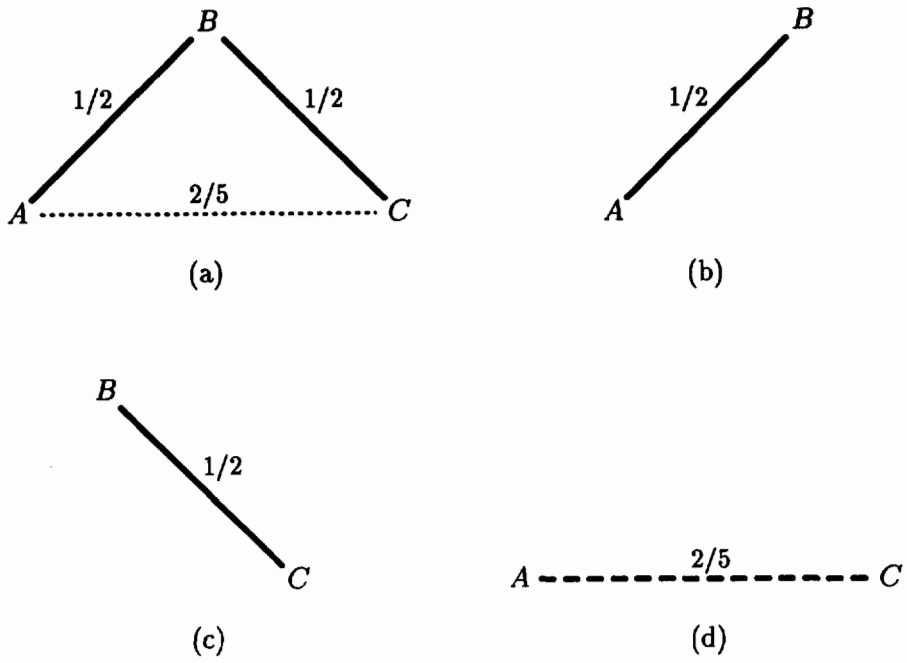


Figure 5.6: Essential and redundant predicates

implied by $\widehat{AB}$ and $\widehat{BC}$. This requirement is satisfied by our example predicates, since $\mathrm{A}\_x \prec \mathrm{C}\_x$ is indeed implied by $\mathrm{A}\_x \prec \mathrm{B}\_x \wedge \mathrm{B}\_x \prec \mathrm{C}\_x$.

Note that the relations and predicates of a query such as query (5.6), in which the transitive relation $\prec$ is in fact the equivalence relation $=$, can be mapped onto the transitive-chain formalism in several different ways. In each such mapping, $\mathcal{R}_c$ would contain **LineItem**, **Part**, and **Source**, and $\mathcal{P}_c$ would contain the three predicates that appear in the query. But $<_c$ could equally well be *any* total order on the names **LineItem**, **Part**, and **Source**, and the conditions of the formalism would be satisfied.

Let us now return to the more abstract example depicted in Figure 5.6(a). We will refer to $\widehat{AB}$ and $\widehat{BC}$ as the *essential* predicates of the chain, since neither can be inferred as a logical consequence of two or more other predicates in the chain; $\widehat{AC}$, on the other hand, is a *redundant* predicate, because it can be inferred from $\widehat{AB}$ and $\widehat{BC}$. In Figure 5.6(a), the essential predicates are shown as solid lines, and the redundant predicate $\widehat{AC}$ as a dotted line. When $\prec$ happens to be an equivalence relation, the distinction between essential and

redundant predicates becomes somewhat arbitrary. However, we still make the distinction in that case, arbitrary though it may be, because it will permit us to treat all chains in the same manner.

Figure 5.6(a) may serve as the join graph for a variety of join expressions. Among the possibilities are the following:

$$(A \bowtie_{\mathrm{A\_}x \prec \mathrm{B\_}x} B) \bowtie_{\mathrm{B\_}x \prec \mathrm{C\_}x \wedge \mathrm{A\_}x \prec \mathrm{C\_}x} C \tag{5.7}$$

$$A \bowtie_{\mathrm{A\_}x \prec \mathrm{B\_}x \wedge \mathrm{A\_}x \prec \mathrm{C\_}x} (B \bowtie_{\mathrm{B\_}x \prec \mathrm{C\_}x} C) \tag{5.8}$$

$$(A \bowtie_{\mathrm{A\_}x \prec \mathrm{C\_}x} C) \bowtie_{\mathrm{A\_}x \prec \mathrm{B\_}x \wedge \mathrm{B\_}x \prec \mathrm{C\_}x} B \tag{5.9}$$

Now plainly the predicate $\mathrm{A\_}x \prec \mathrm{C\_}x$ is redundant in both (5.7) and (5.8); it could be omitted from either without making a jot of difference. But what about expression (5.9)? Omitting $\mathrm{A\_}x \prec \mathrm{C\_}x$ would affect the efficiency of evaluation of (5.9), for without this predicate, the join of A and C would become a Cartesian product. Nonetheless, the omission of this predicate would in no way affect the final result of (5.9), because the presence of the two predicates $\mathrm{A\_}x \prec \mathrm{B\_}x$ and $\mathrm{B\_}x \prec \mathrm{C\_}x$ assures that $\mathrm{A\_}x \prec \mathrm{C\_}x$ must hold for any result tuple. Consequently, for the purposes of determining the result cardinality of expression (5.9), we may regard $\mathrm{A\_}x \prec \mathrm{C\_}x$ as being entirely redundant, just as it was in (5.7) and (5.8).

These examples illustrate the fact that join remains commutative and associative when redundant predicates come into play—as indeed it must, since redundant predicates are still predicates, and what is true of join in the general case must also hold in the special case where the predicates happen to conform to a particular structure. Therefore we may legitimately continue to speak of the join cardinality of a set of relations, and to base our cardinality computations on join graphs and not on join expressions.

The only new development with the introduction of transitive chains is that in a given join graph, some edges may represent redundant predicates whose selectivities ought to be excluded from the cardinality computation. We will deal with this new development by completely separating the transitive chains from the other predicates in a join optimization problem. To this end, we will require that the Blitzsplit algorithm be given an additional argument that characterizes the transitive chains; and in the interest of simplicity, we will

assume initially that we are dealing with just a single chain c. The cardinality computation for a relation set $\mathcal{S}$ will then proceed in three steps:

1. We will compute a preliminary result cardinality for $\mathcal{S}$ by the method of Section 5.2, taking into account only the ordinary, nontransitive predicates.

2. Then we will compute the *selectivity of c in $\mathcal{S}$*, as explained in detail below, to account for the transitive predicates.

3. We will multiply the preliminary result cardinality for $\mathcal{S}$ by the selectivity of c in $\mathcal{S}$ to obtain a final result cardinality for $\mathcal{S}$.

The first step involves nothing new, and the third is immediate, so it is the second step to which we must turn our attention.

5.3.2 Selectivity of a Chain in a Set

Let us explore further the example discussed above. Given a chain c with $A <_c B <_c C$, we saw that in the join of $\{A, B, C\}$, the predicates $\widehat{AB}$ and $\widehat{BC}$ were essential and $\widehat{AC}$ was redundant. The selectivities of the two essential predicates ought to be factors in the cardinality computation for $\{A, B, C\}$, while the redundant predicate should be disregarded. Altogether, then, the predicates of the chain will contribute a factor of $\mathit{selectivity}(\widehat{AB}) \cdot \mathit{selectivity}(\widehat{BC}) = 1/2 \cdot 1/2 = 1/4$. On this basis we may say that the selectivity of c in $\{A, B, C\}$ is $1/4$.

Until now we have been focusing on the set $\{A, B, C\}$ because our example chain c involves exactly the relations A, B, and C. But what is the meaning of the selectivity of c in sets *other than* $\{A, B, C\}$? Let us first consider subsets of $\{A, B, C\}$.

In the singleton $\{A\}$, no predicates come into play, and so the selectivity of c in $\{A\}$ is 1. Similarly for $\{B\}$ and $\{C\}$.

In $\{A, B\}$, only $\widehat{AB}$ comes into play (Figure 5.6(b)), and we may ignore $\widehat{BC}$ and $\widehat{AC}$. Then the selectivity of c in $\{A, B\}$ is $\mathit{selectivity}(\widehat{AB}) = 1/2$. Similarly for $\{B, C\}$ (Figure 5.6(c)). The case of $\{A, C\}$ is interesting (Figure 5.6(d)). Here the only predicate that comes into play is $\widehat{AC}$, which was redundant in the join of $\{A, B, C\}$. But $\widehat{AC}$ is

not redundant in the join of $\{A, C\}$. As we have already remarked, the join $A \bowtie_{A\_x \prec C\_x} C$ would become a Cartesian product if the predicate were omitted. Predicate $\widehat{AC}$ is shown in Figure 5.6(d) as a dashed and not a dotted line because it has become essential in the present context. Accordingly, the selectivity of c in $\{A, C\}$ is $\mathit{selectivity}(\widehat{AC}) = 2/5$.

Next consider a superset of $\{A, B, C\}$. Suppose we are faced with computing the join cardinality of $\{A, B, C, D\}$. If we retain our example chain c without changes, then the join graph for $\{A, B, C, D\}$ will include Figure 5.6(a) as a subgraph. Once again $\widehat{AB}$ and $\widehat{BC}$ will be essential, and $\widehat{AC}$ will be redundant, exactly as in the join of $\{A, B, C\}$. The presence of D makes no difference at all to the effect of the chain c; the selectivity of c in $\{A, B, C, D\}$ is just $1/4$.

Finally, consider the set $\{A, B, D\}$, which is neither a subset nor a superset of the set $\{A, B, C\}$. In the join of $\{A, B, D\}$, $\widehat{AB}$ is the only predicate from c that comes into play, and so the selectivity of c in $\{A, B, D\}$ is $1/2$. Again the presence of D makes no difference.

Generalizing from these examples, one can see that the selectivity of a chain c in an arbitrary set $\mathcal{S}$ is equal to the selectivity of c in $\mathcal{S} \cap \mathcal{R}_c$. Consequently, if we can determine the selectivity of c in just the subsets of $\mathcal{R}_c$ (including $\mathcal{R}_c$ itself), we will obtain a complete characterization of the chain's effect in *all* sets.

5.3.3 Computing Selectivities of Chains

The technique we shall use to compute chain selectivities should come as no surprise. We will use dynamic programming to compute the selectivity of a chain c in all subsets of $\mathcal{R}_c$. This computation will be performed using an auxiliary table separate from the table that we have used in the Blitzsplit algorithm up to this point.

The recurrence that will permit us to apply dynamic programming to chain selectivity computation is straightforward. Before we present it, however, it may be helpful to look at a few examples of chains involving a larger number of relations than we have considered so far.

Suppose the chain c traverses the relations in $\mathcal{R}_c = \{A, B, C, D, E\}$, with

$$A <_c B <_c C <_c D <_c E.$$

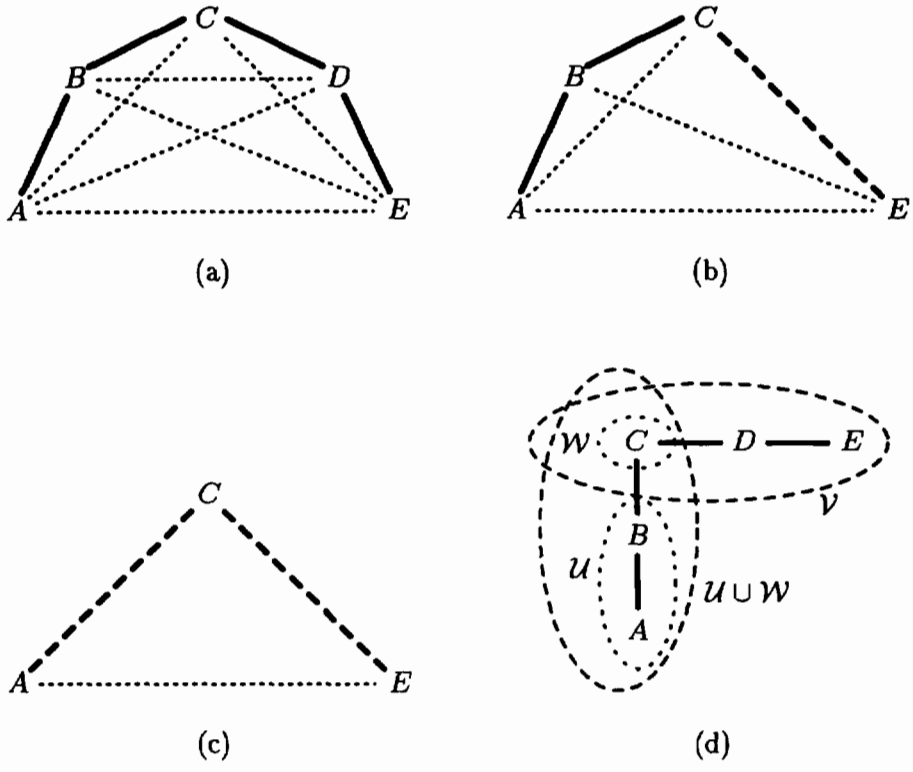


Figure 5.7: Adjacency in a longer chain

As illustrated in Figure 5.7(a), most of the chain's predicates are redundant in the join of $\{A, B, C, D, E\}$. Only the four predicates connecting *adjacent* relation names are essential; we define relation names R and R'' to be *adjacent in $\mathcal{S}$ with respect to* $<_c$ just if R and R'' are distinct and there is no R' in $\mathcal{S}$ that comes between R and R'' in the ordering $<_c$. When two relation names R and R'' are *not* adjacent, and there is an R' such that $R <_c R' <_c R''$ or $R'' <_c R' <_c R$, then the definition of transitive chains provides that $\widehat{RR''}$ is logically implied by $\widehat{RR'}$ and $\widehat{R'R''}$, and hence is redundant.

Now, keeping c the same, consider what happens if we remove D from the set of relation names to be joined (Figure 5.7(b)). The removal of D has caused C and E to become adjacent, and consequently $\widehat{CE}$ is essential in the join of $\{A, B, C, E\}$. In effect, $\widehat{CE}$ takes the place of $\widehat{CD}$ and $\widehat{DE}$, which disappeared along with D. Altogether the removal of D resulted in a loss of two essential predicates and a gain of one, for a net loss of one.

The reader can verify that the removal of any other individual relation name would have similarly reduced by one the total number of essential predicates in the join. The total is now three.

Going a step further and removing B from the join graph in Figure 5.7(b), we obtain in Figure 5.7(c) a graph for the join of $\{A, C, E\}$. The removal of B has resulted in a net loss of one more essential predicate, and now there are two such predicates remaining. In fact, Figure 5.7(c) has exactly the same form as Figure 5.6(a). The fact that $\{A, C, E\}$ is embedded in a chain involving 5 relations has no bearing on the join graph for $\{A, C, E\}$, and all that matters to the determination of the essential predicates in $\{A, C, E\}$ is the *restriction* of $<_c$ to $\{A, C, E\}$. We care about the larger chain in which $\{A, C, E\}$ is embedded only insofar as it will affect the placement of information about $\{A, C, E\}$ in the dynamic programming table for the chain.

The foregoing observations may be summarized as follows. Given $c = (\mathcal{R}_c, \mathcal{P}_c, <_c)$ and nonempty $\mathcal{S} \subseteq \mathcal{R}_c$, the essential predicates from $\mathcal{P}_c$ in the join of $\mathcal{S}$ are just the $|\mathcal{S}| - 1$ predicates $\widehat{RR''}$ with R and R'' adjacent in $\mathcal{S}$ with respect to $<_c$. The selectivity of c in $\mathcal{S}$ is the product of the selectivities of these $|\mathcal{S}| - 1$ predicates. Note that even the somewhat degenerate examples of Figures 5.6(b)–(d) conform to this rule: when $\mathcal{S}$ contains but two relation names, they are necessarily adjacent, and the lone predicate connecting them is therefore always essential. Indeed the rule holds for singleton sets $\mathcal{S}$ as well, since the graph for such a set admits no predicates at all, and hence no essential predicates.

Figure 5.7(d) illustrates the decomposition of a set of relation names—in this instance, $\{A, B, C, D, E\}$—into two overlapping subsets—namely, $\{A, B, C\}$ and $\{C, D, E\}$—whose essential predicates together comprise the essential predicates of the whole. The selectivity of the chain in $\{A, B, C, D, E\}$ is therefore the product of its selectivities in these two subsets. Our recurrence for computing the product of a set's essential selectivities is based on this decomposition.

To state the recurrence, we need to introduce a variant of the *least_subset* operation described in Section 5.2 above. Let *least_in_c* be the operation identical to *least_subset* except that *least_in_c* orders relation names by $<_c$ rather than by $<$. In other words, *least_in_c* is chain-specific; if we encounter a new chain c', we must replace *least_in_c* with

the operation *least_in_c'* based on $<_{c'}$. The desired recurrence is then

$$\begin{aligned} sel\_c(\mathcal{S}) &= sel\_c(\mathcal{U} \cup \mathcal{W}) \cdot sel\_c(\mathcal{V}) \\ \text{where } & \mathcal{S} \subseteq \mathcal{R}_c, \\ & \mathcal{U} \cup \mathcal{V} = \mathcal{S}, \\ & \mathcal{U} <_c \mathcal{V}, \\ \text{and } & \mathcal{W} = least\_in\_c(\mathcal{V}). \end{aligned} \tag{5.10}$$

The notation $\mathcal{U} <_c \mathcal{V}$ should be understood to mean that every element of $\mathcal{U}$ precedes every element of $\mathcal{V}$ in the ordering $<_c$.

5.3.4 Relation-name Aliases

We assumed above that there is a distinct $<_c$ for each chain, and correspondingly a distinct *least_in_c*. But while these assumptions were useful in defining recurrence (5.10), it might be undesirable to have to support a multiplicity of *least_in_c* functions in an implementation. The *least_subset* function corresponding to the relation-name ordering $<$ of Section 5.2 had the attractive property that a simple and highly efficient concrete realization of it had already been given in Chapter 4. Arbitrary *least_in_c* functions corresponding to arbitrary orderings $<_c$ might be more difficult to realize; both the simplicity and efficiency of the optimization code could suffer.

We can get by with a single *least_subset* function for all chains if we are willing to use different names, or *aliases*, for the same relation in different contexts. The approach of using aliases is best illustrated by considering an optimization problem involving more than one chain; our pseudo-code below will provide for just one chain, but the extrapolation to more than one chain, using relation-name aliases, will be straightforward.

Suppose, then, that our join optimization problem involves relations F, G, H, I, and J. Suppose further that F, G, and I participate in a chain c with $F <_c I <_c G$ (we shall refer to this chain informally as the FIG chain); and that F, I, and J participate in a separate chain c' with $J <_{c'} I <_{c'} F$ (the JIF chain). Now in Chapter 4 we found it convenient to refer to relations by the generic names R_0, R_1, etc., with the understanding that these names were just stand-ins for the actual names by which the relations might

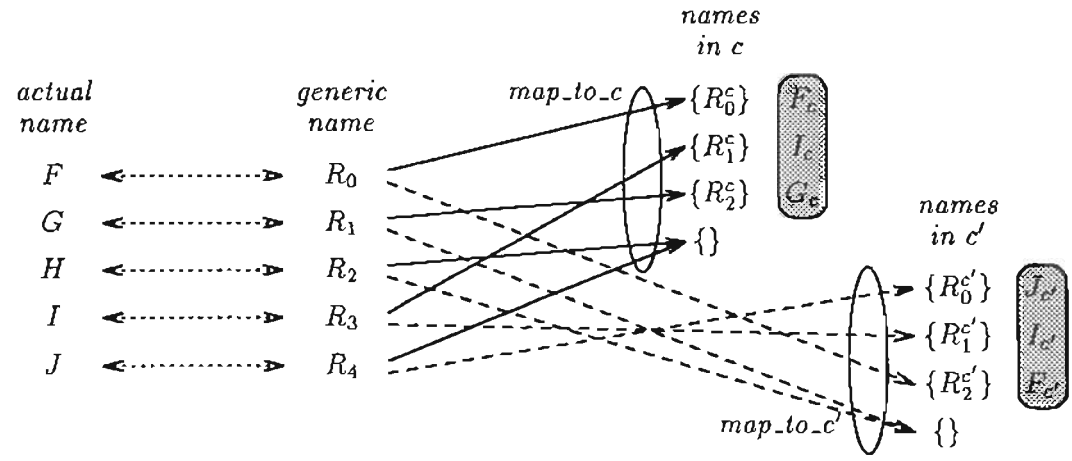


Figure 5.8: Relation-name aliases

be known externally to the optimizer. In the present example, R_0 might stand for F, and R_1 for G, and so on. This association between actual and generic names is depicted in Figure 5.8 by the bidirectional arrows at the left-hand end of the figure.

The assignment of actual names to generic names is unconstrained; an assignment in which R_0 stands for J, R_1 for I, and so on, would work just as well as the assignment in Figure 5.8. However, if we take as given that $R_0 < R_1 < \ldots$, then the assignment of actual names to generic names will affect the implicit ordering of the actual names. For example, the assignment in Figure 5.8 implies the ordering $F < G < H < I < J$.

Ordinarily we are not much concerned with this ordering, for while the ordering makes a difference to the details of our cardinality computations in Section 5.2, it makes no difference to the end results of those computations. But the introduction of chains changes the picture somewhat; some orderings may now appear more attractive than others. Indeed, it is tantalizing to observe that there exist assignments of actual to generic names that yield orderings consistent with the FIG chain (i.e., with $F < I < G$), and that there also exist assignments consistent with the JIF chain. If we were to take a FIG-consistent assignment, then we could use our efficient implementation of *least_subset* for *least_in_c* as well; alternatively, if we were to take a JIF-consistent assignment, our implementation of *least_subset* could double as the implementation of *least_in_c'*.

Unfortunately, though, there can exist no assignment consistent with *both* FIG and JIF. In a similar vein, it would appear that no single implementation can serve for both *least_in_c* and *least_in_c'*, since these functions are intrinsically incompatible. For example, the FIG chain requires $\mathit{least\_in\_c}(\{F, I\}) = \{F\}$, whereas the JIF chain requires $\mathit{least\_in\_c'}(\{F, I\}) = \{I\}$.

Yet there is nothing to stop us from circumventing the difficulty by using different sets of names in different contexts. In our computations involving the FIG chain, let R_0^c stand for F, and R_1^c for I, and R_2^c for G. (These names appear in the right-hand portion of Figure 5.8.) It is to be understood that, by default, indexed names are always ordered according to their indices; thus $R_0^c < R_1^c < R_2^c$, reflecting the ordering $F <_c I <_c G$. Then we will have $\mathit{least\_subset}(\{R_0^c, R_1^c\}) = \{R_0^c\}$, which we may interpret as saying that $\mathit{least\_in\_c}(\{F, I\}) = \{F\}$. On the other hand, in our computations involving the JIF chain, let $R_0^{c'}$ stand for J, and $R_1^{c'}$ for I, and $R_2^{c'}$ for F. Then we may interpret $\mathit{least\_subset}(\{R_2^{c'}, R_1^{c'}\}) = \{R_1^{c'}\}$ as saying that $\mathit{least\_in\_c'}(\{F, I\}) = \{I\}$.

By now we have introduced up to three distinct aliases for each of the actual relation names in our example. This profusion of aliases will cause no trouble as long as we take care to use the aliases in a consistent manner. In a given context we will restrict ourselves to using aliases only of like kind. Specifically, in contexts having nothing to do with chains, we will use the unadorned generic names R_0, R_1, etc., and no others; in the context of selectivity computations for the FIG chain, we will use only the names R_0^c, R_1^c, and R_2^c; and for the JIF chain, we will use only the names $R_0^{c'}$, $R_1^{c'}$, and $R_2^{c'}$. By separating the use of distinct classes of names into mutually exclusive contexts, we pave the way for reusing the same concrete representations for these names across contexts. In Chapter 4, we represented R_k by the integer k; we may continue to do so, and we may likewise represent R_k^c and $R_k^{c'}$ by the integer k, since the meaning of k will be unambiguous in a given context. The same efficient implementation of *least_subset* may be used in all contexts.

5.3.5 Translation of Relation Names

There remains just one more issue to be addressed in connection with relation-name aliases. If a given relation is to be known by more than one name, we need a way to translate

among the different names as we move from one context to the next. When we give pseudo-code for handling chains below, it will be seen that it suffices to translate from names of the form R_k to chain-specific names of the form R_i^c and $R_j^{c'}$—translations in the other direction, or between pairs of chains, are not needed.

The right-hand portion of Figure 5.8 illustrates the translations applicable to our example. The mapping *map_to_c*, represented by the solid arrows, yields aliases consistent with the FIG ordering. For example, *map_to_c* takes R_0, representing F, to R_0^c, which (in the context of the FIG chain) again represents F. In the same spirit, *map_to_c* takes R_3, representing I, to R_1^c, and R_1, representing G, to R_2^c, as required to obtain the FIG ordering. By contrast, the mapping *map_to_c'*, represented by the dashed arrows, yields aliases consistent with the JIF ordering.

In one detail, the description just given of the mappings *map_to_c* and *map_to_c'* is not strictly accurate. Our description implied that these mappings were functions of type *relation_name* $\rightarrow$ *relation_name*—that is, functions which, given a relation name as input, produce another relation name as output. But it will prove more convenient to associate with these mappings the typing *relation_name* $\rightarrow$ **set**[*relation_name*]. Rather than mapping R_0 to R_0^c, as suggested above, *map_to_c* will actually map R_0 to the singleton $\{R_0^c\}$, and similarly for the other cases. The advantage in taking the codomain of these mappings to be **set**[*relation_name*] is that this typing admits the *empty set* as a possible result. For example, since H (represented by R_2) does not participate in the FIG chain, there is no sensible choice of names in c that *map_to_c* can map it to; the empty set is therefore a logical alternative. Similar considerations naturally apply to *map_to_c'*. In Figure 5.8, note that both the solid and dashed arrows emanating from R_2 lead to the empty set, reflecting the fact that H participates in neither the FIG nor the JIF chain.

Now suppose one wished to use *map_to_c* to map a *set* of generic names to a set of names in c. For example, suppose one wished to obtain the set of names in c corresponding to the set $\{R_0, R_2, R_3\}$—which represents the set of actual names $\{F, H, I\}$. One could accomplish this mapping by applying *map_to_c* to each of the elements R_0, R_2, and R_3 in succession—thus obtaining $\{R_0^c\}$, $\{\}$, and $\{R_1^c\}$—and then taking the union of these results to obtain $\{R_0^c, R_1^c\}$—which represents $\{F, I\}$ in c. In short, a set representing $\{F, H, I\}$ is

mapped to a set representing $\{F, I\}$ in c.

Recall from Section 5.3.2 our observation that the selectivity of a chain c in a set $\mathcal{S}$ is the same as the selectivity of c in $\mathcal{S} \cap \mathcal{R}_c$. In other words, relation names in $\mathcal{S}$ that do not participate in c are irrelevant to the selectivity of c in $\mathcal{S}$. In our example, the selectivity of the FIG chain in $\{F, H, I\}$ is the same as its selectivity in $\{F, I\}$. Consequently, the fact that *map_to_c* in effect maps $\{F, H, I\}$ to $\{F, I\}$, and simply discards H, is not only not harmful, but to the contrary, is exactly what we want. Our table of selectivities for the FIG chain will be indexed by sets of names in c. Given a set $\mathcal{S}$ of generic names, we may use *map_to_c* to map $\mathcal{S}$ to a possibly smaller set $\mathcal{S}_c$ of names in c; then taking $\mathcal{S}_c$ as an index into the table of selectivities, we will obtain the selectivity of $\mathcal{S}$ in c, without ever explicitly computing an intersection of the form $\mathcal{S} \cap \mathcal{R}_c$.

5.3.6 Code for Computing Chain Selectivities

We shall break our discussion of the code for handling transitive chains into two parts. In this, the first part, we shall focus on the construction of a table of selectivities for a given chain c, and in the second part we will address the changes required of the Blitzsplit algorithm to incorporate such a table in its cardinality computations.

The construction of a table of selectivities for chain c can take place entirely within the context of the chain; that is, for now we may take all relation names to be names in c—names of the form R_0^c, R_1^c, etc. Later on, when we address the Blitzsplit algorithm proper, the names in c will have to coexist with other kinds of names. To avert the possibility that names in c might then be confused with names in other chains, or with generic names, we shall give each kind of name a different type in the pseudo-code. Specifically, the names R_0^c, R_1^c, and so on, will be referred to as having type *relname_in_c*.

Figure 5.9 gives pseudo-code to construct a table *sel_c* of selectivities of a chain c in all subsets of $\mathcal{R}_c$ (including the empty subset!). The code opens with the declaration of a new type *pred_in_c*, which is exactly the same as the type *predicate*, except that it specifies a predicate's endpoints using relation names in c rather than ordinary relation names. This type declaration is followed by a declaration of the table *sel_c*, the table of selectivities of c in subsets of $\mathcal{R}_c$. This dynamic programming table will have one entry for each subset

```
type pred_in_c =
        record
            endpoints : set[relname_in_c]
            selectivity : real
        end

var     sel_c : array indexed by set[relname_in_c] of real

        ⋮

procedure build_chain_table(R_c : set[relname_in_c], P_c : set[pred_in_c])
    sel_c[∅] := 1.0
    for each R, R' ∈ R_c do
        sel_c[{R}] := 1.0
        sel_c[{R, R'}] := 1.0
    end for
    for each p_c ∈ P_c do
        sel_c[p_c.endpoints] := p_c.selectivity
    end for
    for m := 2 to |R_c| do
        for each S_c ⊆ R_c such that |S_c| = m do
            U_c := least_subset(S_c)
            V_c := S_c − U_c
            W_c := least_subset(V_c)
            sel_c[S_c] := sel_c[U_c ∪ W_c] * sel_c[V_c]
        end for
    end for
end procedure
```

Figure 5.9: Code to calculate chain selectivities

of $\mathcal{R}_c$, and accordingly is indexed by sets of relation names in c. The number of entries in this table is therefore $2^{|\mathcal{R}_c|}$; for typical chains one may expect this table to be quite small.

The procedure *build_chain_table* for filling in the table of chain selectivities is basically a pseudo-code transcription of recurrence (5.10), wrapped in a loop that iterates through all subsets of $\mathcal{R}_c$ in a manner that conforms to the subsets-first assumption. The code prior to the loop initializes the table entries for empty, singleton, and two-relation subsets of $\mathcal{R}_c$.

5.3.7 Changes to the Blitzsplit Algorithm

To accommodate redundant predicates, the Blitzsplit algorithm itself must be revised to make use of the selectivities provided by the procedure *build_chain_table* discussed above.

Figure 5.10 shows revisions to the declarations for the Blitzsplit algorithm. The type *map_to_c* defines a representation for the mapping from relation names of the form R_i to *sets* of relation names of the form R_j^c. This mapping is represented as an array of sets of relation names in c. (As before, we may assume that these sets will in turn be represented as integers. Thus, in the optimization of a join of n relations, a mapping of type *map_to_c* can be expected to be realized as an array of n integers.)

The dynamic programming table *table* is augmented with two new fields. The field *names_in_c* holds, for each set $\mathcal{S}$, the subset of relation names in $\mathcal{S}$ that map to names in $\mathcal{R}_c$; this subset is represented as a set of relation names in c. This set will serve as an index into the table *sel_c* to obtain the selectivity of c in $\mathcal{S}$. The second field added to *table* is *pre_card*, which represents the result cardinality of $\mathcal{S}$ before the chain selectivity is taken into account.

Given these changes to the declarations, Figure 5.11 shows the needed revisions to the procedures of the algorithm. The revised algorithm takes a chain as input in addition to its other inputs. The chain is specified through the sets $\mathcal{R}_c$ and $\mathcal{P}_c$, and through a mapping *map_to_c* of type *map_to_c*. Thus, the ordering of the relation names in $\mathcal{R}_c$ is implicit—we assume that this ordering is provided by the type *relname_in_c*; it is the role of the mapping *map_to_c* to map relation names in $\mathcal{R}$ to names in $\mathcal{R}_c$ in such a way that the names in $\mathcal{R}_c$ are ordered appropriately for the chain.

Given these inputs, procedure *blitzsplit* begins by invoking *build_chain_table* to initialize the table *sel_c*. The only other change to procedure *blitzsplit* is that it helps *init_singleton* to initialize the *names_in_c* fields of the singleton sets. If a set R does not participate in the chain c, then *map_to_c*$[R]$ will just be the empty set; if R *does* participate in the chain, then *map_to_c*$[R]$ will be a singleton set of the form $\{R^c\}$.

Procedure *init_singleton* initializes the new *pre_card* field in the same way as the

```
see also declarations in Figure 5.9

    ⋮

type map_to_c = array indexed by relation_name of set[relname_in_c]

    ⋮

var  table : array indexed by set[relation_name] of
     record
         names_in_c : set[relname_in_c]
         fan_sels   : real
         pre_card   : real
         cardinality : real
         best_lhs   : set[relation_name]
         cost       : real
     end
```

Figure 5.10: Changes to declarations to support use of chains

cardinality field, because a singleton set of relations cannot involve any redundant predicates, and hence there is no distinction to be made between *pre_card* and *cardinality* for singletons.

In *compute_properties*, the *names_in_c* field is computed using a trivial recurrence. If we temporarily disregard the type distinction we are now making between relation names in general and relation names in c, the recurrence may be stated as follows:

$$\mathcal{S} \cap \mathcal{R}_c = (\mathcal{U} \cap \mathcal{R}_c) \cup (\mathcal{V} \cap \mathcal{R}_c) \quad \text{where} \quad \mathcal{U} \cup \mathcal{V} = \mathcal{S}. \tag{5.11}$$

This recurrence is a direct consequence of the distributivity of set intersection over set union, and draws on the fact that the *names_in_c* field of each set $\mathcal{S}$ may be loosely thought of as representing $\mathcal{S} \cap \mathcal{R}_c$.

Then the *pre_card* field is computed as *cardinality* was computed before the introduction of redundant predicates—for *pre_card* represents cardinality before redundant predicates are taken into account. Finally, *cardinality* itself combines the cardinality in *pre_card* with the selectivity of c in $\mathcal{S}$, as given by the auxiliary table *sel_c*.

```
procedure blitzsplit( R : set[relation_name], rel_data : rel_data, P : set[predicate],
                      [R_c : set[relname_in_c], P_c : set[pred_in_c], map_to_c : map_to_c] )
    [build_chain_table(R_c, P_c)]
    for each R ∈ R do
        init_singleton(R, rel_data, [map_to_c[R]])
    end for
        ⋮
end procedure

procedure init_singleton(R : relation_name, rel_data : rel_data,
                                                [U_c : set[relname_in_c]] )
    [table[{R}].names_in_c := U_c]
    table[{R}].fan_sels := 1.0
    [table[{R}].pre_card := rel_data[R].cardinality]
    table[{R}].cardinality := rel_data[R].cardinality
    table[{R}].best_lhs := ∅
    table[{R}].cost := 0.0
end procedure

procedure compute_properties(S : set[relation_name])
    U := least_subset(S)
    V := S − U
    W := least_subset(V)
    Z := V − W
    [table[S].names_in_c := table[U].names_in_c ∪ table[V].names_in_c]
    table[S].fan_sels := table[U ∪ W].fan_sels * table[U ∪ Z].fan_sels
    table[S].[pre_card] := table[U].[pre_card] * table[V].[pre_card]
                                               * table[S].fan_sels
    [table[S].cardinality := table[S].pre_card * sel_c[table[S].names_in_c]]
end procedure
    ⋮
```

Figure 5.11: Changes to Blitzsplit algorithm to support use of chains

5.4 Summary and Discussion

In this chapter we have shown how the Blitzsplit algorithm can be extended to accommodate predicates by adjusting its cardinality computations with the appropriate predicate selectivity factors. We began by restricting our attention to collections of independent predicates, and then went on to consider collections of predicates that were related to one another through logical implications. In both cases, we were able to modify the Blitzsplit algorithm's cardinality computations without changing any code in procedure *find_best_split*, which is responsible for the $O(3^n)$ and $O(n2^n)$ components of the algorithm's complexity. The code we added had time complexity $O(2^n)$, and the changes we sketched also had only a small effect on space complexity.

It is quite conceivable that variations on the techniques illustrated here could be used to support predicates under different sets of assumptions as well. For example, one might wish to provide for predicates that were statistically correlated, though free of any logical interdependencies. Another interesting and useful extension would be support of hyperedges in the join graph. These extensions would probably prove more challenging to support than the cases illustrated here, but there is no evident reason why it should not remain possible to compute cardinalities in time $O(2^n)$ by taking advantage of the dynamic programming context. Regardless of what kinds of predicates are supported, there are only $O(2^n)$ sets of relations for which a result cardinality needs to be computed; and as we have seen, the computations for a given set can be shortened through judicious use of information previously computed for smaller sets.

However, the foregoing remarks oversimplify somewhat, for the presence of predicates can affect time complexity in subtle ways that we have not yet discussed. Our comments above presumed that because the *code* for *find_best_split* remained unchanged when predicates were added, it could be taken for granted that the *time* spent in *find_best_split* would also remain unchanged. That presumption is not quite correct. The addition of predicates to a given Cartesian product optimization problem can in fact cause the time spent in *find_best_split* to go up—or down—for the following reason: Predicates change the *costs* associated with sets of relations in ways that can affect the execution frequency

of the conditional code in *find_best_split*. Because of this effect, the Blitzsplit algorithm will prove sensitive to join-graph topology, even though our predicate-handling technique treated all join graphs as if they were cliques. Empirical studies of this sensitivity to join-graph topology, as well as other performance traits of the algorithm, will be the focus of the next chapter.

Chapter 6

Performance Analysis

We now examine the performance of the Blitzsplit algorithm in the presence of predicates. The mission of this chapter is two-fold: to present measurements of the algorithm's performance on a wide range of input queries, and to offer detailed explanations for the performance behavior we observe. Our explanations will be informal, but in principle it ought to be possible to recast these explanations in the form of mathematical models. Such models might be expected to have the power to predict performance of our algorithm under a greater variety of conditions than we consider in our measurements. Thus, the performance observations of this chapter are chiefly empirical, but have been conducted with a view to pursuing a more analytical treatment in future work.

We begin by discussing our experimental design for measuring the performance of our join optimizer. After presenting a summary of the measurements we obtained using this design, we go on to attempt to interpret these results. Our interpretive efforts lead to an investigation into the frequency with which the Blitzsplit algorithm needs to carry out complete cost computations on the various "splits" of sets of relations that it examines in its inner loop. We conclude with a commentary on the implications of this analysis.

6.1 Experimental Design

In this section we present our approach to benchmarking the Blitzsplit join optimizer. We begin by discussing general problems in studying performance of a join-order optimizer. We then describe our way of dealing with those problems, and present the salient features of a parameterization of the join-query space that we use in our empirical studies. (The

full details of this parameterization are relegated to Appendix C.) We also point out some of the limitations of this parameterization, which should be regarded as a starting point, not the last word, in the systematic study of join-optimizer performance.

6.1.1 Difficulties in Empirical Studies

Earlier, in Chapter 4, we examined the Blitzsplit algorithm's performance in Cartesian product optimization. We noted there that it was difficult to obtain a thorough characterization of performance because of the large *dimensionality* of the space of possible queries. We sidestepped the difficulty by considering only a fixed number of relations in each query, and by parameterizing the base-relation cardinalities along just two dimensions.

Those techniques of dimensionality reduction remain applicable here. However, now we face new difficulties, because the introduction of predicates also introduces many additional dimensions to the space of possible queries. In the worst case, there are $n(n-1)/2$ predicate selectivities, which may all vary independently. Moreover, in the space of selectivity combinations there is no obvious analogue to the two-dimensional parameterization that we used to generate cardinalities in Chapter 4.

A further complication in studying performance of join-order optimization is that it no longer suffices to use just the naive cost model κ_0 that we applied to Cartesian products. In essence there is only one way to compute a Cartesian product, and the cost of this computation is always roughly proportional to the product of the input cardinalities (and hence to the result cardinality). But there are many alternative join-computation strategies; join-optimization performance measurements lack realism if they fail to take these alternatives into account.

The present study is not the first to encounter these complications. Join-order optimization, unlike Cartesian product optimization, has an extensive literature. But in turning to previous performance studies as a guide, one finds no ready solutions to the problem of constructing effective test suites. These studies have not converged on a single, standard benchmark for join-order optimization, but instead take many disparate approaches, each with its own particular strengths. Reproducing the experimental conditions of these various approaches may involve guesswork; in some instances, the published

description of the measurement methodology is evidently intended only to give a sense of the approach, not to permit duplication of the experiments [20, 28].

Even when the descriptions are more thorough [54, 55], experimental conditions may be difficult to duplicate because of their dependence on the use of a particular pseudo-random number generator and seed. (Understandably, published accounts of the experiments rarely (if ever) specify such details as random number generation.) Random number generation arises when a benchmark seeks to report average behavior of an optimization method over some class of queries. The corresponding variances are usually not reported, but one suspects that they are often large,[1] and that the observed averages are not necessarily close to the true averages. In such cases, the choice of random number generators is an issue, as one might observe rather different averages if a different random number generator were used.

The lack of a single, commonly accepted, and well-defined benchmark for join-order optimization makes it difficult to compare different algorithms on the basis of speed and effectiveness. Presentations of new algorithms in the literature generally do include such comparisons with earlier algorithms [12, 59]; producing these comparisons necessitates *reimplementing* the earlier algorithms [12] unless pre-existing implementations are readily available. This reimplementation effort is unfortunate both because of the labor it entails, and because of the danger that it could introduce performance bugs (or indeed other bugs) in some of the algorithms involved. Such bugs could conceivably go undetected.

Some benchmarks yield misleading results. Steinbrunn [54] reports an instance of unintentional benchmark bias in measurements he and his collaborators had made in an earlier study. In the earlier study [55], they had found that a join-optimization heuristic known as RDC performed competitively with other heuristics; however, subsequent experiments using a more sophisticated cost model revealed that the RDC heuristic was actually rather fragile, and worked well only under special conditions.

[1] Graphical presentations of averages at each of various parameter settings give clues about variance even when the variance is not reported. Smooth graphs tend to correspond to low variances. Jagged graphs may be a sign of large variances—though not necessarily: in Chapter 9 we shall see graphs in which the jaggedness is intrinsic, and not due to measurement uncertainties.

6.1.2 Our Measurement Approach

It is beyond the ambitions of the present work to propose a standard benchmark for join-order optimization. At the same time, there is no single previous benchmark that appears adequate for our purposes. Our measurement approach is based on the following premises:

- Our test queries should be easy to duplicate.
- Reported performance should give a sense of which queries are easiest to optimize, and which are hardest. Averages taken across multiple queries obscure the patterns of variation, and are therefore to be avoided.
- Cardinality values that occur in actual queries cannot be assumed to fall within any particular range. Hence the test queries should cover a very wide range of cardinalities.
- The test queries should use a variety of distinct join-graph topologies.
- Measurements should be run using several different cost models, as no single cost model can be considered definitive. Using cost models that have appeared in previous studies is preferable to inventing new cost models.

To satisfy the last two requirements, we borrow from the comprehensive survey by Steinbrunn et al. [54, 55]. Steinbrunn's measurements were run using several different join-graph topologies and cost models; here we use a subset of those topologies and cost models. But we depart from Steinbrunn in the assignment of cardinalities and selectivities. We parameterize the base-relation cardinalities along two dimensions, as in Chapter 4; we calculate the selectivities according to a formula that tends to minimize the variability in the intermediate-result cardinalities yielded by different sets of relations. Thus, our test queries remain entirely deterministic, and are parameterized along four dimensions in total: one for the cost model, one for the join-graph topology, and two for the base-relation cardinalities. We shall comment on particular details of this parameterization below; the complete details are summarized in Appendix C.

Aside from the fact that we have added two new dimensions for the cost model and join-graph topology, the present parameterization differs from that of Chapter 4 in two

respects. First, here we fix the number of relations n at 15 rather than at 10. One obtains qualitatively similar behavior across the different values for n, but the effects we shall observe become quantitatively more interesting with larger n; the choice of 15 is near the upper end of the query sizes that can be handled in reasonable time. Second, we have replaced the "max/min" cardinality parameter with a parameter we simply call "variability," which ranges from 0 to 1. When variability is 0, the base-relation cardinalities are all equal; as variability rises, the cardinalities spread out. Thus, variability plays much the same role as the max/min parameter; the difference is that the new parameterization does not yield fractional cardinalities between 0 and 1, which cannot arise in base relations in actual queries.

6.1.3 Shortcomings of our Parameterization

Our join-graph parameter ranges over four diverse topologies: the *chain*, the *cycle* augmented with three cross-edges (which we refer to as "*cycle*+3"), the *star*, and the *clique*. Although these alternatives provide diversity, they also leave huge gaps in the sense that the topology of a given actual query might not be especially similar to any of the four we consider.

Moreover, for a given join-graph topology, we let the *cardinality rank* of each relation determine its position in the join graph. For example, in the *star* graph, we always place the relation with the largest cardinality at the hub of the star. In the case of *chain* and "*cycle*+3" graphs, we assign the relations to the positions in the join graph in such a way that each predicate tends to connect a relation whose cardinality is *below* the median to another relation whose cardinality is *above* the median. (The details of these policies are given in Appendix C.) The predicate connections of an actual query might be organized in an entirely different manner. Limited experimentation with variations on our default cardinality arrangements has so far failed to turn up any cases where optimization times depart by a large amount from the optimization times reported below; but our failure up to this point to find such cases may merely mean that we have not yet looked hard enough.

A further deficiency in the join-graph parameterization is that we make no allowance for varying the selectivities independently from the base-relation cardinalities. Within

each topology, the selectivity of a given edge is completely determined by the cardinalities of the relations it connects, in accordance with the formula in Appendix C. Thus, our measurements give no systematic account of the manner in which performance may vary depending on selectivity values.

This deficiency may not in fact be grave, for we have observed from *ad hoc* experiments that performance is not especially sensitive to the selectivity values. But we presume that the selectivity values must make a difference in some situations, for the simple reason that the join-graph topology itself makes a difference. In limiting cases (i.e., as certain selectivity values approach 1), distinct topologies can "morph" into one another. In the regions of transition there would have to be sensitivity to the selectivities. However, our measurement strategy fails to identify those regions.

Instead, for a given join-graph topology our formula for computing selectivities is intended to give near-worst-case values, so that our measured timings are more likely to overstate than understate the timings that would be obtained with different selectivities. Later in this chapter we give evidence that our choices are in fact near-worst-case. At the same time, that evidence will suggest that extensive additional measurement would be needed to gauge the performance of our algorithm under less pessimistic conditions.

Finally, our parameterization of the cost model leaves much to be desired. The three models we use, which are defined in detail in Appendix C, are the naive model introduced in Section 3.2 (κ_0), a sort-merge cost model (κ_{sm}), and a disk-nested-loops model (κ_{dnl}). There is precedent for these cost models, as all three are drawn from the performance analysis of Steinbrunn et al. [55]; moreover, they are appealing choices for benchmarking purposes because of their simplicity, and because they are very different from one another. But they lack realism. Steinbrunn's revised survey [54] abandons the cost models of the earlier version, and instead adopts a single, more complicated and more realistic cost model that combines features of the earlier cost models, as well as adding new features. One surmises that it was chiefly this change in cost models that revealed the fragility of the RDC heuristic, as noted above.

The use of simplistic cost models may not have such severe consequences here as it did in benchmarking the RDC heuristic. Since our optimization method is a form of

exhaustive search, the *quality* of the solutions obtained is not an issue, as it is for stochastic and heuristic methods. What is at issue here is merely the *time* required for optimization. The simplicity of the cost models used here has the virtue of revealing fundamental effects in the performance of the Blitzsplit algorithm. Future studies will be needed to examine how these effects are altered by the use of more realistic cost models.

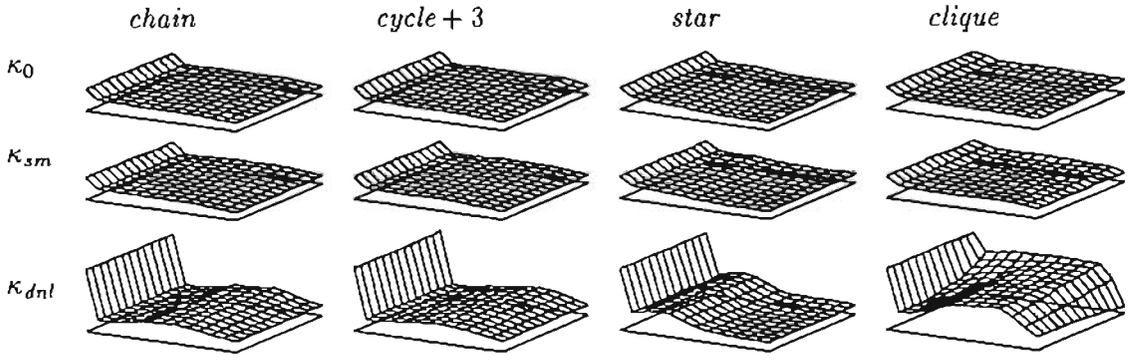
6.2 General Performance Traits

The parameterization of the join-query space discussed above, and mapped out in detail in Appendix C, served as the basis for empirical measurements of the performance of the Blitzsplit algorithm with predicates. Here we present a summary of those measurements, together with general observations about their significance.

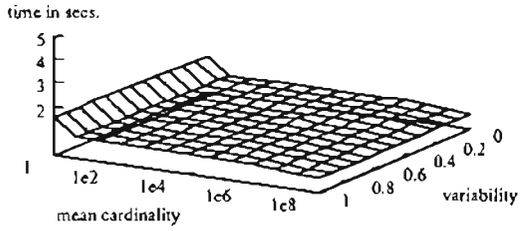
The array of graphs in Figure 6.1(a) shows HP 9000/755 timings of runs of the Blitzsplit algorithm taken over our 4-dimensional parameter space with $n = 15$. The rows and columns of the array represent two axes of the space, a cost-model axis and a join-graph-topology axis; and within each cell inside the array, two more axes are represented—a long axis for mean base-relation cardinality, and a short axis for the variability among the base-relation cardinalities. Moving left-to-right along the long axis corresponds to increasing mean base-relation cardinality, and moving back-to-front along the short axis corresponds to increasing variability among the base-relation cardinalities. (Again, refer to Appendix C for details.) Figures 6.1(b) and (c) show two of the array cells in enlarged form, with labeled axes to give a sense of scale. Note that the vertical axis represents *optimization time* (and not plan cost).

Figure 6.1 shows that under the naive cost model, 15-way joins are optimized in times comparable to those we obtained for 15-way Cartesian products in Chapter 4. On the HP, the latter were typically optimized in about 0.9 seconds; here it is harder to say what is "typical," but optimization time under the naive model rarely falls outside the range 0.6–1.1 seconds. Incorporating predicates appears to make the execution time of the Blitzsplit algorithm more variable, but not necessarily greater.

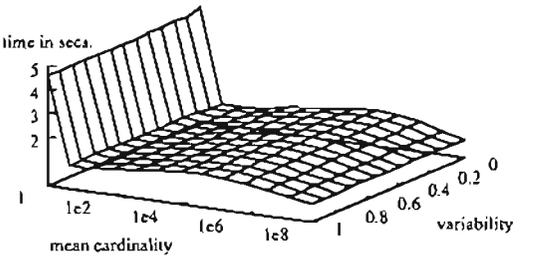
The chaise-longue-like shapes in the figures reflect two basic performance properties



(a) Four-dimensional summary of performance sensitivities



(b) $\kappa_0/chain$



(c) $\kappa_{dnl}/cycle + 3$

Figure 6.1: Optimization times for 15-way joins under various conditions

of our algorithm. First, as we saw already to be the case for Cartesian products in Chapter 4, performance degrades (sometimes dramatically) as the mean cardinality of the base relations approaches 1; but mean cardinality does not have to be large to escape this effect. Second, performance is substantially affected by the cost model, but the performance differences diminish as mean cardinality increases (and also, in the case of cliques, as variability increases).

We shall investigate these effects in detail as we proceed, but here we comment briefly on what is new in the present performance graphs in comparison with those of Figure 4.9. In Chapter 4 we were dealing with just a single cost model, the naive cost model represented by the cost function κ_0. Because the split-dependent component κ_0^{split} of that cost function was trivial, it did not matter how many times κ_0^{split} was executed in procedure

find_best_split (page 76). At low mean cardinalities, the execution frequency of the conditional code in *find_best_split* increased, but since the code in question did not perform much computation, the effect on overall execution time was not dramatic.

Here we are dealing with several different cost models, and we see just how much the execution count of κ^{split} can vary, and how much difference it can make when it does vary. When we introduced a nested **if** structure in Section 3.4.3 to reduce the execution count of κ^{split} (*cf.* page 86), we observed that with that structure, the κ^{split}-execution count could be expected to lie between $(\ln 2/2)n2^n$ and 3^n. At the left-hand edge of the graphs in Figure 6.1, where mean cardinality is 1, the κ^{split}-execution counts are presumably close to 3^n; and we see that when κ^{split} involves nontrivial computation, as it does in the disk-nested-loops model (κ_{dnl}), the evaluation of κ^{split} evidently represents the bulk of the optimization effort. As the mean cardinality moves away from 1, it is apparent that the κ^{split}-execution count drops precipitously. The structuring technique of Section 3.4.3 thus appears to be validated empirically.

A second point of contrast to note between Figure 4.9 and Figure 6.1 concerns the behavior of the Blitzsplit algorithm at higher cardinalities. In Figure 4.9(a) we saw that as mean cardinality increased, optimization time fell off rapidly because of cost overflows. But it was also the case that those cost overflows reflected optimization failures: when cost overflowed, no optimal plan could be found.

Here the decline in optimization time with increasing cardinalities is less pronounced, because the presence of predicates has a moderating effect on intermediate-result cardinalities; and when the cardinalities are moderated, cost overflows naturally occur much less frequently. Nonetheless, they do still occur, and the reductions in optimization time at higher cardinalities are apparent in Figure 6.1. But notably, the occurrence of cost overflow in the presence of predicates does not necessarily reflect an optimization failure. It is possible for cost to overflow for some sets of relations $\mathcal{S} \subsetneq \mathcal{R}$, and for an optimal plan for the join of $\mathcal{R}$ to be found nonetheless. In Chapter 7 we shall make use of this effect to improve on the performance graphs shown in Figure 6.1.

6.3 Execution Counts and Fingerprints

The observations of the previous section suggest, not surprisingly, that variability in the execution count of κ^{split} accounts for most of the variability in the Blitzsplit algorithm's optimization time for different queries. It is also the κ^{split}-execution counts that determine what sort of degradation in performance we will encounter if we move to more elaborate cost models, in which computation of κ^{split} will involve more work. One might even hope that by understanding κ^{split}-execution counts, one could *predict* the performance of our optimizer under a new cost model for which it had not yet been tested.

Unfortunately, such predictive power may be difficult to achieve. A complicating factor in the study of κ^{split}-execution counts is that these counts are not independent of the cost model. Thus, when we change cost models, we can expect changes in both the *number* of κ^{split}-executions and in the *time* required for each of them. Nonetheless, κ^{split}-execution counts offer valuable insight into the algorithm's behavior, and we shall study them in depth.

To establish a baseline, we shall begin by considering κ^{split}-execution counts under the naive cost model. Under this model, the *time* required for each κ^{split}-execution is zero, and the *number* of κ^{split}-executions is exactly equal to the number of updates of *best_cost_so_far*. (We are assuming that the invocation of κ^{split} is enclosed in the nested **if** structure of Figure 3.4 (page 86).) Since we have an analytical estimate for the latter quantity (namely $(\ln 2/2)n2^n$ in the "average" case), it follows that this same formula predicts the typical κ^{split}-execution count under the naive model. By comparing actual execution counts against the prediction, we may gain insight into influences on cost-function execution counts that did not enter into our original estimate.

Subsequently we shall study κ^{split}-execution counts under the disk-nested-loops cost model. For completeness, perhaps we ought to study κ^{split}-execution counts under the sort-merge cost model as well. However, the graphs in Figure 6.1 suggest that our algorithm's performance characteristics under the sort-merge cost model do not differ greatly from those under the naive cost model, and lack the interesting features seen under the disk-nested-loops cost model. Our inclusion of the sort-merge cost model up to this point has

served chiefly to show that a nontrivial cost computation need not inflate optimization times significantly.

6.3.1 Join-query Fingerprints

To get a better handle on κ^{split}-execution counts, let us now narrow our focus. Rather than considering the *total* number of executions of κ^{split} in the optimization of a query, we shall look at the κ^{split}-execution count during just a small portion of the optimization run.

Recall that the main loop in procedure *blitzsplit* (page 76) successively processes sets consisting first of two relation names, then three relation names, and so forth. In particular, there comes a time when all the sets of *seven* relation names are processed; and for the present we shall focus our attention on the processing of just these seven-relation sets. (It is immaterial to the present discussion that the processing of seven-relation sets may in reality be interleaved with the processing of sets of other sizes, as we saw in Chapter 4.) Thus, staying within the context of 15-way join optimization, we consider the number of executions of κ^{split} in the processing of sets such as $\{R_0, R_1, R_2, R_3, R_4, R_5, R_6\}$, $\{R_2, R_4, R_6, R_8, R_{10}, R_{12}, R_{14}\}$, and so on.

Assuming that it is reasonable to study sets of a particular size, why is *seven* a good size to look at? One might equally well have chosen to examine sets of a different size, but the size seven has the advantage of lying approximately midway between the smallest and largest sizes, and therefore holds promise of being at least somewhat representative of other sizes. Moreover, in a 15-way join optimization problem, there are *many* seven-relation sets to be processed. To be precise, there are $\binom{15}{7} = 6435$ such sets; only the eight-relation sets occur in equal abundance, since $\binom{15}{8} = \binom{15}{7}$. At sizes beyond eight (or below seven), the numbers of sets taper off: there are 5005 sets of size nine, 3003 of size ten, and so on. Because they are few in number, the very largest and very smallest sets presumably do not have a major influence on performance. Consequently, our observations regarding the seven-relation sets are likely to remain pertinent even if it should turn out that these sets are not especially representative of the sets lying at the extremes.

Now let us consider the optimization of a specific 15-way join problem from our tests

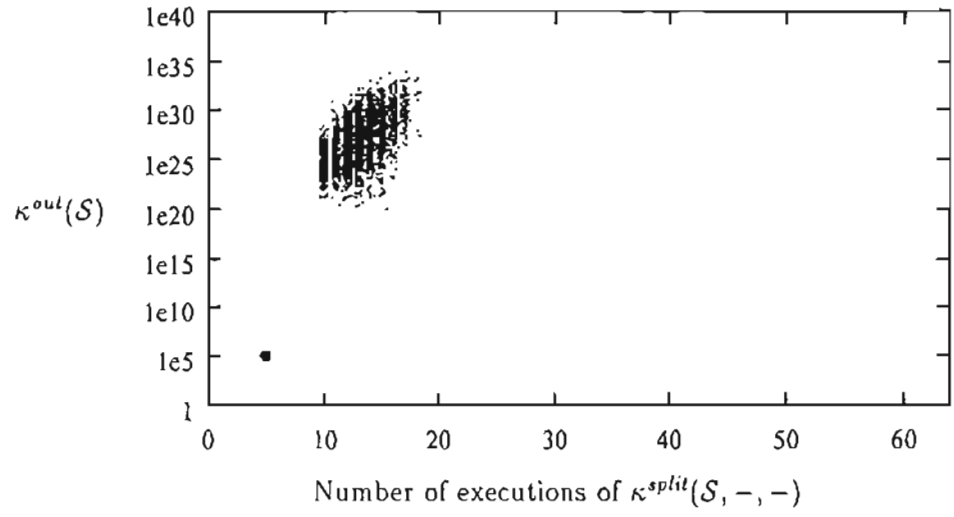


Figure 6.2: Fingerprint for a sample query

of Section 6.2, namely the star query whose base relations have a mean cardinality of 10^4 and variability of 0.5. In other words, the problem we are considering corresponds to the point at the very center of the surface in the $\kappa_0/star$ cell of Figure 6.1. Supposing that this problem is submitted to *blitzsplit* for optimization, the question we now seek to answer is the following: What determines the number of executions of κ^{split} during the processing of the seven-relation sets?

One way to characterize the executions of κ^{split} for the seven-relation sets—and by extension, for the query as a whole—is to take a *fingerprint* of the optimization process in the manner illustrated in Figure 6.2. There are 6435 points in the plot—one for each seven-relation set. The position of each dot is determined by properties of the set $\mathcal{S}$ that the dot represents, as follows.

- The x-coordinate is the number of executions of κ^{split} that occur in the search for the best split of $\mathcal{S}$. The x-axis runs from 0 to 63 because our algorithm will loop through 63 splits of each seven-relation set. (A seven-relation set actually has $2^7 - 2 = 126$ possible splits, but recall that under symmetric cost models, only half of them, or 63, need to be considered.)

- The y-coordinate (which is scaled logarithmically) is the value obtained for $\kappa^{out}(\mathcal{S})$. The y-axis runs only as high as 10^{40} because we deem cost distinctions above this value to be of no interest: as noted in Section 4.8, plans whose costs are this large cannot possibly have practical utility. In our plots, y values larger than 10^{38} are shown as being equal to 10^{38}.

Since in the example at hand we are using the naive cost model κ_0, and since $\kappa_0^{out}(\mathcal{S}) =$ Cardinality($\mathcal{S}$), an alternative reading of the y-coordinates is that they represent the join cardinalities of the various seven-relation sets.[2] With other cost models, too, one may expect a close connection between $\kappa^{out}(\mathcal{S})$ and the join cardinality for $\mathcal{S}$; so a reading of the y-coordinates as a representation of join cardinality should be at least approximately valid for a variety of cost models.

There is a tendency for multiple points in the plot to fall so close to one another that they would be indistinguishable from a single point if they were plotted with absolute fidelity. To counter this effect, with the aim of allowing each set $\mathcal{S}$ to make a contribution to the plot, we have smeared the points out somewhat from their true positions. (The displacement is within ± 0.4 on the x-axis, and within a factor of $\sqrt{10}$ on the y-axis.) Despite the smearing, there are still areas in the plot where points occur in such heavy concentrations that they pile up on top of one another. The blotch at coordinates $(5, 10^5)$ is one such area, whose actual density is not evident from the plot. This small blotch actually comprises about 3000 points, or nearly half of the points in the entire plot. The fingerprints of all our star queries have such a blotch, and in examining the plots it is helpful to think of these corner blotches as loci of enormous mass, like neutron stars.

6.3.2 Significance of Fingerprints

Having discussed above the mechanics of reading Figure 6.2, we now turn to the interpretation of the information in the plot.

At the most basic level, a fingerprint that lies mostly near the left-hand edge of the plot is good news; a fingerprint near the right-hand edge of the plot is bad news. The further

[2]Recall that Cardinality($\mathcal{S}$) refers not to the cardinality of the set $\mathcal{S}$ of relation names, but to the cardinality of the relation that results from the join of the relations named by $\mathcal{S}$.

to the right the fingerprint's center of mass, the more effort the optimizer is expending in calculating the split-dependent costs given by κ^{split}. The vertical positions of the points, by contrast, have no immediate bearing on optimization effort. But the vertical position is informative in other ways, and later we will see that it bears on optimization effort after all.

Note that, for the query at hand, the plot indicates that there are two categories of seven-relation sets: those whose join cardinality (as depicted by vertical point position) is low, and those whose join cardinality is high. This dichotomy has a simple explanation. The query described by the plot is a star query, with some relation at the hub of the star; consequently, the seven-relation sets may be divided into those that include the hub, and those that do not. The sets lacking the hub induce join subgraphs with no edges, and hence join as Cartesian products—giving large result cardinalities; whereas the sets that include the hub induce join subgraphs that are themselves stars, and join to yield results of modest cardinality. Indeed, because the selectivities in our test queries are chosen to counteract the variation in the base-relation cardinalities, the join cardinalities of all sets that include the hub are actually identical.

Next let us examine in more detail the x-coordinate values, i.e., the κ^{split}-execution counts exhibited for the various seven-relation sets. Recall the reasoning we pursued in Section 3.4.2 in obtaining our estimate for the total execution count of the innermost conditional block in *find_best_split*. We assumed that the splits would be examined in random order; from this assumption we argued that in *find_best_split*'s search for the best split of a given m-relation set $\mathcal{S}$, the expected number of executions of the innermost block was $H_{2^m} \approx m \ln 2$. Given the naive cost model, κ^{split} is executed if and only if the innermost block is executed, so we should expect the κ^{split}-execution count also to be about $m \ln 2$. Here we are looking at seven-relation sets, but m is effectively 6 since only 2^6 splits are to be considered. Hence, on average, we expect to see $6 \ln 2 \approx 4.2$ executions of κ^{split} for each seven-relation set.

This expectation is nearly met in the case of the sets that contain the hub; these sets all entail exactly 5 executions of κ^{split}. On the other hand, in the case of the sets that do not contain the hub, the observed κ^{split} execution counts fall in the range 10 to

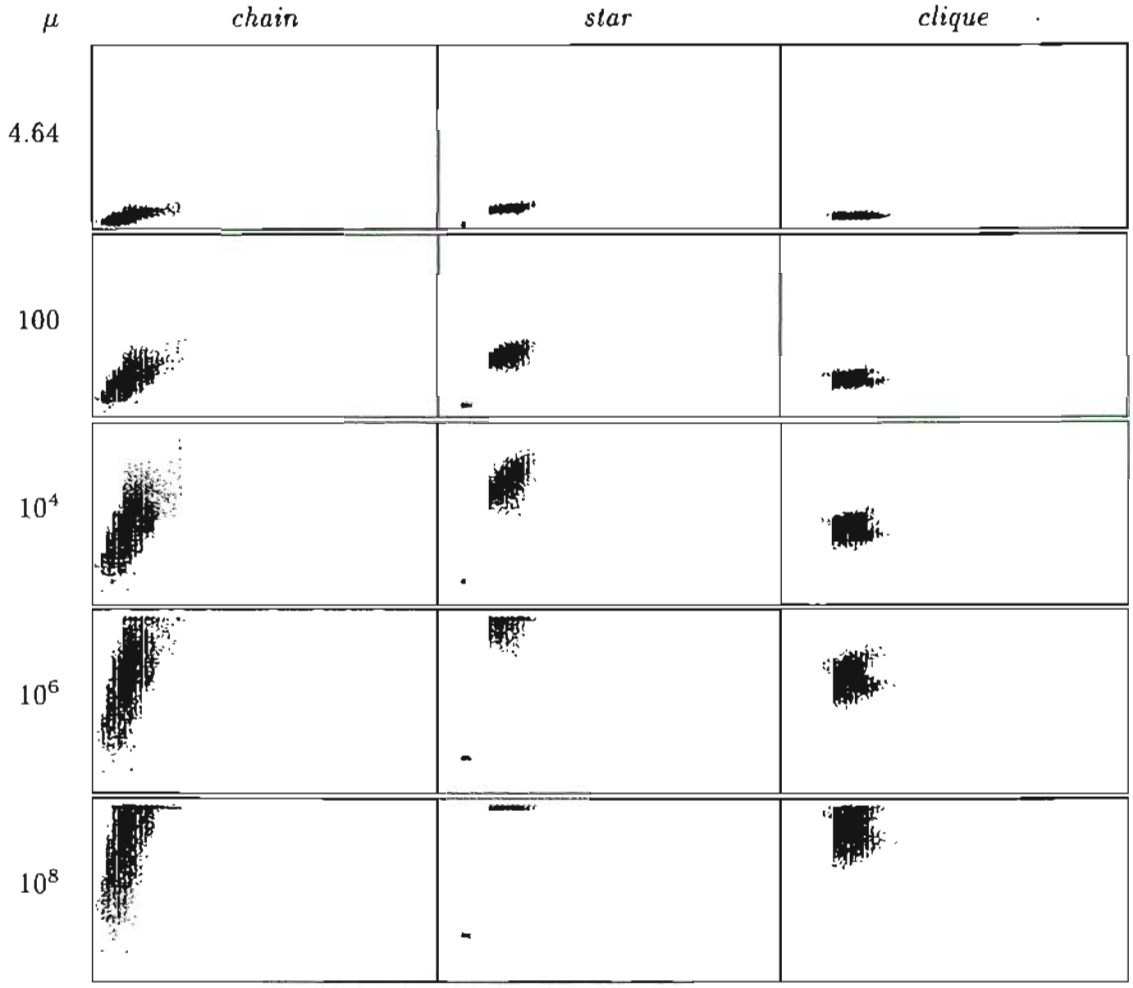
17—substantially above the predicted average. The explanation for this effect is not immediately obvious; in particular, *in the case of the naive cost model*, there is no apparent reason why the cardinality of a set should have anything to do with the number of executions of κ^{split}. (The situation is quite different in the case of the disk-nested-loops model, as we shall see presently.)

We attribute the discrepancy between the predicted and observed behavior to the fact that the alternative splits are not, in fact, examined in random order. It may be possible to reduce the discrepancy by changing the algorithm to examine the splits in a different (but still deterministic) order, or by permuting the names of the base relations R_0 through R_{14} so that their cardinalities are not necessarily in ascending order. However, investigation of these possibilities shall be left to the future.

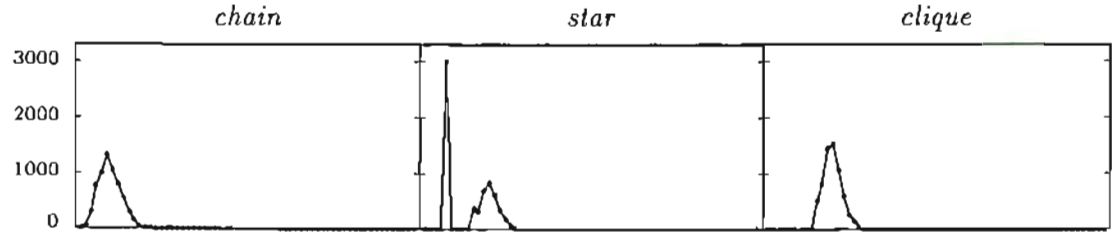
6.4 Fingerprints for Various Queries

Let us now see what we can learn about different kinds of queries by examining their fingerprints. Figure 6.3(a) shows the fingerprints obtained under the naive cost model from a variety of queries. All the queries represented are 15-way join problems, constructed according to the parameterization discussed in Section 6.1.2 above. Henceforth we shall refer to such queries as our *basic test queries*. The *variability* of the mean base-relation cardinalities is fixed at 0.5 for all the queries considered in Figure 6.3. There remain two parameters that vary: the join-graph topology, and the mean base-relation cardinality (denoted μ in the figure). The first, second, and third columns of plots depict, respectively, fingerprints for chain, star, and clique queries. (We omit the *cycle*+3 topology because its characteristics are very similar to those of the chain.) Plots in successive rows of the figure represent queries whose mean base-relation cardinalities range from 4.64 to 10^8. (We do not consider the effects of varying μ between 1 and 4.64. Behavior in this low range is likely to be interesting and complex (though fingerprints for queries with μ *exactly* equal to 1 are boring, one-point blobs); but for the present it may be just as well not to become embroiled in any more complexity than necessary.)

It is at once apparent that the fingerprints within each topology grouping have their



(a) As function of join-graph topology and mean base-relation cardinality



(b) Cumulative point densities at each κ^{split}-execution count (with $\mu = 10^8$)

Figure 6.3: Fingerprints under the naive cost model

own distinctive features. We already discussed above the general appearance of a star-query fingerprint, and we see here that all the star queries depicted in Figure 6.3 have the same character. Now let us make sense of the fingerprints for chains and cliques.

The chain-query fingerprints show that the join cardinalities of the seven-relation sets span a wide range, more or less continuously—certainly with no large gaps. But the points representing these sets become very sparse at both the low and high ends of the range. (The thinning at the upper end is obscured in the plots where large numbers of points jam up against the ceiling at 10^{38}, but still occurs in principle.)

Both the smoothness of the distribution and the thinning at the extremes can be understood in terms of the join graph. The seven-node subgraphs of a fifteen-node chain join graph may have anywhere from zero to six edges. Those with zero edges represent joins that are actually Cartesian products, just as in the case of the star queries; the join cardinalities of the corresponding seven-relation sets will generally be large. But whereas in the case the star query, roughly *half* the seven-node subgraphs lack edges, the same holds for only a small handful of subgraphs in the case of the chain. Likewise, whereas in the case of the star, roughly half the seven-node subgraphs have *six* edges, and hence, in general, low join cardinalities, in the case of the chain the seven-node subgraphs with six edges are again something of a rarity. Most of the subgraphs have between two and four edges, and yield cardinalities of intermediate but highly nonuniform magnitude.

The clique-query fingerprints are the most compact. Again an explanation is found in the join graph. Any seven-node subgraph of a clique is itself a clique; thus, these seven-node subgraphs are all topologically isomorphic. Consequently, any variability in the join cardinalities of the corresponding relation sets is due solely to variability in the values that annotate the join graph—i.e., the base-relation cardinalities and the predicate selectivities. In the queries under study, the variation in the base-relation cardinalities is significant—they range from $\sqrt{\mu}$ to μ^2—but the predicate selectivities are computed in such a way that they partially compensate for the variation in the base-relation cardinalities. The net effect is that the smallest join cardinalities for the seven-relation cliques are much larger than in the case of chains and stars, while the largest join cardinalities are much *smaller* for cliques than for chains and stars.

We noted in Section 6.3.2 above that in the case of the star query, there was an unexpected correlation between a point's x- and y-coordinates. Here we see that this correlation appears, to varying degrees, in the fingerprints for the chain and clique as well: in each instance, the point clusters slope upwards as x increases. However, while the correlation is in evidence *within* each fingerprint, there appears to be no similar correlation *across* fingerprints. That is, the successive fingerprints for $\mu = 4.64, 100, 10^4, \ldots$, show ever-increasing join cardinalities for the seven-relation sets, as evidenced by the increasing upward reach of the point clusters; but these clusters show no corresponding horizontal movement. We conjectured in Section 6.3.2 that the correlation seen inside an individual fingerprint was due to obscure details of our implementation, and not to any intrinsic difficulty in finding the best split for a set with a large join cardinality. The present evidence supports this conjecture, inasmuch as large cardinalities *per se* are not associated with high execution counts for κ^{split}.

Thus far we have not ruled out the possibility that there is *hidden* horizontal movement from one fingerprint to the next. One can imagine the center of mass of the points migrating slowly to the right with increasing μ, even as the envelope of the points remains stationary. To eliminate this possibility, one may plot the point densities in the manner illustrated in Figure 6.3(b). The graphs in this portion of the figure show, for the case $\mu = 10^8$, the *number* of fingerprint points lying along the vertical line determined by each x-value. The peak of the point densities for the chain query lies at $x = 6$; for the clique query, at $x = 13$; and there are *two* peaks for the star query—one at $x = 5$ (the position of the blotch), and another at $x = 13$ (the mode of the points representing hubless sets of relations). The same peaks are obtained when one plots the corresponding graphs for the other values of μ; indeed, the graphs for the different μ are so similar as to be virtually indistinguishable. Because they add no new information or insight, we omit the point-density graphs for $\mu = 4.64$ through 10^6.

6.5 Execution Counts under the Nested-Loops Model

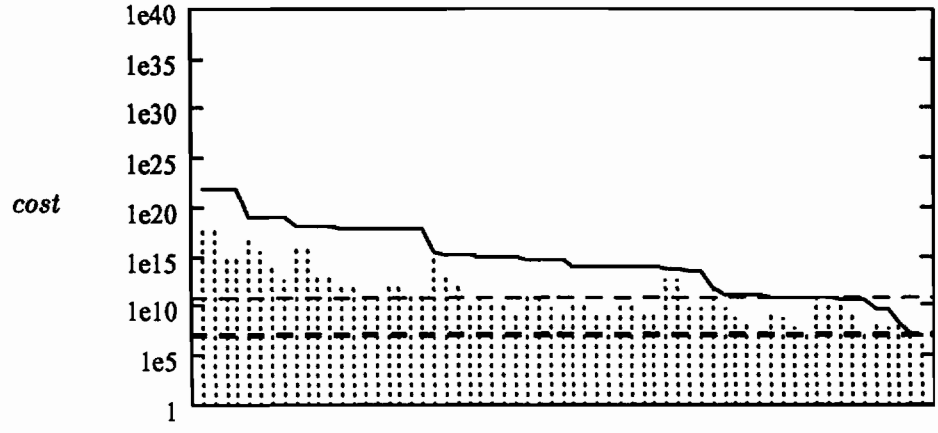
From the foregoing analysis under the naive cost model, we now have a baseline for further fingerprint studies. The naive-cost-model fingerprints showed the ways in which the behavior of our implementation departs from the ideal behavior predicted by our theoretical model. We found that even when $\kappa^{split} \equiv 0$, the number of executions of κ^{split} was somewhat larger than expected. In the case where $\kappa^{split} \not\equiv 0$, we have no quantitative expectation—only the expectation that the execution counts for κ^{split} should increase further still. Here we present empirical results on the extent of that increase under the disk-nested-loops cost model.

6.5.1 Split-Graphs

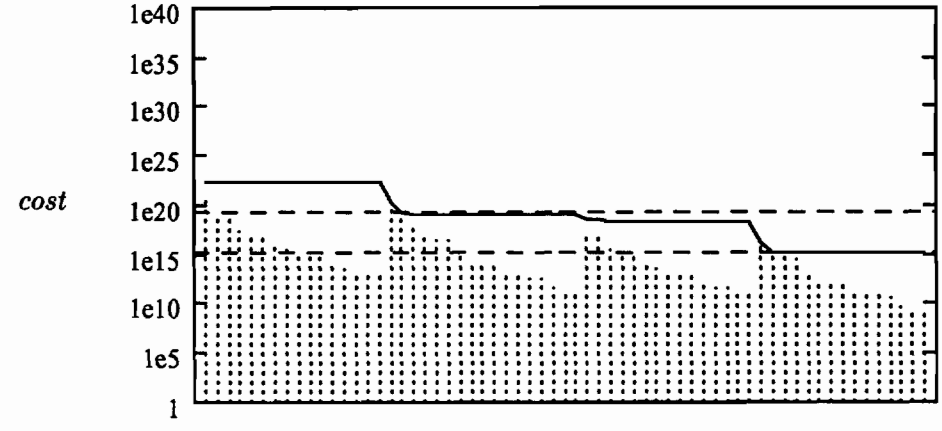
Before discussing fingerprints under the disk-nested-loops cost model, we introduce one additional visual aid, to which we give the name *split-graph*. Figure 6.4 illustrates two such graphs. A split-graph may be thought of as an exploded view of a single dot in a fingerprint plot. That is, each dot in the fingerprint represents a seven-relation set; the split-graph for such a set gives information that helps to explain why the dot for that set lies where it does in the fingerprint.

Both the split-graphs in Figure 6.4 pertain to the optimization under the disk-nested-loops model of the 15-way chain query with $\mu = 10^4$ and variability 0.5. (These particulars are not important at this stage, and are given only for completeness.) Figure 6.4(a) gives information about the set $\{R_0, R_1, R_3, R_5, R_8, R_{12}, R_{13}\}$, and Figure 6.4(b) gives information about the set $\{R_0, R_2, R_6, R_7, R_8, R_9, R_{11}\}$. The captions of the two graphs refer to these sets as "easy" and "hard," respectively, for reasons that will be explained in Section 6.5.2 below.

To interpret the information in the graphs, recall that there are $2^7 - 2 = 126$ possible splits of a seven-relation set; but since we are working with a symmetric cost model, any split $(\mathcal{U}, \mathcal{V})$ is equivalent—where cost is concerned—to its mirror image $(\mathcal{V}, \mathcal{U})$. Hence there are 63 distinct splits that *find_best_split* must consider when it is invoked on either of the sets under discussion (or, indeed, on any other seven-relation set). Each of these 63



(a) Costs of all splits of the "easy" set $\{R_0, R_1, R_3, R_5, R_8, R_{12}, R_{13}\}$



(b) Costs of all splits of the "hard" set $\{R_0, R_2, R_6, R_7, R_8, R_9, R_{11}\}$

Figure 6.4: Split-graphs for a chain query with $\mu = 10^4$ and variability 0.5 under the disk-nested-loops cost model

splits is represented in the graphs of Figure 6.4 at a separate position on the x-axis.

Now, *find_best_split* goes through several steps in considering the viability of a particular split of a set $\mathcal{S}$ into two sets $\mathcal{S}_{lhs}$ and $\mathcal{S}_{rhs}$ (*cf.* Figure 3.2, page 76). First, the costs of $\mathcal{S}_{lhs}$ and $\mathcal{S}_{rhs}$ are added to obtain *operand_cost*, and then $\kappa^{split}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs})$ is added to *operand_cost* to give *dependent_cost*. If *dependent_cost* is less than the current *best_cost_so_far*, then *dependent_cost* becomes the new value of *best_cost_so_far*. Thus, it

is the value of *dependent_cost* that determines how each split stacks up against the others. The splits in each graph in Figure 6.4 are arranged in order of decreasing *dependent_cost*, and the solid curve in the graphs plots the descent of *dependent_cost* as one moves from the poorest split to the best split for the set. (Note that the order in which the splits appear in the split-graphs has nothing to do with the order in which they are examined by *find_best_split*. Note also that splits whose *dependent_cost* values are equal are ordered arbitrarily.)

It is the value of *dependent_cost* that ultimately determines the viability of a split, but the value of *operand_cost* also holds considerable interest: Whenever *operand_cost by itself* exceeds *best_cost_so_far*, there is no need to go to the trouble of computing $\kappa^{split}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs})$ (*cf.* Section 3.4.3, page 86). But how often, one may ask, does *operand_cost* exceed *best_cost_so_far*? The graphs in Figure 6.4 take a step towards answering this question by showing the values of *operand_cost* for each split, as well as the values of *dependent_cost*. The values of *operand_cost* appear as vertical dotted spikes that reach up towards the curve depicting *dependent_cost*.

The lower, "filled-in" portion of each of the graphs thus represents operand costs, while the white space sandwiched between the *operand_cost* and *dependent_cost* values represents the difference between the two—in other words, the white space represents the contribution of κ^{split}.

Note, though, that it is easy to be deceived about the quantities represented by the white space. In some split-graphs there is only a narrow band of white space, and the area it occupies is small in comparison with that of the filled-in space below it. In most such cases, one would be mistaken in inferring that *dependent_cost* is chiefly determined by *operand_cost*, and that κ^{split} makes only a small contribution towards the total. Because the vertical axis of the graphs is scaled logarithmically, the linear distance representing the κ^{split} contributions is drastically compressed; in most of the graphs we will examine, the contribution of κ^{split} actually vastly overshadows that of *operand_cost*, and yet appears visually as the smaller of the two.

In this sense, the split-graphs are misleading. But however deceptive they may be *numerically*, they tell the truth about the relative significance of *operand_cost* and κ^{split}

in optimization. We shall see later that in some situations, the numerical value of κ^{split} makes no difference whatsoever; and in those situations where it does make a difference, it generally matters only inasmuch as it exceeds *operand_cost* by at least an order of magnitude.

One final descriptive matter: The horizontal, dashed lines in the graphs show the successive values taken on by *best_cost_so_far* during *find_best_split*'s processing of the sets in question. There are three such lines in Figure 6.4(a), and seven in Figure 6.4(b); but because some of these lines fall so close to one another, they are not all distinguishable. What appear to be thick, dashed, horizontal lines are actually clumps of several thinner lines.

6.5.2 Split-Graph Shape and Cost-Function Execution Count

As noted above, we refer to the set depicted in Figure 6.4(a) as an "easy" set, and to the set in Figure 6.4(b) as a "hard" set. The reason for these designations derives from the number of executions of κ^{split} in the processing of each of the sets: *five* executions for the "easy" set, and *forty-five* executions for the "hard" set.

This difference in the number of executions of κ^{split} is extreme, but not surprising in light of the relative contributions of the *operand_cost* values in the two cases. In the case of Figure 6.4(a), by the time the second value of *best_cost_so_far* has been set, the *operand_cost* values for nearly all splits protrude above the level of *best_cost_so_far*. Consequently, there is no need to evaluate κ^{split} for these splits—they can be thrown out on the basis of their *operand_cost* value.

By contrast, in Figure 6.4(b), most of the *operand_cost* values do not protrude above the level of *best_cost_so_far* even after *best_cost_so_far* has been reduced to its final, lowest value. As a result, the loop in *find_best_split* can eliminate a split on the basis of its *operand_cost* value in only a handful of instances. In all the remaining instances, evaluation of κ^{split} is required to determine that the split is not competitive.

There appears to be a component of "luck" in the fact that only five κ^{split} executions are needed to process the set of Figure 6.4(a). Perhaps the word *luck* does not quite apply, since our algorithm is deterministic and will always process the given set in the

same way. But one can well imagine that if the splits were processed in a different order, the execution count for κ^{split} might turn out to be somewhat larger. Even so, the odds that it would be *much* larger are slim. We will find that while there is no guarantee of a low κ^{split}-execution count in a set whose split-graph has the appearance of Figure 6.4(a), the preponderance of such sets nonetheless do yield very low κ^{split}-execution counts.

Here, then, we see a striking effect. It would not be surprising to find that evaluation of κ^{split} could generally be avoided when the contribution of κ^{split} was small compared to that of *operand_cost*; for then the situation would be only marginally different from the situation that obtains under the naive cost model, where κ^{split} contributes nothing at all. But in Figure 6.4(a) we have values of κ^{split} that exceed the corresponding values of *operand_cost* typically by *five orders of magnitude*—and yet, for the most part, these monstrous quantities turn out to be irrelevant.

Evidently the crucial feature of Figure 6.4(a) that leads to a low execution count for κ^{split} is simply this: *Most* of the values of *operand_cost* are larger than *some* of the values of *dependent_cost*. Seen in this light, the numerical values contributed by κ^{split} do indeed appear to be beside the point, except in a few instances. Specifically, it is important that there exist several splits for which *operand_cost* and κ^{split} are *both* small in comparison with the *operand_cost* values encountered in most of the other splits. The existence of several such relatively low-cost splits makes it likely that in the processing of the set, *best_cost_so_far* will quickly fall to a level that undercuts the bulk of the *operand_cost* values.

In Figure 6.4(b), it is precisely the absence of low-cost splits, in the sense just described, that makes it necessary to evaluate κ^{split} for nearly all splits. The smallest values of κ^{split} are huge (that is, *really* huge), and consequently the smallest values of *dependent_cost* are also huge. Most of the values of *operand_cost* are very small by comparison.

We may go one step further and observe that the existence of low-cost splits for a set $\mathcal{S}$ is not entirely accidental, but is associated with another property of sets of relations that we have remarked on before. Consider again the induced join graph for $\mathcal{S}$. When this graph contains many edges, there are many ways the edges can be apportioned when $\mathcal{S}$ is split into two sets $\mathcal{S}_{lhs}$ and $\mathcal{S}_{rhs}$. That is, some edges may end up belonging to $\mathcal{S}_{lhs}$,

and some to $\mathcal{S}_{rhs}$, while those that belong to neither will furnish join predicates for the join of $\mathcal{S}_{lhs}$ and $\mathcal{S}_{rhs}$. As it happens, the set depicted in Figure 6.4(a) induces a join graph with a relatively large number of edges. The resultant variations in predicate assignments in the splits of this set are reflected in the gradual, sloping descent of *dependent_cost* in Figure 6.4(a).

By contrast, if the join graph for $\mathcal{S}$ has no edges, or few edges, there are few ways to apportion the edges in the splits of $\mathcal{S}$. Accordingly, the *dependent_cost* curve for such sets characteristically consists of a relatively small number of discrete, nearly flat steps, connected by abrupt transitions, as in Figure 6.4(b).

This contrast has another manifestation as well. When a set $\mathcal{S}$ induces a join-graph with many edges, as in Figure 6.4(a), the associated join cardinality will generally be relatively small. When the join-graph has few edges, as in Figure 6.4(b), one may expect to encounter larger join cardinalities. Thus, sets that appear low down in a fingerprint plot will tend to have "easy" split-graphs, while sets that appear higher up will tend to have "hard" split-graphs.

6.6 Fingerprints under the Nested-Loops Model

We now examine fingerprints under the disk-nested-loops cost model. In particular, we shall observe the evolution of the fingerprints under increasing mean base-relation cardinalities. As the fingerprints evolve, the dots representing individual sets will be seen to migrate in ways that are tied to the split-graphs for those sets. The patterns that emerge from these observations will help to explain the peculiar bulges in the performance graphs of Figure 6.1.

We begin by considering chain queries. The left-hand column of Figure 6.5 shows fingerprints of 15-way chain queries under the disk-nested-loops model. The queries shown are taken from our repertoire of basic test queries, with the mean base-relation cardinality μ ranging from 4.64 to 10^8, and with the variability held fixed at 0.5.

The middle and right-hand columns of the figure show split-graphs that correspond to the fingerprints in the left-hand column. The sequence of split-graphs in the middle

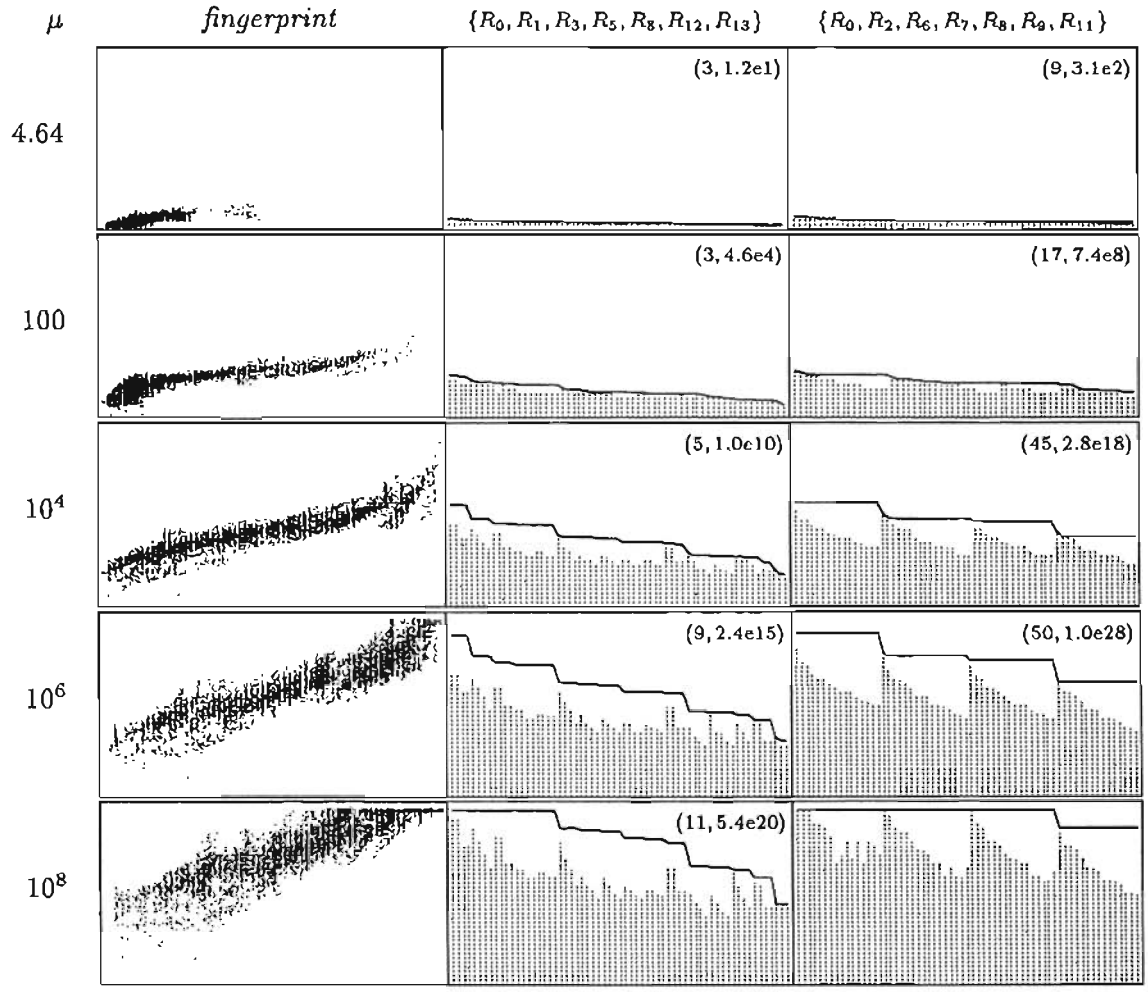


Figure 6.5: Chain-query fingerprints and split-graphs for various μ

column traces the trajectory of an arbitrarily chosen "easy" set, while the sequence in the right-hand column traces the trajectory of an arbitrarily chosen "hard" set.

(The "easy" set is $\{R_0, R_1, R_3, R_5, R_8, R_{12}, R_{13}\}$; in accordance with the *chain* topology described in Appendix C, the induced join graph of this set has four predicate edges, which connect the following pairs of relations: R_0–R_8, R_1–R_8, R_5–R_{12}, and R_5–R_{13}. The "hard" set is $\{R_0, R_2, R_6, R_7, R_8, R_9, R_{11}\}$. The induced join graph of this set has two predicate edges, which form the connections R_0–R_8 and R_2–R_9.)

As noted previously, each split-graph corresponds to an *individual point* in the corresponding fingerprint. The coordinates in the upper right-hand corner of each split-graph show the positions of those points in the adjacent fingerprint.

Let us now examine the sequence of fingerprints; we will come back to the split-graphs in a moment. We see that as cardinality increases, the fingerprint drifts upwards—but it also spreads out. The fingerprint's center of mass moves farther and farther to the right, reflecting increasing numbers of executions of κ^{split}. These increasing execution counts correspond to the rise in optimization time observed as mean cardinality ranges up to about 10^4 in the $\kappa_{dnl}/chain$ cell of Figure 6.1.

As cardinality rises above 10^4, the rightward drift of the cloud of points in the fingerprint sequence continues unabated, yet Figure 6.1 shows that optimization time starts to drop off in this region. This drop is due to a separate effect, also evident in the fingerprint sequence. With the larger cardinalities, we encounter increasing numbers of sets $\mathcal{S}$ for which the split-independent cost $\kappa^{out}(\mathcal{S})$ rises above 10^{38} and overflows the single-precision floating-point representation that we use for plan costs. As explained in Chapter 4, we skip the loop in *find_best_split* when we encounter such sets. Thus, we start to obtain substantial savings when there are many such sets.

Sets whose split-independent cost overflows appear in the fingerprints as points that are jammed up against the upper boundary of the plot. They are shown as having y-coordinates of 10^{38}, though their true y-coordinates may be much higher. Their x-coordinates show the number of executions of κ^{split} that *would be* required if costs were maintained in double-precision floating-point, and the loop in *find_best_split* were executed for these sets as for all other sets. But since that loop is in fact skipped for the sets in question, the actual number of executions of κ^{split} for these sets is zero. To visualize their effect on optimization time, one can imagine the points representing these sets as simply being removed from the fingerprint plots once their split-independent cost overflows.

In the sequence of fingerprints in Figure 6.5, the dot representing a particular set may trace a trajectory that begins in the lower left-hand corner of the plot, then gradually moves upward and to the right until finally the dot jams up against the top of the plot. The effort involved in processing the set increases with the dot's motion toward the right,

but when the dot bumps into the upper boundary, the effort abruptly drops back to zero. The abruptness of this drop is not evident in Figure 6.1, because the effect of any individual set on optimization time is small. But as more and more sets are gradually removed from consideration, the cumulative effect becomes large.

6.6.1 Trajectories as Seen through Split-Graphs

The trajectories of the dots representing different sets are plainly not all alike. It is apparent from the gradual dispersion of the dots that some of them are drifting rightward far more quickly than others.

The reason for the variation in drift rates can be understood in terms of split-graphs. Though "easy" and "hard" are relative terms, these classifications are useful in explaining drift rates. The dot for the "easy" set depicted in the middle column of Figure 6.5 drifts slowly—its x-coordinate successively take on the values 3, 3, 5, 9, and 11 as the mean base-relation cardinality runs from 4.64 to 10^8. The x-coordinate for the "hard" set depicted in the right-hand column, by contrast, progresses rapidly through the values 9, 17, 45, 50, and 55.

It is interesting to note that in both the sequences of split-graphs, the values of κ^{split} increase much more quickly than the *operand_cost* values (i.e., the band of whitespace occupies an increasingly large proportion of the area under the *dependent_cost* curve). There appears to be some difference between the "easy" set and the "hard" set in the rate of growth of the whitespace, but not enough of a difference to account for the divergence of the execution counts in the two cases.

Instead, the effect at work is that "easy" sets tend to remain "easy" as cardinalities increase: in these sets, as noted above, there exist splits for which both *dependent_cost* and κ^{split} are relatively small. The splits that possess this trait tend to retain the trait even when the cardinalities increase. The split-graphs for the "hard" set, on the other hand, show a gap at the right-hand edge between the *dependent_cost* curve and the *operand_cost* spikes below it. At low cardinalities, this gap is not readily discernible, but it is still present. In other words, the "hard" set does not fundamentally change its character with the changing cardinalities. It always possesses the character of a "hard" set, only the

consequences are less severe at lower cardinalities. As long as the whitespace at the right-hand edge of the split-graph is narrow compared to the range spanned by the *operand_cost* values, one can expect to obtain relatively low κ^{split}-execution counts.

6.6.2 Behavior of Star Queries

We shall not give detailed consideration to the "*cycle*+3" topology, because its characteristics are very similar to those of the *chain* topology. But the optimization of star queries under the disk-nested-loops cost model reveals interesting differences from what we saw in the case of the chain. We shall now analyze the star in the same manner that we just analyzed the chain.

The surface in the $\kappa_{dnl}/star$ cell of Figure 6.1 differs only subtly from the $\kappa_{dnl}/chain$ cell to its left. Both have qualitatively the same shape; in both cases, as cardinality rises, optimization time first increases, and then later drops off again. But in the star-query case the effects are more compressed and more exaggerated.

These differences are reflected in the star-query fingerprints in Figure 6.6. About half the points in these fingerprints—those constituting the blotch in the lower-left hand corner—remain fairly stationary throughout the sequence. But the remaining mass of points moves quickly to the right as cardinality rises, accounting for the corresponding rapid increase in optimization time. At the same time, this mass of points also moves upward rather quickly; before long, large numbers of these points are jammed up against the upper boundary of the fingerprint. Once again, the accumulation of points at this boundary corresponds to a reduction in optimization time as the loop in *find_best_split* is skipped for larger and larger numbers of sets.

Like the corresponding figure for the chain-query case, Figure 6.6 includes two columns of split-graphs alongside the fingerprints. As before, the middle column traces the trajectory of an arbitrarily chosen "easy" set, and the right-hand column trace the trajectory of an arbitrarily chosen "hard" set. The fingerprint dot for the "easy" set remains within the blotch at the lower-left corner throughout the fingerprint sequence. The dot for the "hard" set belongs to the mass of dots that moves upward and rightward.

(Here, the "easy" set is $\{R_0, R_3, R_4, R_6, R_7, R_{12}, R_{14}\}$. In accordance with the star

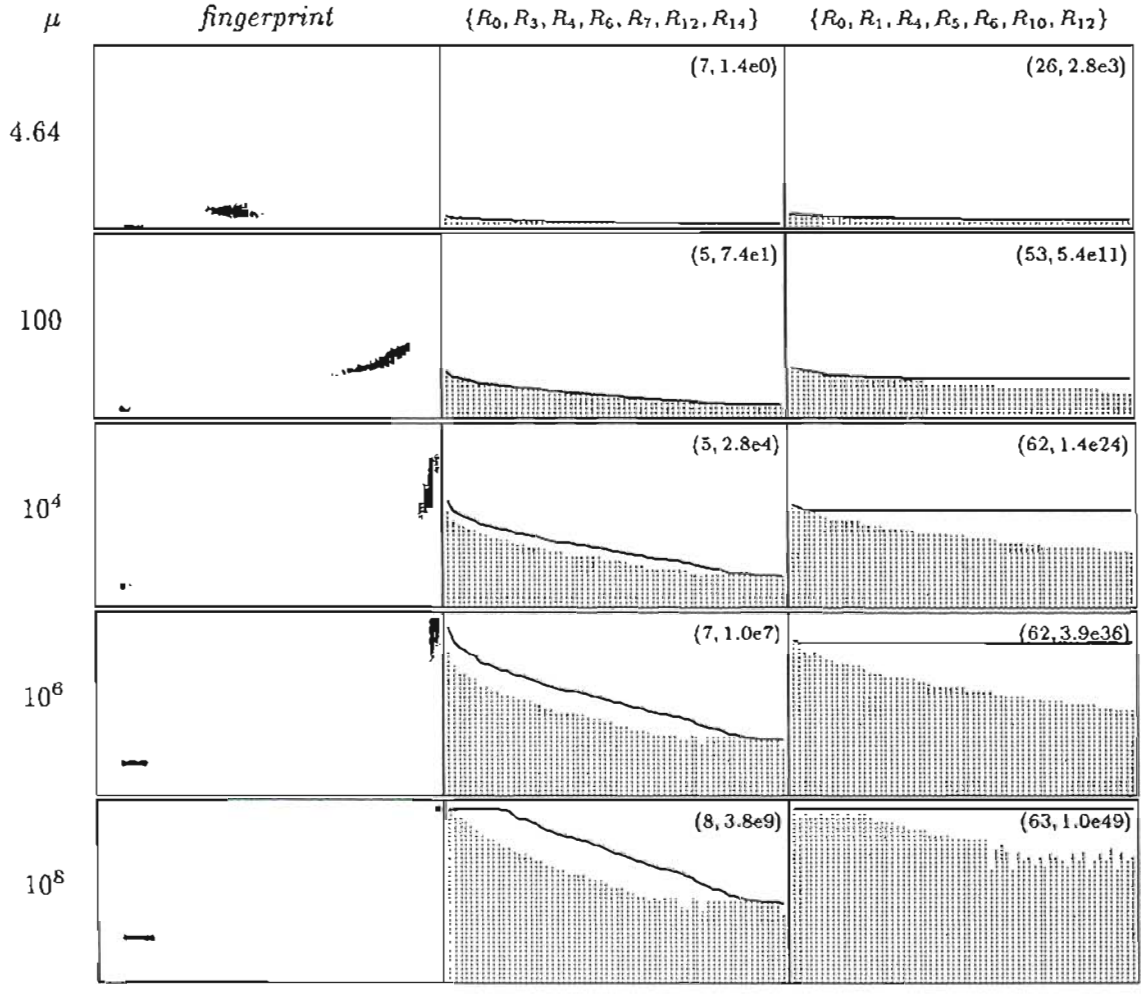


Figure 6.6: Star-query fingerprints and split-graphs for various μ

topology described in Appendix C, its induced join graph has predicate edges between R_{14} and each of the other relations in the set—i.e., six predicates altogether. The "hard" set is $\{R_0, R_1, R_4, R_5, R_6, R_{10}, R_{12}\}$, whose induced join graph has no predicate edges at all.)

In the case of the chain, the distinction between "easy" and "hard" sets was one of degree, but here it is absolute. Sets that contain the hub of the star are "easy"; the rest are "hard." The "easy" sets all induce join graphs with six predicates—a sufficient number that the split-graphs for such sets show a *dependent_cost* curve that descends

very smoothly and gradually, in the manner seen in the middle column of Figure 6.6. The "hard" sets induce join graphs with *no* predicates; the resultant split-graphs in the right-hand column of Figure 6.6 show *dependent_cost* curves that are essentially one flat plateau except near the left-hand edge.

In light of the appearance of these split-graphs, the behavior of the star-query fingerprints is entirely unsurprising. In the "easy" graphs, the lowest *dependent_cost* values undercut nearly all the *operand_cost* values; in the "hard" graphs, the lowest *dependent_cost* values undercut almost none of the *operand_cost* values.

One feature of the "easy" split-graphs deserves comment. The band of whitespace in these graphs grows to be quite wide at the larger cardinalities, but the gap between the *dependent_cost* curve and the *operand_cost* values always closes near the right-hand edge of each graph. The reason for this closing of the gap is that near the right-hand edge of the graph, one encounters the *left-deep* splits—the splits in which the right-hand side consists of a single relation.[3] Given a star join-graph topology, it is only in the left-deep splits that *both* the left-hand side and right-hand side are free of Cartesian products. Consequently, the products of the input cardinalities are lowest for the joins formed from the left-deep splits; and in our test queries, the total cost of such a join can be expected to be roughly comparable to the total cost of the join at its left input, since both these joins involve cardinalities of similar magnitude.

6.6.3 Behavior of Clique Queries

Figure 6.1 showed that the optimization times for clique queries are elevated relative to the times for chains and stars. However, even in the case of cliques there was a hint of the same pattern exhibited by the other topologies—an increase in optimization time as cardinality rises from low to moderate, and then a drop as cardinality rises still further, from moderate to high.

The fingerprints and split-graphs in Figure 6.7 give another perspective on the difficult

[3]Some of the splits in question are technically *right-deep* rather than left-deep. But since we are assuming symmetrical join costs, we need not make a distinction between left-deep and right-deep. We shall maintain the convention of referring to splits of either kind as *left-deep*.

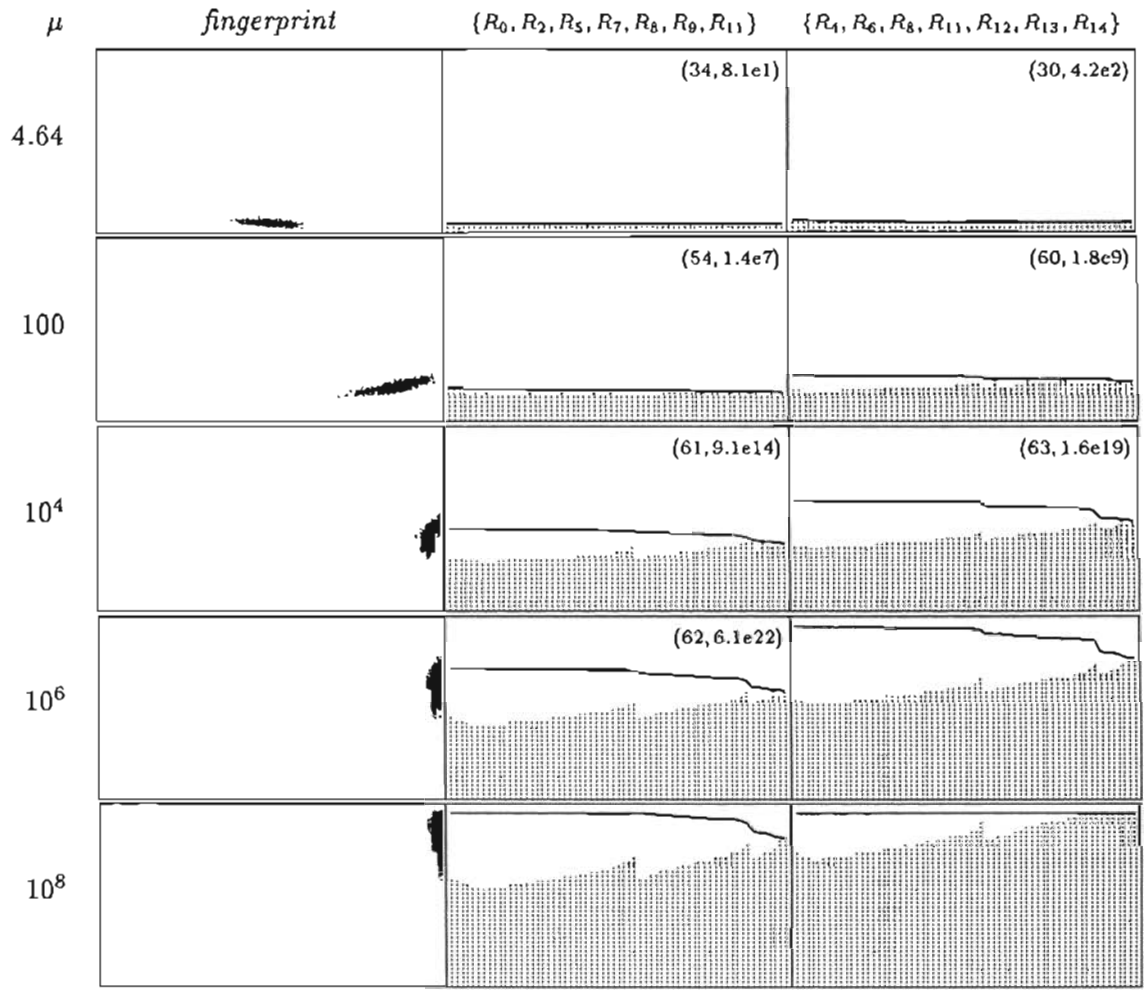


Figure 6.7: Clique-query fingerprints and split-graphs for various μ

character of the clique queries. As before, the figure shows two columns of split-graphs, for the sets $\{R_0, R_2, R_5, R_7, R_8, R_9, R_{11}\}$ and $\{R_4, R_6, R_8, R_{11}, R_{12}, R_{13}, R_{14}\}$, respectively. In this instance, there is little justification for calling one of the sets "easy" and the other "hard." One might more accurately characterize them as being "horrible" and "even more horrible." In the case of the clique, there *are* no "easy" sets.

(Both the sets represented in these split-graphs induce join graphs that are cliques, as do all subsets in a clique topology; but one difference between the two sets represented here is that the "even more horrible" set contains relations of larger cardinalities. As

explained in Appendix C, in our basic test queries, the relation R_0 is assigned the lowest cardinality of any relation, and the cardinalities ascend with the relation indices, so that R_{14} has the highest cardinality.)

In fact, the clique behavior exhibited in Figure 6.7 is in several respects qualitatively different from the behavior we have observed for chains and stars:

- As just noted, in the clique there are apparently no "easy" sets. All points in its fingerprint move in more or less the same way as cardinality increases.
- In the split-graphs for the clique query, low *operand_cost* values correlate with high *dependent_cost* values—just the opposite of what occurs in the chain- and star-query split-graphs.
- At no stage in the clique-fingerprint evolution do we find points jamming up against the upper boundary of the fingerprint plot. Yet Figure 6.1 does show a pronounced drop in clique-query optimization time at high cardinalities.

Presumably all the seven-relation sets behave more or less similarly because they are all topologically isomorphic; i.e., the join subgraphs they induce are all subcliques. What little difference one does find among the different sets is due to variations in the base-relation cardinalities and predicate selectivities. These variations are moderate in our test queries, and as we have seen before, it is the larger differences related to the presence or absence of predicate connections that distinguish "easy" from "hard" sets.

These comments explain why all the sets behave alike; but why do all the sets exhibit "hard," rather than "easy" behavior? The answer to this question is bound up with the negative correlation between the *operand_cost* and *dependent_cost* values in the split-graph for any given set. In the case of the chain and the star, high values for *dependent_cost* and *operand_cost* both tended to be associated with the presence of Cartesian products in one or both operands; hence these values were positively correlated. Here, Cartesian products are not an issue. Instead, high values for *dependent_cost* are associated with splits that assign roughly equal numbers of relations to the left-hand side and the right-hand side: such splits yield joins that have a large number of join predicates, and hence the product

of the operand cardinalities is large in proportion to the result cardinality. This large product appears as a term in the cost function κ_{dnl}^{split}, and thereby pushes up the value of *dependent_cost*. But these same splits, in which the numbers of relations on the left and right are roughly balanced, are the ones that give the lowest *operand_cost* values. When one operand contains more relations that the other, its cost is significantly higher, and drives up the sum of the operand costs.

Finally, we turn to the question of why the drop in optimization times at high cardinalities is not reflected in our fingerprints. The explanation is simple: our fingerprints represent only the *seven*-relation sets. Fingerprints of the eight-, nine-, and ten-relation sets would show point clusters at higher and higher levels, and the larger the sets, the more points would be jammed up against the upper boundary of the plot. In fact, the same is true in the case of the chain and star queries—though we did see points accumulating at the upper boundary even in the seven-relation fingerprints, the effect would have been stronger had we examined the behavior of slightly larger sets. Note, however, that the influence of the sets that are *much* larger—e.g., those of thirteen, fourteen, and fifteen relations—is small, because as noted earlier, the number of such sets is small.

6.7 Summary and Discussion

In this chapter we have attempted a systematic exploration of the performance characteristics of the Blitzsplit algorithm. Such an exploration presents challenges because there is no standard assessment procedure for join-order optimization.

We began by defining a parameterization for a suite of basic test queries that we submitted to the Blitzsplit algorithm for optimization. Using relatively simple cost models, we obtained timings that are very favorable compared to those that have been reported for other methods. Moreover, we gave evidence that our worst-case timings occur only under rare conditions, and that for a broad spectrum of queries the optimization time falls well below the worst case. At the same time, our performance graphs revealed sensitivity to the cost model, and displayed mysterious bulges for which there was no immediate explanation.

We then sought to understand the algorithm's performance in more detail, especially with a view to anticipating what kind of effects we would encounter under more computation-intensive cost models. To that end, we investigated the aspects of a query that affected the number of executions of the split-dependent component κ^{split} of the cost function κ—for it is this component of cost to which the algorithm will inevitably show the greatest sensitivity.

We introduced query *fingerprints* and *split-graphs* for the purpose of probing the κ^{split}-execution counts. We found that the somewhat mysterious patterns in Figure 6.1 are not so mysterious after all, but have straightforward explanations in terms of our fingerprint plots and split-graphs.

One can extrapolate from these explanations and begin to make some predictions as well. For example, one might predict that redistributing the selectivities among the predicates (preserving the product of these selectivities) ought not to change the overall shapes of the fingerprints, but would presumably cause some additional spreading of the fingerprint dots along the vertical axis. It is therefore not surprising that in our *ad hoc* experiments with altered selectivities, we observed performance very similar to that of our basic test queries. One can imagine going further, and attempting to *quantify* the effects of such changes by constructing analytical models. However, such a modeling effort is beyond the ambitions of the present work, and will have to be left to the future.

The applicability of some of our observations to actual query processing may be questioned. In particular, the relevance of the disk-nested-loops cost model is suspect, since, aside from being fairly simplistic, it assigns prohibitively high costs to queries involving large cardinalities. One could not possibly hope to execute our basic test queries at large cardinalities if the execution costs were in fact those given by the disk-nested-loops model.

However, this deficiency in the disk-nested-loops model does not necessarily compromise the utility of our measurements and analysis. On the contrary, it is a reasonable conjecture that the effects observed under this model illustrate the worst-case behavior of our algorithm with regard to the κ^{split}-execution counts. Cost models that do not include a term proportional to the product of the join-input cardinalities can be expected to yield split-graphs with narrow bands of whitespace. In cost models that *do* include such a term,

the band of whitespace will become wide, sooner or later, as cardinality increases.

More sophisticated cost models create a synthesis of these two alternatives, giving consideration to nested-loops strategies where they are feasible, and avoiding them elsewhere. One may expect the behavioral characteristics of such a synthesis to be intermediate between those of our naive model and those of our disk-nested-loops model. By focusing separately on two distinct potential sources of cost, these simplistic models bracket the range of behaviors one would be likely to encounter in realistic cost models.

Chapter 7

Pruning Cost Computations

We now consider further improvements to the Blitzsplit algorithm based on the observations of the previous chapter. To motivate these improvements, let us briefly review both what is good and what is bad about the algorithm's performance, as shown by our observations.

What is good is its sheer speed under simple cost models. With the Blitzsplit algorithm, we obtained optimization times that, even in the worst case, were lower by several orders of magnitude than those that have been reported for other methods.

But one must be cautious in interpreting these timings. Earlier studies of join-optimization performance suggest that when heavier-weight join-enumeration strategies are used, the effort of join enumeration is the limiting factor in join-optimization performance [40, 45]. Under those conditions, one can equally well measure performance using a simple cost computation or a more complicated one—the effort involved in the cost computations is dominated by the join-enumeration effort in either case. However, once the join-enumeration effort is pared down to almost nothing, as the Blitzsplit algorithm has done, the tables are turned. Cost computations emerge as a significant and possibly the dominant component of optimization time.

Seen in this light, what is bad about our algorithm's performance is that it does *not* do a good job of economizing on the *number* of executions of the cost function κ. Assuming, as we have done, that the computation of κ is separated into a split-independent component κ^{out} and a split-dependent component κ^{split}, we have seen that we need to be especially concerned about the number of executions of κ^{split}. Even under the idealized assumptions of Sections 3.4.2 and 3.4.3, the κ^{split}-execution count does not necessarily

compare favorably with the cost-function execution counts that would be obtained with algorithms that consider only "feasible" sets of relations in the manner of Starburst or Volcano. For when the join-graph topology is a chain, for example, the complexity of optimization in Starburst or Volcano is *polynomial* in the number of relations n; whereas our algorithm necessarily has *exponential* complexity—and in particular, gives a κ^{split}-execution count of $(\ln 2/2)n2^n$ according to our idealized analytical estimate.

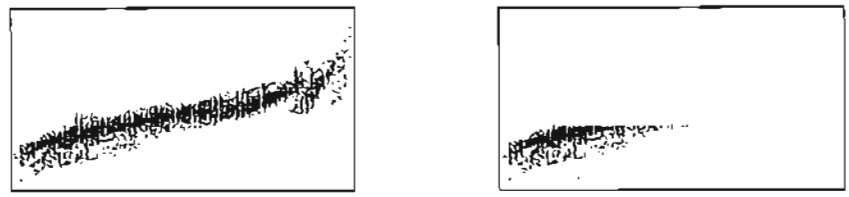
Moreover, the fingerprint studies of the previous chapter show how far the actual κ^{split}-execution counts can diverge from the ideal. In terms of κ^{split}-execution counts, our algorithm appears to be paying a heavy price for including consideration of Cartesian products.

At the same time, the fingerprint studies suggest a way of avoiding the penalty associated with Cartesian products. Except in the case of the clique—whose difficulty has nothing to do with Cartesian products in the first place—the points that lie furthest to the right in the fingerprint plots also tend to be found high up in the plots. The techniques described below take advantage of this fact to cut the κ^{split}-execution counts, sometimes drastically.

7.1 Pruning by Plan-Cost Thresholds

As a first measure to avoid the penalty incurred by Cartesian products, we apply a simple pruning technique that we refer to as *pruning by plan-cost thresholds.* The idea behind this technique is simply to remove from the fingerprints all points above a particular threshold.

This technique is implemented by means of a small change to the code in *find_best_split* (*cf.* page 76). Recall that κ^{out} represents the component of a cost function κ that depends only on the set $\mathcal{S}$ of relations whose join is to be optimized, and not on any particular split of $\mathcal{S}$ into left- and right-hand sets of relations. Consequently, $\kappa^{out}(\mathcal{S})$ can be computed for the set $\mathcal{S}$, once and for all, *outside* the inner loop that examines the individual splits of $\mathcal{S}$. As described in Section 4.8, our actual implementation of the Blitzsplit algorithm entirely skips the inner loop in *find_best_split* whenever $\kappa^{out}(\mathcal{S})$ threatens overflow, i.e., whenever $\kappa^{out}(\mathcal{S})$ exceeds 10^{35}. But the threshold 10^{35} can just as well be replaced by some other



(a) Full fingerprint (b) Truncated fingerprint

Figure 7.1: Fingerprint with and without truncation by a plan-cost threshold

threshold τ chosen to reduce the effort in optimization.

The effect of such a change at the level of fingerprints is illustrated in Figure 7.1. Figure 7.1(a) shows the fingerprint under the disk-nested-loops model of the basic chain query with mean cardinality 10^4 and variability 0.5. In Figure 7.1(b) we see the result of removing the dots for all sets S such that $\kappa^{out}(S)$ lies above the threshold $\tau = 10^{14}$. (The value 10^{14} has no special significance; we comment further on the choice of τ below.) Evidently, Figure 7.1(b) reflects a much lower total κ^{split}-execution count than does the original fingerprint in Figure 7.1(a). Imposition of the threshold has eliminated many dots, and moreover has removed them mostly from the right-hand half of the fingerprint, the province of "hard" sets.

Under the naive cost model as well, plan-cost thresholds can remove large numbers of sets from a fingerprint. The absence of "hard" sets under the naive cost model makes the effect of the thresholds only slightly less compelling. Indeed, plan-cost thresholds make sense for any cost model κ for which the split-independent component κ^{out} is nonzero. If κ^{out} is zero, all the points in a fingerprint plot will be glued to the x-axis, and a plan-cost threshold will have no effect. But any realistic cost model ought to have a non-zero split-independent component that charges some cost proportional to output cardinality. The cost need not be large to make the plan-cost thresholds effective.

There remains the question of how to choose a threshold τ. Plainly, the lower the threshold, the greater the savings in optimization time. But in lowering the threshold too aggressively, one runs the risk of rendering some queries unoptimizable. That is, if the threshold is set below the cost of the optimal plan for a given query, the optimizer will be

unable to find any plan for that query. (Note, though, that in no case will the optimizer commit the insidious error of mistaking a suboptimal plan for an optimal one. If any plan is found, it will be optimal.) There is no obvious criterion for choosing a threshold that strikes a good compromise. Below we consider several perspectives on this problem.

7.2 Experimental Runs with Plan-Cost Thresholds

Our first attack on the setting of threshold values is straightforward if *ad hoc*, and gives us the experimental results reported below. We reason as follows:

If a query is expected to execute quickly, then it is imperative that it also be optimized quickly, for in general one would like optimization time to be smaller than (preferably, *much* smaller than) execution time [30]. By the same token, if the query execution will be extremely long, then it is probably acceptable to allow more time for optimization, as long as the optimization time is still small compared to the execution time.

On this basis, it seems reasonable to choose a plan-cost threshold that represents a query-execution time on the order of one hour. The optimizer will then directly find optimal plans for queries whose estimated execution time (after optimization) is below one hour. On the other hand, given a query that cannot run in under one hour, the optimizer will fail to find a plan; it will then be necessary to take remedial action. The remedial action we take is to reoptimize with a higher threshold—for lack of a better value, we set the second threshold at 10^9 hours. The second threshold will clearly suffice for any query that one may reasonably hope to execute. But for the sake of completeness we make allowance for a *third* optimization pass if the second threshold is still too low. It will turn out that this third pass is required for some of our basic test queries.

In our experimental runs with plan-cost thresholds, we make no attempt to reuse information from one optimization pass to the next; each optimization pass is independent. As a result, in those cases where we optimize first with a threshold of one hour, and then with a threshold of 10^9 hours, the aggregate time spent on optimization will be the *sum* of the time needed to optimize with the lower threshold and the time needed to optimize with the higher threshold. Similarly, if there are *three* optimization passes, the aggregate

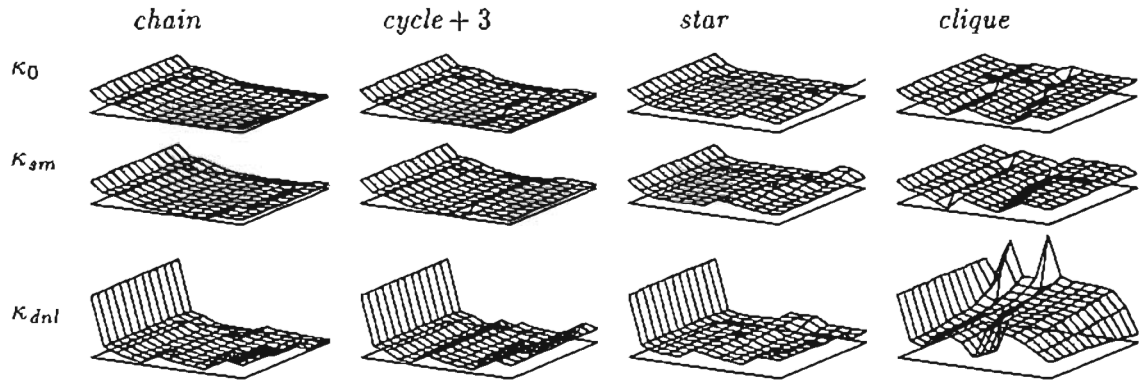
optimization time will be the sum of the optimization times for each pass. In this sense, the time spent on the earlier passes ends up being completely wasted.

Our cost models have no time units associated with them, so to choose thresholds that represent one hour, we need to make common-sense estimates. Since the costs in the naive and sort-merge cost models count tuples, one may take the cost units for these models to represent times on the order of microseconds. The costs in the disk-nested-loops model count disk-block transfers, and presumably represent times on the order of tens of milliseconds. Using these ball-park estimates as a guide, we take the threshold 10^9 as an approximation for one hour under the naive and sort-merge cost models; for the disk-nested-loops model, the corresponding threshold is 10^5.

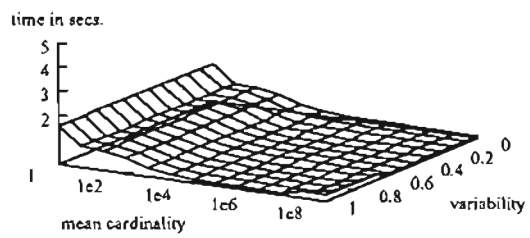
Figure 7.2, which is organized the same way as Figure 6.1 on page 153, shows the optimization times we obtain after altering our optimizer to incorporate the plan-cost thresholds just described. The optimization times depicted in Figure 7.2 tend to run markedly lower than the corresponding times in Figure 6.1. In particular, for chain queries under the naive cost model, Figure 7.2(b) exhibits optimization times that settle down to a scant 0.1 second. Figure 7.2(c) illustrates a variation of the same effect for the case of "*cycle*+3" queries under the disk-nested-loops model: optimization times rapidly drop off as cardinality rises—but then ripples appear where the plan-cost thresholds are exceeded, and additional optimization passes are required.

Plan-cost thresholds are effective for all join-graph topologies, but the benefits are most pronounced for chain-like graphs. Recall that the lowest points in the chain-query fingerprints become extremely sparse; a sufficiently low threshold will cut away the vast majority of the points and leave behind only a handful, nearly all of which correspond to "easy" sets.

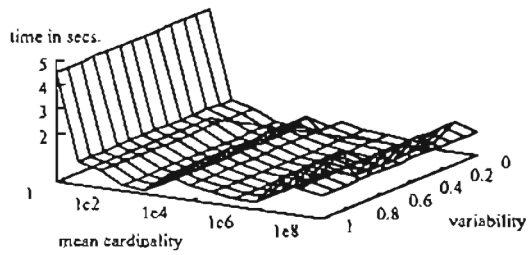
There is another way of understanding the effectiveness of plan-cost thresholds for chain queries. A well-placed plan-cost threshold tends to make a fairly clean separation between the sets that involve Cartesian products and those that do not. In the terminology of Starburst, the threshold cuts away the "infeasible" sets and leaves behind the "feasible" sets. Moreover, in searching for the optimal plan for each feasible set, our optimizer will reject splits whose *operand_cost* exceeds the threshold, and hence will tend to perform a



(a) Four-dimensional summary of performance sensitivities with plan-cost thresholds



(b) $\kappa_0/chain$; threshold at 10^9



(c) $\kappa_{dnl}/cycle + 3$; thresholds at 10^5, 10^{14}

Figure 7.2: Optimization times for 15-way joins with plan-cost thresholds

full cost computation only for the splits that correspond to "feasible" joins. The net result is that our κ^{split}-execution counts fall to a level that lies in the neighborhood of the number of feasible joins for the query. The effect is particularly dramatic for chain queries because, as we have noted, the number of feasible joins for such a query is only polynomial in the number of relations in the query. But for star queries as well, we obtain κ^{split}-execution counts consistent with the feasible-join counts set forth by Ono and Lohman [45].

In effect, when equipped with plan-cost thresholds, our optimizer excludes Cartesian products after all. But the exclusion is cost-based and not topology-based. Since the more conventional topology-based exclusions only approximate the effects that are actually desired, we obtain two advantages by directly applying a cost-based criterion. First, we do not *necessarily* exclude Cartesian products—we just *tend* to exclude them. If the optimal plan for a query includes a Cartesian product, then our cost-based exclusion will admit

that Cartesian product and allow the optimizer to generate the optimal plan that includes it. Second, the cost-based exclusion allows us to prune away excessively costly sets even when they do *not* involve Cartesian products. Because of this second effect, we sometimes obtain κ^{split}-execution counts that actually fall *below* the feasible-join counts based on topology.

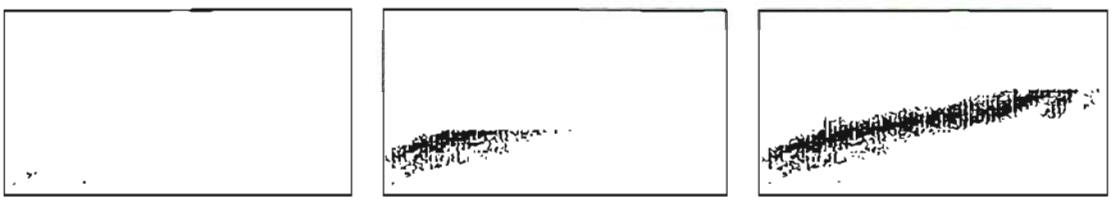
7.3 Considerations in Choosing Plan-Cost Thresholds

We still face the challenge of choosing effective plan-cost thresholds for a given query. The benefits of these thresholds diminish when the thresholds fail to give a fairly tight bound on the cost of the optimal plan.

One way to obtain a fairly tight plan-cost threshold, suggested by Roberto Bayardo [3], would be to precede the exhaustive-search optimization with a brief run of a stochastic join-optimization method such as iterative improvement. An alternative, more brute-force strategy is to use multiple optimization passes with successively larger thresholds, as described in Section 7.2 above, but with much finer spacing between the thresholds. In this way the successive thresholds could work their way up to a tight upper bound by approaching it from below.

This seemingly ungainly strategy may well prove worthwhile in situations where cost computations dominate the optimization time. Figure 7.3 gives a sense of the trade-off involved in adopting a finer spacing of the thresholds. The figure presents several views of the disk-nested-loops fingerprint for our basic chain query with mean cardinality 10^4 and variability 0.5. In Figure 7.3(a), we see the effects of the successive plan-cost thresholds we used in our experiments of Section 7.2 above; i.e., the successive thresholds are 10^5, 10^{14}, and 10^{23}. Figure 7.3(b) shows the effect we would have obtained by spacing the thresholds only a factor of 10 apart, i.e., by using successive thresholds of 10^5, 10^6, 10^7, and so on.

In our experiments, the threshold 10^5 proved too low for the query under consideration, and so it had to be reoptimized with the next threshold of 10^{14}, which turned out to be adequate. But it is apparent that processing all the sets in the second fingerprint



$\tau = 10^5$ $\tau = 10^{14}$ $\tau = 10^{23}$

(a) Thresholds spaced a factor of 10^9 apart

$\tau = 10^5$ $\tau = 10^6$ $\tau = 10^7$

(b) Thresholds spaced a factor of 10 apart

Figure 7.3: Fingerprints truncated by successively larger plan-cost thresholds

of Figure 7.3(a) entails a hugely greater effort than is required for the first fingerprint. This big jump in effort is reflected in the height of the leftmost ripple in the $\kappa_{dnl}/chain$ and $\kappa_{dnl}/cycle+3$ cells of Figure 7.2(a); this pronounced ripple effect is even more clearly discernible in Figure 7.2(c). Thus, the appeal of using the succession of thresholds illustrated in Figure 7.3(b) is that the amount of effort needed for successive optimization passes would be greatly reduced. One would expect the ripple heights to diminish correspondingly.

On the other hand, in switching to more closely spaced thresholds, one runs the risk that a much larger number of optimization passes might be needed, and the total effort involved might therefore be larger. As it happens, the threshold of 10^6 illustrated in the middle of Figure 7.3(b) suffices for our example query, and so the third and subsequent optimization passes can be avoided. But there is no assurance that other queries might not pose greater difficulties.

However, even if many optimization passes should be required, the amount of redundant effort involved can be capped. As we argue next, it is not necessary for any of the cost computations to be repeated. What does need to be repeated on successive passes is the enumeration of the join alternatives. Since the number of passes will be logarithmic in the optimal plan cost, the enumeration effort will also become logarithmic in the optimal plan cost. This resultant moderate increase in enumeration effort for expensive queries should not be grounds for concern in cases where the cost computations dominate the total optimization time.

7.4 Plan-Cost Slices

Our strategy for using plan-cost thresholds in the experiments of Figure 7.2 was rather crude. On successive optimization passes, we simply reexecuted the Blitzsplit algorithm with different thresholds, without making any substantive internal changes to the algorithm. However, in taking this crude approach, we unnecessarily discarded valuable information obtained in one run of the algorithm, and then had to recreate that information in the next run.

A better approach would be to retain the dynamic programming table from one run to the next, at first building a table full of guesses and "question marks," and then using successive passes to patch up and improve the table until it contained an optimal plan—though it might still be full of guesses and "question marks" elsewhere. Then rather than calculating all costs up to an ever-increasing *threshold*, each pass of the optimizer would peform cost calculations related to a particular horizontal *slice* of the query fingerprint. The resultant savings in cost-function executions would be especially beneficial if one were dealing with more computationally intensive cost functions than we have considered in this work.

Here we briefly sketch how such an approach to iterative application of the Blitzsplit algorithm could be implemented. One would need to extend the dynamic programming table to maintain *two* cost columns, rather than just the one we had before (*cf.* Figure 3.1). The Cost column we had before would continue to represent total plan cost, but the Cost

entry for a given set $\mathcal{S}$ would not always be the *lowest* cost of any plan for $\mathcal{S}$; instead, it would be an *upper bound* on such costs—that is, a running *best cost so far* that would be carried over from one iteration of the algorithm to the next. Meanwhile, the *new* cost column would store the values of the split-independent cost $\kappa^{out}(\mathcal{S})$ for each set $\mathcal{S}$. The values of this new column would be calculated up front, and would never change subsequently.

The Blitzsplit algorithm would then run iteratively, and each iteration would concern itself with a cost-slice defined by an interval $[slice\_low, slice\_high)$. The upper end of the interval used in one iteration would become the lower end of the interval used in the next iteration, so that successive slices would stack up on top of one another. The central observation to make is that on any given iteration, the split-dependent cost $\kappa^{split}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs})$ for a given split should never be calculated unless

$$slice\_low \leq \text{Cost}(\mathcal{S}_{lhs}) + \text{Cost}(\mathcal{S}_{rhs}) + \kappa^{out}(\mathcal{S}) < slice\_high. \tag{7.1}$$

What is a little tricky about this rule is that the values of $\text{Cost}(\mathcal{S}_{lhs})$ and $\text{Cost}(\mathcal{S}_{rhs})$ could *change* from one iteration to the next. However, by the same token, this rule (together with the subsets-first assumption) ensures that *if* one of operand costs should change on a given iteration, its new value cannot possibly be less than *slice_low*—so there is no danger of failing to cost a split because its operands suddenly "shrink away" beneath the cost-slice window! Moreover, it is not difficult to see that under this rule, the split-dependent cost $\kappa^{split}(\mathcal{S}, \mathcal{S}_{lhs}, \mathcal{S}_{rhs})$ for a given split will be calculated *at most once*, no matter how many times the algorithm iterates.

On successive iterations, the slice levels work their way upward, while the plan costs work their way downward; when the two converge, optimization is complete. Specifically, when a cost below *slice_high* is obtained for the set $\mathcal{R}$ of all base relations, one may deduce that the plan rooted at the table entry for $\mathcal{R}$ is optimal.

In terms of cost-function execution counts, this iterative approach could be expected to perform quite well. Not only does it not repeat any cost computations, but it goes out of its way to postpone performing these computations in the first place; it finally performs them only when it becomes quite clear that they cannot be pruned away with confidence.

If sufficiently thin slices were used, then κ^{split} would almost never be computed for a split whose *operand_cost* exceeded the lowest *dependent_cost* for the set. In effect, this iterative algorithm would tend to drive the points in a fingerprint plot somewhat leftward, in addition to removing nearly all the points that lay above the level of the optimal plan cost.

Just *how* well this iterative approach would perform would naturally depend on the query to be optimized. An approach of this kind should be especially effective for fingerprints that become sparse in their lower regions; the queries that yield such fingerprints include chain queries, but they can also include queries of other topologies when there is significant variation in both the base-relation cardinalities and predicate selectivities. Our basic test queries are inadequate for studying the performance of an iterative algorithm of this kind because by construction those queries tend to suppress vertical variation in the fingerprint plots. A more meaningful benchmark would very likely have to consider a larger number of dimensions of parameterization of the input query space.

7.5 Summary and Discussion

In this chapter we have sought to address a weakness of the Blitzsplit algorithm that manifests itself when the cost function κ, and in particular its split-dependent component κ^{split}, is computationally intensive. For while the Blitzsplit algorithm performs extremely rapid enumeration of join alternatives (or "splits"), it potentially performs a much larger number of cost-function executions than the Starburst and Volcano algorithms.

In response to this weakness, we observed first that through the simple and rather crude technique of applying *plan-cost thresholds*, it was possible to reduce the number of cost-function executions substantially, especially for "chain-like" queries. These reductions were reflected in improved optimization times for nearly all the basic test queries to which we had applied our original performance measurements previously. We noted, moreover, that with appropriately chosen plan-cost thresholds, cost-function execution counts could be brought down to levels comparable with those that obtain in the Starburst and Volcano algorithms. In particular, cost-function execution counts for chain queries become

polynomial in the number of relations in the join.

We then speculated on how the notion of plan-cost thresholds might be refined into a notion of *plan-cost slices* that could more reliably reduce the number of cost-function executions. Although correct implementation of plan-cost slices involves some tricky details, the mechanisms required are not complicated, and can be expected to prune away cost computations to a degree that might be difficult to achieve by any other means.

The idea of plan-cost slices deserves further exploration. It should be noted that pursuit of cost-computation savings by this means involves a trade-off that may be regarded as counterintuitive: for by imposing an extra layer of iteration on the Blitzsplit algorithm, plan-cost slices effectively sacrifice the algorithm's rapid enumeration of join alternatives. In a sense, such a trade goes against the spirit of the Blitzsplit algorithm, which was constructed to perform enumeration at blinding speeds.

But it is precisely its fleetness in enumeration that allows the Blitzsplit algorithm to be applied in ways that would be inconceivable under a heavier-weight enumeration strategy. Optimizers that use heavier-weight strategies cannot afford the luxury of iterating the entire enumeration process many times over, and consequently they do not have the option of repeatedly going back and revisiting their pruning decisions. They have no choice but to make these decisions conservatively, lest they prune too much. Thus constrained, heavier-weight optimizers are likely to miss some of the pruning possibilities that become so important when costing of plans becomes computationally intensive.

Chapter 8

A Stochastic Extension

The performance results for the Blitzsplit algorithm reported in Chapters 6 and 7 were encouraging, but there are reasons for seeking even better performance. No matter how much pruning one achieves in the cost computations, the Blitzsplit algorithm's complexity always retains an exponential term that renders the algorithm infeasible for queries involving very large numbers of relations. Furthermore, direct extension of the Blitzsplit algorithm to incorporate consideration of physical properties (*cf.* Section 2.6.3) could degrade its performance substantially. In this chapter we consider a stochastic extension of the Blitzsplit algorithm that can handle larger numbers of relations, and that has the potential to be more resilient in the presence of complicating considerations such as physical properties.

The ideas described in this chapter are not fundamentally new. Swami [56] explored a variety of approaches to join optimization that combined heuristic and combinatorial techniques. One approach he considered was a hybrid technique that he referred to as *local improvement*; this technique improved randomly generated plans by applying dynamic programming to subproblems of the join-optimization problem. The approach we take in the present chapter may be regarded as a straightforward variant of Swami's local improvement, at least in conception. (The details of our approach are different—and more complicated—in large part because we generate bushy plans, rather than just left-deep plans, as Swami did.) Swami reported disappointing performance for the local-improvement technique—other techniques he explored proved far more effective. The present investigation would therefore appear to be poorly motivated by Swami's results.

But two considerations justify giving local improvement a second look. First, the

Blitzsplit algorithm permits rapid solution of subproblems by dynamic programming, and increases the size of subproblem for which exhaustive solutions are feasible. Second, methods analogous to local improvement have proved effective in problem domains unrelated to query processing, and may have something to teach us. The present work on stochastic optimization takes its inspiration largely from the combinatorial-optimization work of Martin and Otto [39], who have devised a stochastic technique they call Chained Local Optimization. They have applied this technique with astonishing success to a variety of well-known intractable problems such as the Traveling Salesman Problem. Their technique is similar to Swami's local improvement, but differs subtly in its inclusion of a "kick" mechanism, about which we will say more in due course.

Part of the appeal of applying a technique such as local improvement to join-order optimization is that it gives us a way of retaining some of the benefits of dynamic programming when moving on to problems involving large numbers of relations. Conventional stochastic techniques achieve only a modest amount of sharing in the analysis (including the costing effort) that goes into comparing the different plans considered. Hybrid techniques that incorporate dynamic programming to solve subproblems can explore a larger proportion of the search space, for a given amount of effort, than those that do not.

In Section 8.1 below, we review the intuitions behind Chained Local Optimization, and contrast it with other stochastic optimization techniques that have been applied to join optimization. The remaining sections of this chapter describe the central mechanisms of our adaptation of the Chained Local Optimization technique to join-order optimization; we refer to the resultant algorithm as the *Stochastic Bushwhack* algorithm.

However, we shall defer incorporation of Chained Local Optimization's "kick" into our algorithm until Chapter 9, in which we conduct empirical studies on the Stochastic Bushwhack algorithm's behavior. We shall see that the algorithm is rather effective even without the "kick," but that inclusion of the "kick" increases its power still further.

8.1 Intuitions about Stochastic Optimization

Consider the function depicted by the solid curve in Figure 8.1. Call this function f.

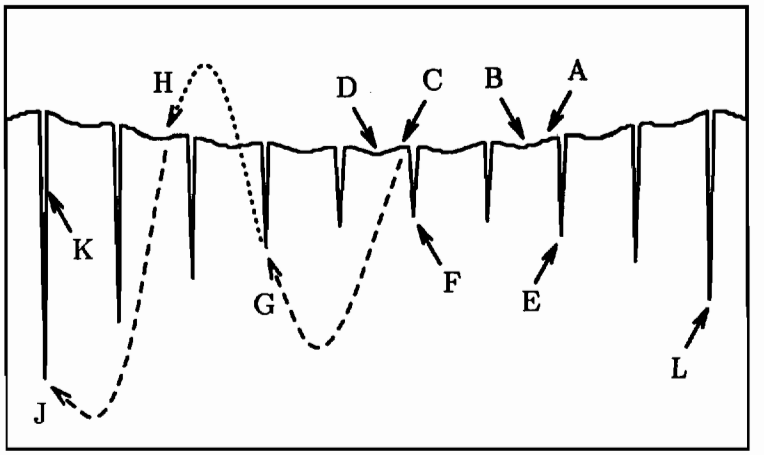


Figure 8.1: A pathological function shape

We see from the figure that f has a gently-sloping, scalloped surface punctuated by deep crevasses.

Now suppose we are faced with the problem of finding the minimum of the function in question; in other words, our objective is to find the x-coordinate of the point at which the function is lowest—which happens to be the point labeled J in the lower left-hand corner of the figure. Let us assume for the present that the only means we have to examine the ups and downs of the function curve is to evaluate $f(x)$ at a succession of values x of our choosing. Thus, we have nothing like the derivative of f at our disposal.

The problem just posed is loosely analogous to the problem of join-order optimization. We may think of the different x-values as corresponding to different join orders or join *expressions*, and the function f as corresponding to the *cost* function defined by equations (2.59) and (2.60) on page 53. Let us therefore try to develop a very approximate, intuitive sense of the strengths and weaknesses of several stochastic join-optimization techniques by considering how these techniques would cope with the problem of finding a minimum for f. Subsequently we will also consider how Swami's local-improvement technique and the Chained Local Optimization technique of Martin and Otto would cope with this problem.

8.1.1 Characteristics of Various Approaches

Iterative Improvement The algorithm known as *iterative improvement* [54, 58] is very simple, but quite effective. Given the problem at hand it might proceed as follows. First, it probes f at a randomly chosen value x; let us say that it probes the point labeled A in Figure 8.1. It then probes f at $x + \delta$ and $x - \delta$, where δ is some suitably small increment, seeking a lower point on the function surface. If it finds such a point, it adjusts x accordingly, and repeats the process. In this way it "climbs" downhill until it reaches a local minimum. In the present example, it might find its way down to B.

The entire hill-climbing is then repeated for a different random starting point, say C. This time the algorithm might find its way to D. After some number of repetitions of this hill-climbing procedure, the algorithm has found its way to an assortment of local minima, B and D among them. The best of these local minima is taken as the solution to the problem.

But one can see the difficulty with iterative improvement in the present instance. The odds of finding a deeper local minimum such as E or F are relatively low, since the large majority of points on the surface lead down to shallower minima. The algorithm therefore requires a very large number of repetitions to stand a good chance of finding a deep minimum.

Simulated Annealing *Simulated annealing* [27, 54, 58] partially overcomes the difficulty encountered by iterative improvement in the problem at hand. The simulated-annealing algorithm is somewhat similar to iterative improvement, but rather than methodically climbing downhill at each step, this algorithm permits both downhill and uphill moves. It probabilistically gives preference to downhill moves so as to ensure net downward motion, and the likelihood of an uphill move declines over time so that the algorithm eventually settles down to a local minimum. But by allowing uphill moves, the algorithm can "climb over" small obstacles that lie between a shallow minimum and a deeper minimum. Thus, starting at B in Figure 8.1, simulated annealing might find its way to E, or starting at D, it might find its way to F.

The problem with simulated annealing is that its progress is *slow*. It makes many

false moves in the course of making its way to a deep minimum. Moreover, as usually formulated, simulated annealing proceeds only from a single starting point, and so it finds only a single local minimum, albeit a deep one. There may be other deep local minima that are far better. (Theoretically, simulated annealing can be parameterized so that it finds a global minimum with probability 1; but the time required may be astronomical.)

Two-phase Optimization *Two-phase optimization* or 2PO [26, 54] attempts to combine the advantages of iterative improvement and simulated annealing. In the first phase, a number of iterations of iterative improvement are performed so as to obtain a low point among the shallow minima. In the present instance, the low point found in this manner might be the point labeled D. In the second phase, the low point so identified is taken as a starting point for simulated annealing. Thus, at the end of the second phase, we might arrive at the point labeled F.

The intuition behind two-phase optimization is that the deepest minima are likely to be found near the deepest among the shallower minima. But this method backfires when faced with a pathological function of the sort shown in Figure 8.1. Here the depth of the deep minima is negatively correlated with the depth of the nearby shallow minima, and the two-phase optimization strategy leads to a deep minimum that does not compare especially favorably with the global minimum.

Transformationless Random Probing The *transformationless random probing* technique of Galindo-Legaria et al. [12] picks each point to probe entirely at random. In other words, unlike most techniques, it is not constrained to probe in the neighborhood of the last point it has probed.

The strength of this approach lies in its rejection of spatial proximity as a criterion for selecting successive points to probe. By forgetting where it has probed before, it avoids getting stuck in local minima such as B and D in Figure 8.1—or for that matter, in deeper but still suboptimal local minima such as E and F.

But this algorithm's forgetfulness is also its weakness. On probing the point K in Figure 8.1, it has come within a hair of finding the global minimum. A small effort

spent exploring the neighborhood of K would reveal points much lower than K itself. However, points near K are no more likely to be probed after K has been probed than they were before. When a function has the shape illustrated in Figure 8.1, the probability of obtaining a very low point strictly through random probes is relatively small—and we can make it as small as we like by narrowing the crevasses.

8.1.2 Incorporation of a Heuristic

The techniques just described were purely stochastic techniques. Let us now move on to consideration of techniques that still have a stochastic aspect, but also incorporate heuristics. Specifically, we consider Swami's *local improvement* technique and the Chained Local Optimization technique of Martin and Otto.

Swami's approach requires an initial join-processing tree as a starting point for the local improvement. Local improvement involves repeated application of dynamic programming to obtain a succession of lower and lower points. Each use of dynamic programming involves only a subset of the relations in the join to be optimized, and hence cannot be expected to give a global minimum. However, the dynamic programming still achieves an exhaustive search of some *subspace* of the space of all possible plans; what is more, by construction the subspace that is searched includes the plan that served as a starting point for the search. It follows that this exhaustive search of the subspace will yield a plan no worse than the starting point, and possibly better.

Using our analogy from Figure 8.1, let us say that the starting point is the point labeled C in the figure. Successive applications of dynamic programming might improve on the starting point by giving, say, points D, F, and G; let us suppose, for the sake of argument, that G cannot be improved upon by this method. Note that while there is a sense in which these successive points are "near" to one another—each adjacent pair of them evidently belongs to a common subspace of plan space—this notion of proximity is rather loose. It might take *many* steps of a purely stochastic technique such as iterative improvement to achieve the same progress that is achieved in a single step of local improvement (though certainly each step of local improvement requires more effort). Perhaps more significantly, in many instances the requisite sequence of steps would not be possible under iterative

improvement, because of obstacles intervening between the starting and ending points of a local-improvement step.

The Chained Local Optimization of Martin and Otto begins the same way as Swami's local improvement. It presumes the existence of some "heuristic" that rapidly improves on a starting point, if improvement is possible within some "distance" of the starting point. Iterated application of dynamic programming, as in local improvement, conforms to the requirements of a heuristic that might be used in Chained Local Optimization. Thus, starting at point C in Figure 8.1, Chained Local Optimization applies the heuristic, and directly obtains the point G (as suggested by the dashed line from C to G).

But rather than settle for G as an approximation to the global minimum, Chained Local Optimization proceeds as follows. The point G is displaced by a random "kick," giving a nearby point, say H (as suggested by the dotted line). The kick provides no immediate benefit, but it provides a new starting point for another application of the heuristic, which may then lead to a new local minimum (e.g., J). The new minimum may or may not improve upon the previous local minimum; if it does, progress has been made. If not, one can go back to the previous minimum and try a different kick (or one may accept the new, inferior minimum with some probability, as in simulated annealing). The entire process of applying the heuristic, and then giving the solution a kick, is repeated until no further improvement is achieved.

As with any stochastic technique, Chained Local Optimization cannot guarantee that a global optimum will be found. In particular, a very deep local minimum such as L in Figure 8.1, which happens to be distant from any equally deep minimum, might well confound any stochastic optimization technique, including Chained Local Optimization. But Martin and Otto have reported success with Chained Local Optimization at finding global minima for notoriously difficult optimization problems.

The intuition behind Chained Local Optimization is that the structure of a solution space may be intricate and jagged, but it is not random. Solutions that are very good but not optimal are likely to share some features with the global optimum. By "kicking" a solution to obtain a new starting point for the heuristic search, rather than taking a new starting point entirely at random, one hopes to hang on to features of the solution that

may lead the way to still better solutions.

8.1.3 Shapes of Join-Plan Spaces

The motivating remarks above revolved around the pathological function illustrated in Figure 8.1. One may question whether the pathologies of this function have anything to do with the realities of plan-spaces for join-order optimization problems.

Ioannidis and Kang [26] argue that large classes of query plan spaces are essentially bowl-shaped. But this observation does not apply universally, and does not rule out pathologies of the sort we have discussed. Moreover, the results of Ioannidis and Kang depend on a mathematical model whose conformity to actual query-plan spaces is not well-established. In particular, the continuous functions in their model may differ in important ways from the corresponding functions on actual plan spaces, which are discrete domains [37].

In this connection it is also worth noting an observation made by Swami [57] and by Galindo-Legaria et al. [12] regarding the proportion of points in query-plan space whose cost is close to that of the global minimum. This proportion tends to *decline* as the number of relations n increases. This effect suggests that at larger n, the plan space is indeed pockmarked with crevasses of some sort.

With the current state of knowledge about the subject, it is difficult either to defend or to refute broader claims about the character of query-plan space. We have motivated the application of Chained Local Optimization through two-dimensional, visual metaphors that may or may not accurately reflect properties of the many-dimensional spaces a query optimizer must deal with. But the experimental results reported in Chapter 9 below suggest that, whatever the flaws of our motivating arguments, the intuitions they provide have not grossly misled us.

8.2 Tightening and Iterated Tightening

We now describe the technique of *tightening* a query plan. The idea behind tightening is that when a join-optimization problem is too large to be solved by dynamic programming,

it may nonetheless be possible to take advantage of dynamic programming in searching for approximately optimal solutions. We will illustrate the concept of tightening, and then *iterated* tightening, by working through a simple example.

8.2.1 A Sample Problem

Suppose we are confronted with the join-optimization problem illustrated by the labeled join graph in the left-hand portion of Figure 8.2(a). The problem is to join five relations A, B, C, D, and E,[1] with the base-relation cardinalities and predicate selectivities shown in the illustration. (The base-relation cardinalities are the numbers inside the small boxes.) We shall assume the naive cost function κ_0, which makes the cost of each join equal to the cardinality of the result.

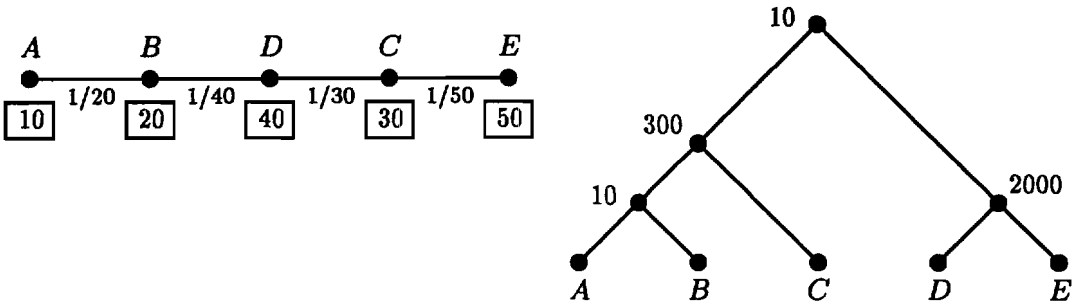
It would be straightforward to optimize the given five-way join by exhaustive search, but for the sake of argument, let us imagine that exact optimization of five-way joins is infeasible. Let us suppose, however, that exact optimization of three-way joins *is* feasible, and can be achieved rapidly. We shall use a series of three-way join optimizations to help us find a good plan for the five-way join.

8.2.2 The Initial Join-processing Tree

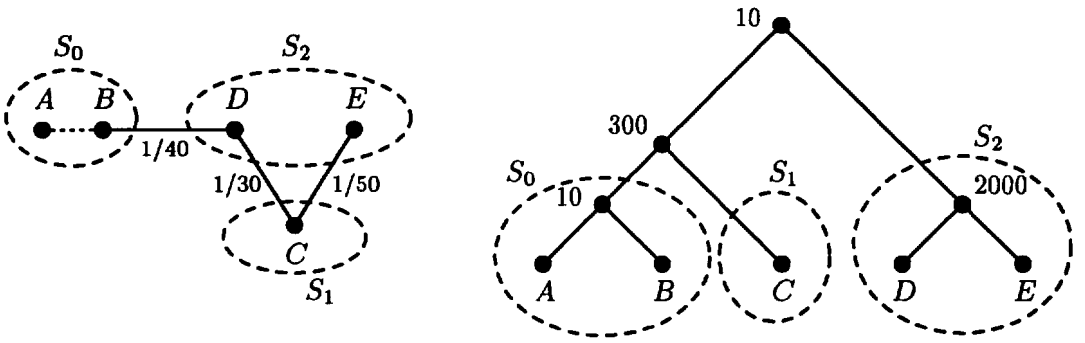
Our technique requires that we start out with *some* plan for computing the five-way join; our tactic will be to improve on the given plan. Let us say that our initial plan is the join-processing tree illustrated in the right-hand portion of Figure 8.2(a). Each node of the processing tree is annotated with the cardinality of the intermediate result computed at that node. Under the assumed naive cost model, the cost of a join is equal to the cardinality of the result, and consequently the annotation at each node can also be read as the cost of processing that node. The total cost of the plan is then the sum of these annotations.

Note that the given initial plan is not a particularly good one. It needlessly computes two sizable Cartesian products—for example, the join of D and E is a Cartesian product,

[1]There is no connection between the use of the relation names A, B, C, D, and E here, and the use of the letters A, B, C, D, and E (among others) as labels in Figure 8.1.



(a) Annotated join graph (left) and one possible initial join-processing tree (right)



(b) Groupings of the base relations into pseudo-relations S_0, S_1, and S_2

(c) Collapsing pseudo-relations into encapsulated pseudo-relations

Figure 8.2: Collapsing a join-optimization problem to a smaller problem

since there is no predicate connecting D and E in the join graph. However, our method makes no requirement that the initial plan be free of such deficiencies.

8.2.3 Collapsing Subtrees to Pseudo-relations

Now let us see how we may use the initial plan as a guide to finding a better plan.

The right-hand portion of Figure 8.2(b) shows one way that the five-way join-processing tree of Figure 8.2(a) can be viewed as a three-way join-processing tree. Each of the three circled subtrees computes a relation, and the relations so computed may be thought of as the inputs of a three-way join. We refer to these three inputs as *pseudo-relations*, and in this example we give them the names S_0, S_1, and S_2.

By extracting a three-way join-processing tree from a larger tree, we create an opportunity to invoke our three-way optimizer to obtain a possibly better tree. But we have a little more work to do before we can invoke the three-way optimizer. The input required by the three-way optimizer consists of the cardinalities of the three relations to be joined, together with the selectivities of the predicates connecting those relations. In the present instance, the "relations" to be joined are the pseudo-relations S_0, S_1, and S_2. There is no difficulty in furnishing their cardinalities: the cardinality of S_0, for example, is just 10—the estimated cardinality of the intermediate result $A \bowtie B$. But what are the predicate connections among S_0, S_1, and S_2? We address this question next.

8.2.4 Collapsing the Join Graph

To determine the predicate connections among the pseudo-relations, let us reexamine the join graph of our original optimization problem. The left-hand portion of Figure 8.2(b) shows an altered view of the original graph, with circles enclosing the sets of relations that have been joined into pseudo-relations in the right-hand portion of the figure. In the context of the three-way join of interest, we must regard each pseudo-relation as a relation; hence, each circled group of relations will become one node of the join graph for the three-way join. The nodes of the three-way join graph are thus obtained by *collapsing together* nodes of the original five-way join graph. We refer to the process of converting the five-way join graph to a three-way join graph as *collapsing the join graph.*

To construct the edges of the collapsed join graph, we must consider which predicates will be applied when the pseudo-relations are joined. For example, when S_0 is joined with S_2, the predicate connecting B and D becomes applicable, since S_0 represents the join of A and B and S_2 represents the join of D and E. From this example one can see that whenever an edge of the original join graph spans two pseudo-relations, that edge will be retained in the collapsed join graph as an edge between the pseudo-relations.

However, in the original join graph, there are *two* edges that span S_1 and S_2—one that connects D and C, and another that connects E and C. Let us give the names p and q to the corresponding predicates; thus, p is some predicate that refers to attributes in D and C, and q is some predicate that refers to attributes in E and C. It follows that a join of D and C will be qualified by p—such a join might be written as $D \bowtie_p C$—and a join of E and C will be qualified by q. But if D and E are first joined with one another, and the result is then joined with C, then both the predicates p and q become applicable in the top-level join. Thus, this join may be written as $(D \bowtie E) \bowtie_{p \wedge q} C$. If one assumes, as we do here, that selectivities of distinct predicates are independent, then the selectivity of the compound predicate $p \wedge q$ is equal to the product of the selectivities of p and q. Since the join of S_2 and S_1 is nothing more than the join of $D \bowtie E$ and C, the predicate qualifying this join is the compound predicate $p \wedge q$.

Extrapolating from this example, one can see that whenever multiple edges of the original join graph span a given pair of pseudo-relations, those edges may be combined into a single edge in the collapsed join graph. The selectivity of the resultant edge is just the product of the selectivities of the combined edges.

(Note that multiple edges can be collapsed to a single edge even if the assumption of independence of predicate selectivities does not hold. All that changes in the more general case is that the selectivity of the collapsed edge must be computed differently.)

Finally, observe that the edge connecting A and B in the original join graph must be discarded when these two relation nodes are collapsed into a single pseudo-relation node in the collapsed join graph. Discarding this edge is appropriate because the predicate it represents has already been applied in the computation of $A \bowtie B$, and its selectivity has been taken into account in the estimation of the cardinality of S_0.

8.2.5 Encapsulation of Pseudo-relations

Figure 8.2(c) shows the result of applying the foregoing observations to the example at hand. The left-hand portion of the figure shows the collapsed join graph, with each pseudo-relation encapsulated in a single node. These nodes are annotated with the estimated cardinalities of the pseudo-relations; for example, the annotation 2000 for S_2 is taken from the estimated cardinality of $D \bowtie E$ in the initial join-processing tree.

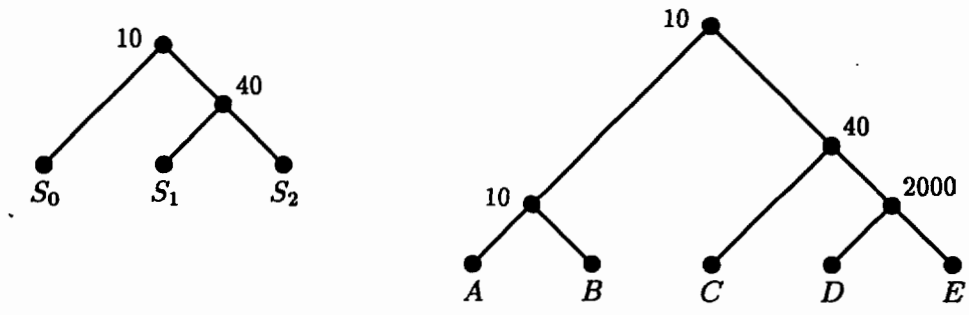
The collapsed join graph contains two edges. The edge connecting S_0 and S_2 reflects the B–D edge in the original join graph, and has the same selectivity. On the other hand, the edge connecting S_2 and S_1 is a synthesis of the D–C and E–C edges in the original graph; the selectivities of these edges, 1/30 and 1/50 respectively, have been multiplied together to derive the selectivity 1/1500 of the synthesized edge.

The right-hand portion of Figure 8.2(c) gives a view of the initial join-processing tree as a plan for the join-optimization problem described by the collapsed join graph. The pseudo-relations at the leaves of the collapsed tree encapsulate subtrees of the original tree in such a way that we may temporarily forget about the details of those subtrees. In particular, we may proceed on the assumption that the evaluation cost of the pseudo-relations is zero; in reality, the evaluation costs of the encapsulated subtrees may be substantial, but whatever they are, they must be paid regardless of how we compute the three-way join of S_0, S_1, and S_2. Thus, for the purposes of comparison with alternative plans for the three-way join, we may estimate the cost of the join-processing tree in Figure 8.2(c) at $300 + 10 = 310$.

If we now apply the three-way join optimizer to the problem described by the collapsed join graph, we expect to obtain a join-processing tree equivalent to—but possibly better than—the tree in Figure 8.2(c).

8.2.6 Subproblem Optimization and Grafting

Dynamic programming algorithms for join optimization do not require an initial join-processing tree as a starting point for subsequent improvement. Rather, they create new trees from scratch. Consequently, we do not in fact need the collapsed join-processing tree



(a) Optimal three-way join (b) With pseudo-relations expanded

Figure 8.3: Tightened join-processing tree, before and after grafting

from Figure 8.2(c) to optimize the three-way join of S_0, S_1, and S_2. All we need is the information in the collapsed join graph: the collapsed relation cardinalities, the collapsed predicate connections, and the associated selectivities.

However, though not required, the collapsed initial join-processing tree does serve a useful purpose, since it gives us an upper bound on the cost of the optimal plan for the three-way join. (In this case, as we have seen, it gives an upper bound of 310.) We can use this upper bound as a plan-cost threshold to reduce the amount of time required in the optimization of the three-way join (*cf.* Chapter 7).

Figure 8.3(a) shows the join-processing tree that results from performing an exhaustive optimization of the three-way join expressed by the collapsed join graph. The evaluation cost for this tree is seen to be $40 + 10 = 50$. Thus, the improvement over the initial collapsed tree is $310 - 50 = 260$.

We can now achieve the same improvement of 260 in the initial, *uncollapsed* join-processing tree of Figure 8.2(a). We do so by *grafting* the subtrees encapsulated by the pseudo-relations onto the optimal tree of Figure 8.3(a), giving the five-way join-processing tree of Figure 8.3(b). The latter tree computes our original five-way join at a cost of 2060—which is 260 less than the initial tree's cost of 2320.

We use the term *tightening* to describe the entire sequence of operations illustrated above. At the end of the sequence, the initial five-way join-processing tree has been "tightened" into a more efficient five-way join-processing tree. Tightening may or may

not improve the initial tree; but as formulated here, tightening is *guaranteed* not to make the initial tree worse.

The relative improvement we have obtained by tightening of our sample initial join-processing tree is modest. In this instance, what is more important than the cost reduction itself is the fact that we have restructured the tree in a way that facilitates further tightening.

8.2.7 Tightening of Subtrees

Above we applied tightening to the top level of the initial join-processing tree. That is, the collapsed join tree computed a three-way join that was equivalent to the initial five-way join. But it is also possible to apply tightening to subtrees of an initial join-processing tree.

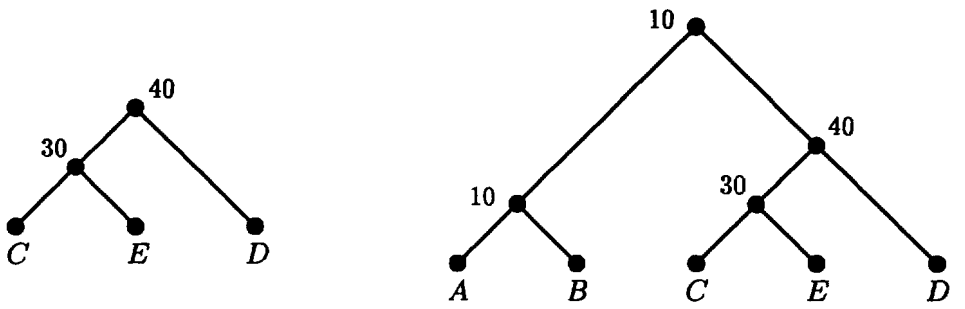
Had we attempted to tighten the subtrees of the initial tree in Figure 8.2(a), we would not have obtained any benefit, since the subtrees in that tree were already optimal. For example, although it is very inefficient to compute the join of D and E (which is really a Cartesian product), the subtree that performs this computation cannot be improved in isolation from the rest of the join-processing tree. Viewed as a tree in its own right, the subtree that computes $A \bowtie B \bowtie C$ in Figure 8.2(a) is also optimal.

However, in general, there are opportunities for improvement within subtrees, and in particular, the tree obtained in Figure 8.3(b) presents such an opportunity. Figure 8.4(a) shows how the cost of the subtree that computes the join of C, D, and E, can be reduced from 2040 to 70. In Figure 8.4(b), the optimized subtree is reintegrated into the five-way join-processing tree, whose total evaluation cost has now been reduced to 90.

8.2.8 Iterated Tightening

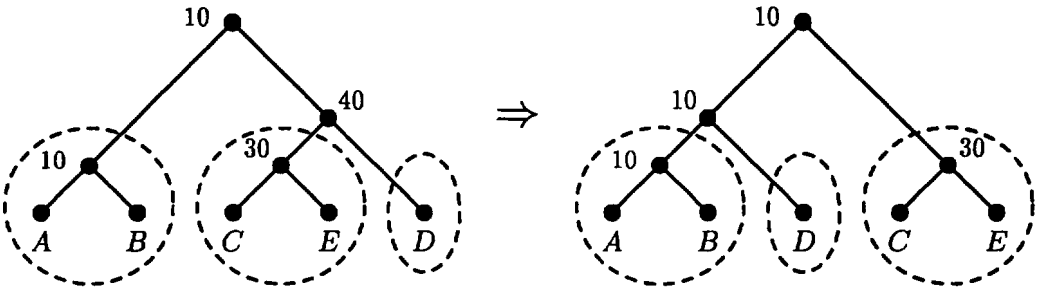
Just as tightening at the top level created an opportunity for improvement in one of the resultant subtrees, now we will find that tightening the subtree has altered the structure of the top-level tree in such a way that it is profitable to tighten it once again.

This time we shall not go into the details of the tightening of the top-level tree. Figure 8.5 illustrates the outlines of retightening the tree that was obtained in Figure 8.4(b).

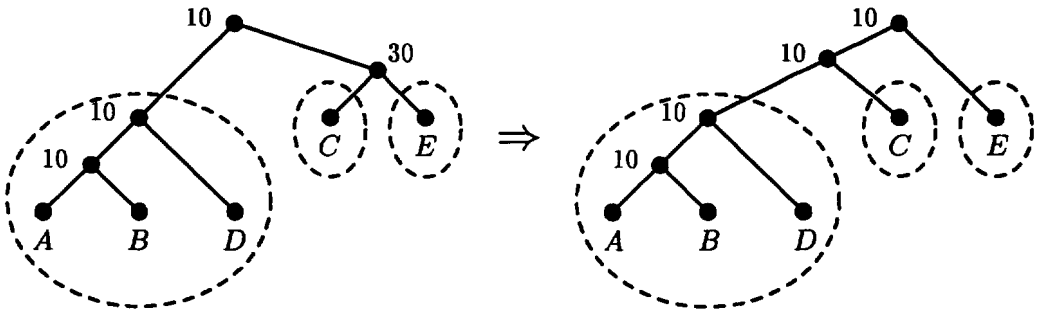


(a) Optimal plan for C, D, and E (b) As a subplan of five-way processing tree

Figure 8.4: Tightening of a subtree



(a) Summary of one tightening operation



(b) A second tightening, using different pseudo-relations

Figure 8.5: Retightening the top-level tree after tightening of subtree

It can actually be retightened twice. In Figure 8.5(a), we start with the tree obtained in Figure 8.4(b) and tighten it by treating the circled subtrees as pseudo-relations—the result of this tightening is shown on the right-hand side of the double-arrow. In Figure 8.5(b), we repeat the process, this time starting with the tree that was obtained on the right-hand side of Figure 8.5(a), but with different pseudo-relations.

8.2.9 Summary

We have concluded our example of tightening and iterated tightening. We began in Figure 8.2(a) with a join-optimization problem, represented as a join graph, and an initial join-processing tree for that problem. Both these ingredients were necessary for tightening. We needed the initial join-processing tree for extracting "pseudo-relations," and we needed the join graph for constructing a "collapsed" three-way join-optimization problem involving the pseudo-relations. Upon solving this smaller problem, we grafted subplans of the initial processing tree onto the optimal three-way plan, and obtained a five-way processing tree that improved on the initial one.

By repeatedly applying tightening operations, we eventually arrived at a plan that happens to be optimal for the five-way join in question. Not surprisingly, it will not always possible to find a global optimum simply by applying a sequence of local improvements, as we have done here. But our example aptly illustrates how each tightening step tends to create opportunities for additional tightening steps.

8.3 An Algorithm for Tightening

We now generalize the examples of tightening given in the previous section, and present pseudo-code that applies tightening steps to an arbitrary join-optimization problem.

The presentation is in two parts. In the present section, we give an algorithm that carries out an individual tightening operation. In the next section, we present an algorithm that implements a particular policy of iterated tightening. In both instances, the presented code is in pseudo-ML, and is abstracted from our implementation in Standard ML [42].

Earlier, in presenting the Blitzsplit algorithm, we used imperative pseudo-code and discussed an imperative implementation, because *updates* of the dynamic programming table were an essential ingredient of the algorithm, and because low-level features of C enabled us to fine-tune the implementation. Here we face somewhat different considerations. Our stochastic algorithm is more complicated than the Blitzsplit algorithm, and for ease of coding and debugging it was preferable to build a prototype in a high-level language. Moreover, using functional pseudo-code permits fairly compact presentation of the algorithm, which would be quite unwieldy if presented in imperative form. In our presentation below, we assume basic familiarity with functional programming notation.

8.3.1 The Tightening Algorithm Proper

Figure 8.6 shows the declarations and functions that together implement the *tightening* operation described in the previous section—but without any iteration of the tightening steps.

8.3.2 Type Declarations

The pseudo-code of Figure 8.6 begins with several type declarations.

The types *rel_index*, *rel_info*, and *pred_info* in Figure 8.6 are analogous to the types *relation_name*, *rel_data*, and *predicate* used in the Blitzsplit algorithm. (See Figure 3.1 on page 75 and Figure 5.4 on page 124.) However, here the representations for *rel_info* and *pred_info* are somewhat simpler, in that we do not bother with record types. Instead, we just represent a relation by its cardinality (which is a *real*), and we let predicates take the form $(\{endpoint_0, endpoint_1\}, selectivity)$, where $endpoint_0$ and $endpoint_1$ are of type *rel_index*, and *selectivity* is a *real*. Note that we use curly braces ("{" and "}") to denote mathematical sets. (In Standard ML, curly braces have a different interpretation.)

The *problem* type is intended to characterize a join-optimization problem; we assume that a join optimizer requires a data structure of type *problem* as its input. Such a data structure consists of a *list* of relation descriptions paired with a *set* of predicate descriptions. For example, the join-optimization problem depicted by the join graph in

```
type rel_index = integer
type rel_info = real
type pred_info = rel_index set * real
type problem = rel_info list * pred_info set

datatype plan = REL of rel_index | JOIN of plan * plan


fun pick (t as REL _, k) (node_list, cost) = (t :: node_list, cost)
  | pick (t, 1) (node_list, cost) = (t :: node_list, cost)
  | pick (t as JOIN(lhs, rhs), k) (node_list, cost) =
      let val k_rhs = min(max(k div 2, k - num_leaves lhs), num_leaves rhs)
      in  pick (lhs, k - k_rhs) (pick (rhs, k_rhs) (node_list, cost + node_cost t))
      end

fun pick_subplans (t, k) = pick (t, k) ([], 0.0)


fun collapse_rels subplans i = j such that REL i is a leaf of subplans_j


fun collapse_preds (f, preds) =
      let val spanning_pairs = {{f i, f i'} | ({i, i'}, sel) ∈ preds; f i ≠ f i'}
          fun selectivity {j, j'} = ∏{sel | ({i, i'}, sel) ∈ preds; f i = j; f i' = j'}
      in  {({j, j'}, selectivity {j, j'}) | {j, j'} ∈ spanning_pairs}
      end


fun graft (subplans, REL j) = subplans_j
  | graft (subplans, JOIN(lhs, rhs)) = JOIN(graft(subplans, lhs), graft(subplans, rhs))


fun tighten (preds, k) plan =
      let val (subplans, curr_cost) = pick_subplans (plan, k)
          val crels = [card t | t ← subplans]
          val f = collapse_rels subplans
          val cpreds = collapse_preds (f, preds)
          val skeleton = optimize_with_threshold curr_cost (crels, cpreds)
      in  graft (subplans, skeleton)
      end
    handle exception Failed_To_Beat_Threshold ⇒ plan
```

Figure 8.6: Tightening algorithm

Figure 8.2(a) might be represented as the *problem*

$$([10, 20, 30, 40, 50], \{(\{A, B\}, 1/20), (\{B, D\}, 1/40), (\{C, D\}, 1/30), (\{C, E\}, 1/50)\}),$$

where A, B, C, D, and E are mnemonic stand-ins for the relation indexes 0, 1, 2, 3, and 4. We shall use these mnemonics in all our examples; similarly, when the need arises, we shall use the pseudo-relation names S_0, S_1, and S_2 rather than the explicit indexes 0, 1, and 2.

Finally, the type *plan* characterizes the output of join optimization, namely a query plan. The *plan* datatype declaration generates data constructors REL and JOIN for creating, respectively, the leaves and the internal nodes of a join-processing tree. For example, the three subtrees encapsulated as pseudo-relations in Figures 8.2(b)–(c) can be constructed as

$$s_0 = \text{JOIN}(\text{REL } A, \text{REL } B) \tag{8.1}$$

$$s_1 = \text{REL } C \tag{8.2}$$

$$s_2 = \text{JOIN}(\text{REL } D, \text{REL } E); \tag{8.3}$$

using these trees as building blocks, one can then construct the initial join-processing tree of Figure 8.2(a) as

$$t_0 = \text{JOIN}(\text{JOIN}(s_0, s_1), s_2). \tag{8.4}$$

(Here and in later examples, we use the lower-case designation s_j for the *tree* encapsulated by a pseudo-relation, while the upper-case designation S_j refers to the pseudo-relation itself (which hides the encapsulated tree). As noted above, a designation of the form S_j is just a mnemonic description of the integer index j which identifies the pseudo-relation.)

8.3.3 Implicit Functions

The *plan* type declaration as just described omits some details present in our actual implementation. In the actual implementation, each node of a processing tree is annotated with various information, such as the estimated cardinality of the join computed at that node, the estimated cost of computing that join, and the total estimated evaluation cost of the subplan rooted at the node in question.

fun *node_cost* (REL _) = 0
  | *node_cost* (*t* **as** JOIN(*lhs*, *rhs*)) = κ(*t*, *lhs*, *rhs*)

fun *total_cost* (REL _) = 0
  | *total_cost* (*t* **as** JOIN(*lhs*, *rhs*)) = *total_cost lhs* + *total_cost rhs* + *node_cost t*

Figure 8.7: Function equivalents to cost annotations on tree nodes

To compensate for the absence of these annotations in the pseudo-code, we shall assume the existence of several functions that furnish cardinalities and estimated costs on demand. We call these functions *card*, *node_cost*, and *total_cost*. The first is self-explanatory, and the latter two behave as if defined in the manner shown in Figure 8.7. In the definition of *node_cost*, the arguments of κ are *representations*, respectively, of the result relation computed at the given join node, and of the relations that serve as the inputs of this join. The definition of *total_cost* restates, in new notation, the definition of *cost* given by equations (2.59) and (2.60) on page 53.

We shall also assume the existence of a function *num_leaves* that counts the number of REL nodes (i.e., the number of leaf nodes) in a plan or subplan.

8.3.4 Functions for Tightening

We now give a very brief account of the functions defined in Figure 8.6.

Function *pick_subplans* The function *pick_subplans* identifies the subplans in a processing tree that are to be encapsulated as pseudo-relations. The number of pseudo-relations desired is supplied as an argument to *pick_subplans*. For example, if t_0, s_0, s_1, and s_2 are as defined above in (8.1)–(8.4), and if the cardinalities and predicates of the join-optimization problem are as illustrated in Figure 8.2(a), then

$$pick\_subplans\ (t_0, 3) = ([s_0, s_1, s_2], 310).$$

The first component of the result is the list of subplans identified for encapsulation. We refer to such a list as a *pseudo-relation plan list.* The second component is the cost of the collapsed processing tree illustrated in Figure 8.2(c).

Note that *pick_subplans* is defined in terms of an auxiliary function *pick*, about which we comment further below.

Function *collapse_rels* Given a pseudo-relation plan list and a relation index, the function *collapse_rels* returns the index of the pseudo-relation that contains the specified relation. For example,

$$\textit{collapse\_rels}\ [s_0, s_1, s_2]\ D = S_2,$$

because the relation D occurs in the tree s_2. Note that we may think of the expression *collapse_rels* $[s_0, s_1, s_2]$ as being a function in its own right. Thus, if we define

$$f = \textit{collapse\_rels}\ [s_0, s_1, s_2], \tag{8.5}$$

then we have $f\ D = S_2$, $f\ A = S_0$, and so on. The pseudo-code uses *collapse_rels* in just this way.

Function *collapse_preds* The function *collapse_preds* collapses a set of predicates that connect relation indexes to a set of predicates that connect pseudo-relation indexes. It does so with the help of a relation-collapsing mapping such as is provided by *collapse_rels*. For example, given f as defined in (8.5) above,

$$\textit{collapse\_preds}\ (f, \{(\{A, B\}, 1/20), (\{B, D\}, 1/40), (\{C, D\}, 1/30), (\{C, E\}, 1/50)\}) \\ = \{(\{S_0, S_2\}, 1/40), (\{S_1, S_2\}, 1/1500)\}.$$

In the code for *collapse_rels*, i and i' represent relation indexes, while j and j' represent pseudo-relation indexes.

Function *graft* The function *graft* grafts a list of subplans onto the leaves of a given tree. For example,

$$\textit{graft}\ ([s_0, s_1, s_2], \text{JOIN}(\text{REL}\ S_0, \text{JOIN}(\text{REL}\ S_1, \text{REL}\ S_2))) = \text{JOIN}(s_0, \text{JOIN}(s_1, s_2)).$$

In the pseudo-code, the notation $\textit{subplans}_j$ denotes element number j in the list *subplans* (counting from 0); e.g., for arbitrary a, b, and c, $[a, b, c]_2 = c$.

Function *tighten* The function *tighten* begins by invoking *pick_subplans* to create a pseudo-relation plan list called *subplans*.

Then it constructs a join-optimization *problem* as follows. First, it creates a list *crels* of pseudo-relation descriptions (i.e., cardinalities) by means of the list comprehension "$[card\ t \mid t \leftarrow subplans]$." Second, it creates a set *cpreds* of collapsed-predicate descriptions by invoking *collapse_preds*.

The join-optimization problem (*crels*, *cpreds*) is then submitted to the optimization function *optimize_with_threshold*, which is assumed to provide an interface to the Blitzsplit algorithm. The arguments (*crels*, *cpreds*) describe the relations and predicates to which the Blitzsplit algorithm is to be applied. But we also furnish the algorithm with the argument *curr_cost*, which represents a plan-cost threshold for optimization, as described in Section 7.1. We assume that *optimize_with_threshold* returns an optimal join plan in the form a join-processing tree. (This optimal plan is named *skeleton* because it has pseudo-relations at its leaves, and is not really a complete plan.)

Finally, the pseudo-relation plan list is grafted onto the optimal plan skeleton produced by the optimization function. The skeletal plan is thus transformed into a complete plan—though the *complete* plan that results is not necessarily optimal.

Note that the plan-cost threshold furnished to the optimization function is derived from the original plan to be tightened. If a plan cannot be found with a cost below this threshold, the optimization function is assumed to fail with the exception *Failed_To_Beat_Threshold*, whereupon *tighten* just returns as output the original plan it was given as input—i.e., the given input plan cannot be tightened, and so it is left as is.

8.3.5 Technical Issues

Identification of Pseudo-Relations

As noted above, we express the function *pick_subplans* in terms of the auxiliary function *pick*. In implementing the latter function, we are forced to make a policy decision, for there are many possible ways to pick the k subplans to encapsulate.

The policy adopted here is based on a breadth-first traversal of the plan tree. In effect, *pick* descends through the tree until it reaches a level where there are k nodes across the

breadth of the tree. It makes appropriate adjustments when the tree is unbalanced, and favors the left branch when k is odd.

That this left-leaning breadth-first policy is the most sensible one is by no means obvious. Note, specifically, that the tightenings illustrated in Figure 8.5 are not possible under a left-leaning policy. One can imagine alternative policies that might be more powerful; in particular, the idea of a stochastic policy has some appeal. On the other hand, the drawback of a stochastic policy is that *because* each tightening of a tree under such a policy offers the possibility of new improvements, one never knows when to stop attempting to tighten a given tree.

Computing Costs of Collapsed Plans

In addition to selecting subplans, *pick* performs the side-task of summing the node costs of the nodes traversed in reaching those subplans. It thus obtains the cost of the collapsed tree that would result if the subplans were replaced by stubs.

This side-task is an annoying complication in the implementation of *pick*. In principle, one could achieve the same result much more simply: If t is the given tree, and *subplans* is the list of chosen subplans, one could simply take

$$total\_cost\ t \quad - \sum_{s \leftarrow subplans} total\_cost\ s.$$

However, we shun this alternative because of the risk of numerical error. Suppose the total cost of the collapsed plan were 100, and that the sum of the costs of the subplans were 10^{25}. The total cost of the original plan t would then be $10^{25} + 100$, but conventional floating-point representations lack the requisite precision to distinguish this value from 10^{25}. Consequently, the total would be recorded as 10^{25}, and subtraction of the sum of the subplan costs would yield 0 as the cost of the collapsed plan.

8.4 The *Stochastic Bushwhack* Algorithm

We now present what we refer to as the *Stochastic Bushwhack* algorithm. Given a join query to optimize, the Stochastic Bushwhack algorithm begins by constructing a *random*

initial join-processing tree for that query. It then iteratively improves the initial tree by invoking the tightening algorithm on different portions of the tree, as described below.

8.4.1 Pseudo-code for the Stochastic Bushwhack Algorithm

Figure 8.8 gives pseudo-code for the Stochastic Bushwhack algorithm. The principal functions of this algorithm are as follows.

Functions *shake_up* and *shake_down* The *shaking* functions implement two forms of iterated tightening. Given an initial join-processing tree, *shake_up* first recursively shakes up its left- and right-hand children, and then performs a tightening operation at the top level. Loosely speaking, *shake_up* tightens all subtrees of the given tree, starting at the leaves, and then working its way upwards.

By contrast, *shake_down* first tightens its argument at the top level, and then recursively shakes down the left- and right-hand children of the result. Thus, *shake_down* once again tightens all subtrees of the given tree, but starts at the top and works its way downwards.

Both *shake_up* and *shake_down* take an argument of the form $(preds, k)$, which is needed only so that it can be passed through to *tighten*. (The *preds* are needed to compute the predicates that will be handed to the Blitzsplit algorithm, and k determines the size of join-optimization problem that will be handed to the Blitzsplit algorithm.) Note also that both *shake_up* and *shake_down* are expressed in terms of the auxiliary function *app_children*, which simplifies the recursion.

Function *itshake* The function *itshake* implements the following iterated tightening policy. Given an initial join-processing tree and a factor *fac*, *itshake* first shakes the tree up, and then shakes it down. If the resultant tree has a total cost below that of the initial tree by at least a factor of *fac*, the entire process is repeated. But if on any given iteration the improvement is less than a factor of *fac*, the recursion ceases, and the most recently obtained tree is returned as the result.

```
fun app_children f (t as REL _) = t
  | app_children f (t as JOIN(lhs, rhs)) = JOIN(f lhs, f rhs)

fun shake_up (preds, k) plan =
      tighten (preds, k) (app_children (shake_up (preds, k)) plan)

fun shake_down (preds, k) plan =
      app_children (shake_down (preds, k)) (tighten (preds, k) plan)

fun itshake (preds, k) fac old_plan =
      let val plan = shake_down (preds, k) (shake_up (preds, k) old_plan)
      in  if total_cost old_plan/total_cost plan < fac
            then plan
            else itshake (preds, k) fac plan
      end

exception Empty_Tree_Bug

fun rand_tree [] = raise Empty_Tree_Bug
  | rand_tree [(i, cardinality)] = REL i annotated with cardinality
  | rand_tree leaves =
      let val z = a random integer from 1 to 2^(length leaves) - 2
          fun divvy (z, []) (l, r) = (l, r)
            | divvy (z, x :: xs) (l, r) = divvy (z div 2, xs)
                  (if z mod 2 = 0 then (x :: l, r) else (l, x :: r))
          val (l, r) = divvy (z, leaves) ([], [])
      in  JOIN(rand_tree l, rand_tree r)
      end

fun bushwhack k fac (rels, preds) =
      let val indexed_rels = [(i, rels_i) | i ← [0 .. length rels - 1]]
          val init_plan = rand_tree indexed_rels
      in  itshake (preds, k) fac init_plan
      end
```

Figure 8.8: The Stochastic Bushwhack algorithm

Function *rand_tree* Given a list of indexed relation descriptions (i.e., a list of (relation index, cardinality) pairs), *rand_tree* constructs a random join-processing tree for joining the given relations. It does so by choosing a random split of the relations into a left-hand list and a right-hand list. By recursive calls to *rand_tree*, random processing trees are generated for the left- and right-hand lists, and the processing trees so obtained are then combined with a JOIN node to obtain a processing tree for the entire list.

Two aspects of this tree-generation scheme deserve comment. First, *no attempt is made to avoid Cartesian products*—the predicates in the join-optimization problem are not even considered in the tree construction.

Second, if the random splits are drawn from a uniform distribution (as implied by the pseudo-code), then the different possible processing trees will *not* all be generated with equal probability. Since the trees that conform to a given nearly-balanced split are far more abundant than the trees that conform to an unbalanced split, uniform generation of the trees would require biasing the split selection in favor of nearly-balanced splits. By using a uniform distribution for the splits, in effect we bias the choice of trees in favor of those that are unbalanced.

However, there is no reason to believe that this bias is harmful. Unbalanced trees are so rare, relatively speaking, that despite the bias we are still far more likely to choose a nearly-balanced tree.

Function *bushwhack* The function *bushwhack* constitutes the top level of the Stochastic Bushwhack algorithm. It simply constructs a random initial join-processing tree for the problem at hand, and then improves it by invoking *itshake*.

8.4.2 Technical Issues

Termination of Iterative Tightening

One of the Stochastic Bushwhack algorithm's parameters is *fac*, which determines at what point *itshake* gives up and stops attempting to tighten the plan further. In our present work, we consistently use a value for *fac* of 1.00001—in other words, effectively we continue tightening as long as there is measurable improvement in the plan, even if the improvement

is minuscule.

Note that no matter how close *fac* is to 1, we cannot guarantee that the plan has been tightened down as much as it can be. In theory, a particular iteration of *itshake* might yield no improvement, and yet it might be possible to obtain improvement on a subsequent iteration. One would have to be especially wary of such an effect if *pick* were made nondeterministic in the manner suggested in Section 8.3.5 above.

In practice, we have found that iterated tightening tends to improve plans in huge leaps, followed in some instances by one or two small corrections before a local minimum is reached. Only rarely does *itshake* recurse more than twice. The precise value of *fac* does not appear to be especially critical.

Problems Involving More Than 30 Relations

In the present work, we do not consider joins of more than 30 relations. There is no obvious inherent reason why the Stochastic Bushwhack algorithm cannot handle larger problems. However, one must pay attention to a couple of issues of numerical representation when the number of relations n rises above 30.

First, on most machines, the integer expression $2^{(\textit{length leaves})} - 2$ in the function *rand_tree* will overflow when n exceeds thirty. To avoid the problem, *rand_tree* would have to be revised. (Note, though, that larger values of n will not cause any difficulty in the calls to the Blitzsplit algorithm inside the function *tighten*. The Blitzsplit algorithm will be called only with smaller numbers of relations or pseudo-relations, and these relations will be identified by very small integers inside the Blitzsplit algorithm, even if they are identified by larger integers in the context of the *tighten* function itself.)

Second, when n exceeds thirty, the cardinalities at some nodes of randomly generated plan trees, and the selectivities of some collapsed predicates, may cause overflow or underflow of double-precision floating-point formats. (For example, the Cartesian product of 31 relations that each have cardinality 10^{10} would have cardinality 10^{310}—too much for the double-precision format of the IEEE standard.) Thus, for larger n, a different representation of cardinalities, selectivities, and costs would be needed.

In our implementation, we simply represent these values in double-precision floating

point. Note that in our implementation of the Stochastic Bushwhack algorithm, and also in the version of the Blitzsplit algorithm that is called from inside the Stochastic Bushwhack algorithm, *we do not take the short-cut of representing costs in single-precision floating point.* It is not clear that tightening could be made to work correctly if costs above 10^{38} became indistinguishable.

In a *tuned* implementation of Stochastic Bushwhack, one might incorporate *two* versions of the Blitzsplit algorithm: one for the cases where the plan-cost threshold was representable in single-precision floating point, and another for the remaining cases.

8.5 Summary and Discussion

In this chapter we discussed intuitions about stochastic optimization, and drawing on those intuitions, we developed a stochastic join-optimization algorithm. As the first step in this development, we presented a *deterministic* "tightening" algorithm that uses the Blitzsplit algorithm to *improve* plans for joining large numbers of relations. The next step was to present the Stochastic Bushwhack algorithm, which generates a *random* initial plan for a join query, and then improves this initial plan through iterated tightening.

The Stochastic Bushwhack algorithm is *ad hoc* in the sense that there is no apparent reason why the tightening steps should be carried out in precisely the sequence prescribed by the algorithm. Indeed, other sequencing policies might work equally well or better. Nor is it clear that starting with a *random* initial plan is a good idea.

Even if we set aside such doubts, and assume that the Stochastic Bushwhack algorithm's design is appropriate, it is not obvious from first principles how one would best take advantage of what this algorithm can do. Should one run the algorithm just once, to obtain an "approximately" optimal plan for a query, and hope for the best? Or, is it advisable, for good results, to run the algorithm many times, so that many different initial plans are explored? How should one choose the parameter k—i.e., the number of relations that will be handed off to the Blitzsplit algorithm in each tightening step?

Because these questions are difficult to answer from first principles, we address them through empirical studies in the next chapter.

Chapter 9

Performance of the Stochastic Extension

In this chapter, we conduct empirical studies on the Stochastic Bushwhack algorithm presented in the last chapter. Performance of a stochastic algorithm has two aspects: the quality of the solutions obtained, and the amount of time it takes to find them. Here we study both of these aspects of performance in the Stochastic Bushwhack algorithm, and the relationship between them.

Swami [57] has broadly characterized the quality of plans obtained from stochastic join optimizers as follows:

- A *low-cost* or *good* plan has an estimated cost within a factor of 2 of the optimum.
- An *acceptable* plan has an estimated cost that is more than twice the optimum, but that does not exceed the optimum by more than a factor of 10.
- Any other plan is *bad.*

(If the optimal cost for a query is unknown, as can happen with large queries, then Swami substitutes the cost of the best plan that can be found by any stochastic optimization technique.)

Swami's distinctions are useful, but here we will pay special attention to plans that are not merely *good*, but *optimal.* For it is apparent that under at least some conditions, the Stochastic Bushwhack algorithm will yield optimal plans. Recall that the algorithm has a parameter k that specifies the number of relations (or pseudo-relations) in the subproblems that will be optimized by exhaustive search. When k is equal to the number of relations n in the original query, the Stochastic Bushwhack algorithm reduces to exhaustive search.

The random initial join-processing tree generated by *bushwhack* becomes irrelevant, as tightening will inevitably transform this tree into an optimal one.

Thus, when $k = n$, a single invocation of *bushwhack* suffices to obtain a solution of perfect quality. On the other hand, the time it will take to obtain this solution will be the same as for exhaustive search. (Actually, the time will be slightly higher, because of additional overhead.) But one can also see that a given invocation of *bushwhack* will *sometimes* yield an optimal solution even when k is *below* n, for if just by blind luck the random initial tree generated by *bushwhack* should have *almost* the right structure, then tightening will again transform this tree into an optimal one. The advantage of reducing k below n is that the time needed for optimization should also be reduced, since the calls to the Blitzsplit algorithm will involve substantially less work. In short, we expect that by adjusting k, we ought to be able to trade off optimization time against the likelihood of obtaining an optimal solution.

In the following, after a few preliminaries, we begin our empirical investigation by examining the expected trade-off in the context of a single, narrow class of join-optimization problems parameterized only by the number n of base relations. We then investigate the extent to which the results obtained for that single class of problems generalize to other join-optimization problems.

Despite our algorithm's similarity to Swami's local-improvement technique, and despite his essentially negative results, we shall find the Stochastic Bushwhack algorithm to be highly effective. In summarizing our observations in Section 9.12, we speculate on some of the possible reasons for the discrepancy between Swami's results and our own.

9.1 Concept of Watersheds

In presenting our measurements below, we shall frequently make reference to the notion that the space of query plans is divided into *watersheds*. Here we explain what is meant by a watershed, and why we care about them.

As we have seen, the way that *bushwhack* optimizes (or approximately optimizes) a query is to first generate a random initial plan for the query, and then to call *itshake* to

tighten the initial plan down to a *local minimum.* One can see that there must be distinct choices of initial plans that *itshake* will tighten down to the *same* local minimum. We can picture plan-space as a mountainous surface with ridges, peaks and valleys; initial plans chosen from the same valley will all be tightened down to the same plan at the bottom of the valley.

There is a sense in which the boundaries between the valleys, and the number of valleys, are intrinsic features of the plan-space, since these topographical features are determined by the costs of the plans, and not by any characteristic of an optimizer. On the other hand, the ridges between valleys are not all of equal height, and so the distinctness of adjacent valleys becomes a matter of interpretation: what might be regarded as a low ridge might also be regarded as a mere bump in the landscape. The tightening operations of the Stochastic Bushwhack algorithm enable it to "tunnel" under the smaller ridges that separate distinct valleys, so that from the perspective of the algorithm, these valleys become indistinguishable. The larger the value of k, the better the algorithm can "tunnel," so that even some of the larger ridges become insignificant.

We shall use the word *watershed* to describe a portion of plan-space in which all plans are tightened to the same local minimum. Thus, when subjected to the action of *itshake*, all plans in a watershed flow down to the same place. The number of watersheds, and the boundaries between them, are *not* intrinsic to plan space, but depend on the parameterization of the Stochastic Bushwhack algorithm.

The division of plan space into watersheds is of interest because it sheds light on the likelihood of obtaining optimal solutions. For example, suppose plan space is divided into four watersheds of roughly equal size. Then regardless of the random initial tree generated by Stochastic Bushwhack, it will be tightened down to one of four local minima. One of those local minima must be the global minimum—so the odds of attaining the global minimum are about one in four.

Now that we have the intuition of a watershed, we must make a minor amendment to its definition. Distinct plans that have the same cost are equally good and need not be distinguished, and hence may be considered to belong to the same watershed. Thus, widely separated valleys whose lowest points have the same "elevation" become connected

in conceptual plan space. We will regard two points in plan space as belonging to the same watershed if they tighten down to points (i.e., to plans) that have the same cost.

9.2 Measurement Procedure

The test queries we use in this chapter are constructed according to the four-dimensional parameterization discussed in Section 6.1.2 and described in detail in Appendix C. However, here we do not fix the number of relations n at 15, and so n becomes another parameter of our tests. In addition, our tests also will vary the *bushwhack* parameter k.

Thus, we have two new dimensions of parameterization in addition to the four we had before, for a total of six dimensions. Typically we will vary two dimensions at a time, while holding the others fixed. In particular, we will hold the cost function fixed at κ_0 in all tests except as noted. To start out, we will focus on queries with mean cardinality 10^4 and variability 0.5—a kind of middle-of-the-road, average case among the basic test queries of each topology; we refer to these as the *canonical test queries* of each topology. Only later on do we explore variation along the dimensions of mean relation cardinality and variability.

Most of the measurements reported below are illustrated first for the case of the *cycle*+3 topology; we later report the analogous measurements for other topologies as well.

For a given query specification, and for a given k, our measurement procedure consists of the following steps:

- Construct the specified query.
- Invoke *bushwhack* 1000 times, with the given query and k as parameters (and with $fac = 1.00001$), thus obtaining iterated tightenings of 1000 randomly chosen initial trees. The 1000 tightened trees so obtained are expected to cover the spectrum of results that *bushwhack* may yield when presented with the given optimization problem.
- Sort the costs of the 1000 tightened trees in ascending order. The resultant sequence of 1000 costs becomes the raw data for the quality-of-optimization analysis described

below.

- Record the total CPU time required for all 1000 runs. This total will serve as the basis for the estimated average running time of *bushwhack* per invocation. (The time needed for each individual invocation is also recorded, but because of clock granularity, these timings are not very informative. However, they do establish approximate bounds on the variability in the running time of individual invocations.)
- Record miscellaneous additional performance statistics, which will be mentioned below where they are relevant.

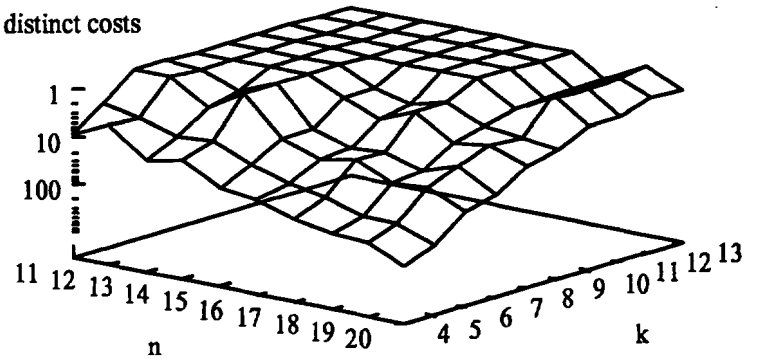
All measurements were taken on a Standard ML implementation of the Stochastic Bushwhack algorithm running on a Sun SPARCstation 20.[1] We used the SPARCstation 20 for these measurements, rather than the SPARCstation 2 or HP 9000/755 mentioned in previous chapters, because of the availability and relatively high speed of the SPARCstation 20. Performance of the SPARCstation 20 is slightly below that of the HP 9000/755.

Our test runs did not maintain memory-usage statistics; note, however, that the memory requirements of stochastic join-optimization techniques tend to be rather modest, and there is no reason to think that the Stochastic Bushwhack algorithm would be an exception to this rule.

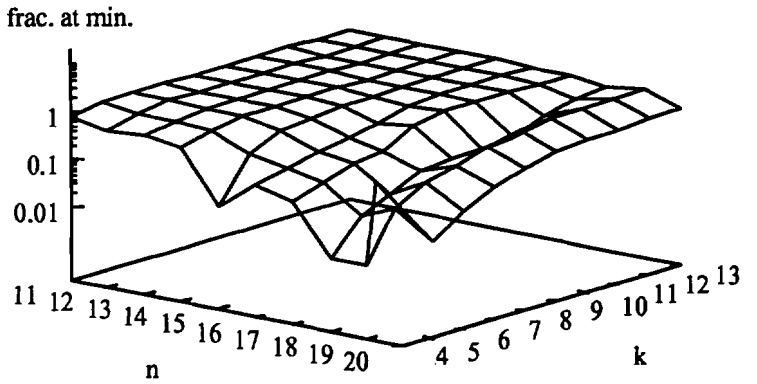
9.3 Division of Plan Space into Watersheds

In this and the next section, we will concern ourselves with the Stochastic Bushwhack algorithm's ability to find true optima for join-optimization problems. As one would expect, a given run of the Stochastic Bushwhack algorithm does not necessarily yield a global optimum; but we will find, for the parameterizations we consider here, that over a sufficient number of runs, the algorithm will always *eventually* find a global optimum for a given join-optimization problem. Naturally we will be interested in knowing how frequently a global optimum is attained. But first we consider a different but closely

[1] The test machine was a lightly loaded SPARCstation 20 with 160MB of main memory running at circa 50MHz under Solaris 2.5.1; the compiler used was Standard ML of New Jersey, version 0.93. Our code for the Stochastic Bushwhack algorithm includes a Standard ML implementation of the Blitzsplit algorithm.



(a) Number of distinct costs obtained in 1000 trials



(b) Fraction of trials that attain minimum cost

Figure 9.1: Number of watersheds, and relative size of optimal watershed

related question: Over a large number of runs, how many different *local minima* does the algorithm yield for a given optimization problem—in other words, how many watersheds are there in that problem's plan space?

In the first set of measurements discussed below, we consider the canonical *cycle* + 3 queries for all combinations of n in the range 11 to 20 and k in the range 4 to 13. (Larger values of n are considered in Section 9.9 below.) Figure 9.1 gives a picture of some basic results regarding the Stochastic Bushwhack algorithm's ability to find global minima. Both graphs in the figure plot values that depend on n and k; n and k vary along the horizontal axes, while the vertical axis represents the dependent value. In Figure 9.1(a),

the dependent value is the number of distinct costs encountered in 1000 *bushwhack* trials—hence the number of distinct watersheds encountered by the 1000 randomly-generated initial plans. The vertical axis, which is on a logarithmic scale, is plotted upside-down. Figure 9.1(b) shows the fraction of the 1000 trials for which *bushwhack* attained a cost that was the global minimum—in other words, the fraction of trials for which *bushwhack* found an optimal plan. We comment further on Figure 9.1(b) in Section 9.4 below.

Figure 9.1(a) is one of several plots that will be displayed with an upside-down vertical axis. The general rule for all the plots is that the "up" direction corresponds to superior outcomes, while the "down" direction corresponds to less desirable outcomes. Thus, the plateau at the top rear of Figure 9.1(a) represents the combinations of n and k that give the best results: For these combinations, where n is relatively small and k is relatively large, the entire space of possible initial plans apparently lies in one large watershed. But as one moves away from this part of the plotted surface (i.e., as n increases and k decreases), the downward slope of the surface reflects the division of the space of initial plans into larger and larger numbers of distinct watersheds.

However, the rise in the number of watersheds is gradual. Considering the many billions of possible plans for the queries under consideration, the division of these plans into roughly ten to one hundred watersheds indicates that on average, each individual watershed must be very large indeed. Recall our observation that the Stochastic Bushwhack algorithm, starting with an initial plan in a particular watershed, transforms that plan into one that is optimal within the watershed. It is in this sense that the Stochastic Bushwhack algorithm finds a "local minimum." Given the size of the watersheds, one sees that these minima are not so local as the term "local minimum" might suggest. When the number of local minima is small, one may expect that just a handful of invocations of *bushwhack* stands a good chance of reaching *all* the local minima—and consequently, of reaching the global minimum as well (since one of the local minima must be the global minimum).

It should be noted that because our watershed counts are based on finite random samples of *bushwhack* runs, we cannot be certain that the counts are reliable. There could be additional watersheds that we have failed to detect. But it is extremely unlikely that these additional watersheds could have escaped our detection unless they occupied only

a very small proportion of plan space. For example, consider a watershed (or collection of watersheds) that represents 1% of the initial plan space. The odds that in 1000 trials we would have failed to encounter any plans in this watershed are $0.99^{1000} \approx 4.3 \cdot 10^{-5}$. Similarly, the odds that we would have failed to detect a watershed occupying 0.5% of the plan space are $0.99^{1000} \approx 0.007$. We may conclude that the portion of plan space not reflected in our measurements probably amounts to well below 1% of the space.

Still, we must be concerned about the possibility that the global minimum might tend to lie in a very small watershed that is difficult to detect. We show next that such situations apparently do not arise in practice.

9.4 Frequency of Attaining Global Minima

To check whether the lowest of the local minima found by *bushwhack* was in fact the global minimum for a given query, we compared the results obtained by *bushwhack* against the results of optimization by exhaustive search. We applied this check to all the test queries discussed in this chapter, with the following exceptions:

- Test queries with $n > 20$ were exempted from this check because of the large amount of computation required to perform exhaustive search.
- Our tests under the disk-nested-loops model were less thorough than the main body of tests carried out under the naive cost model. Verification of optimality was omitted from the disk-nested-loops trials.

In every instance where we computed the global minimum by exhaustive search, the cost so computed coincides with the lowest of the costs obtained in 1000 *bushwhack* trials. In particular, in those cases where our measurements indicate that plan space consists of a single watershed, each run of *bushwhack* yields an optimal plan.

Moreover, in those cases where plan space is divided into multiple watersheds, the watershed that contains the global minimum tends to be *larger*, not smaller, than the others. Figure 9.1(b) illustrates this effect. The vertical axis shows the measured probability that a particular run of *bushwhack* will yield a plan whose cost is the global minimum. The

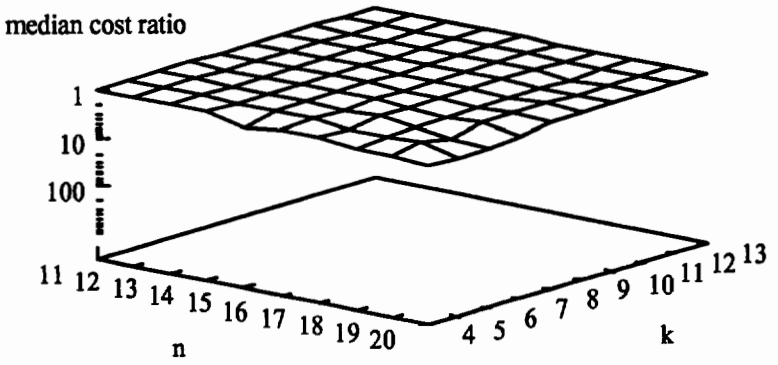
shape of the plotted surface is very similar to the shape seen in Figure 9.1(a); the plateau at the top rear of the surface again shows the good behavior of *bushwhack* when n is relatively small and k is relatively large. Here the interpretation of the plateau is that for the queries in question, a global optimum is obtained with probability 1.

Although the surfaces in the two figures are similar, they are not identical. The plateau in Figure 9.1(b) appears to be slightly larger. The difference reflects the fact that when the plan space divides into d watersheds, the probability that a given run of *bushwhack* will yield the global minimum is nearly always found to be *larger* than $1/d$—and sometimes *much* larger. For example, when $n = 13$ and $k = 4$, the 1000 invocations of *bushwhack* on our test query show that there are at least 15 watersheds, and yet 650 of the 1000 invocations yield an optimal solution. The inferred probability of 0.65 that a random initial plan tree will lie in the optimal watershed is plainly well above 1/15. When we increase k to 6, holding n fixed at 13, the observed number of watersheds drops to 3, while the probability of hitting a global minimum rises to 0.98, which is again well above $1/3 = 0.33$. The bias towards the optimal watershed is not always so pronounced as in these examples, but there is usually some such bias, and it appears to be an extremely rare occurrence that there is significant bias *away from* the optimal watershed.

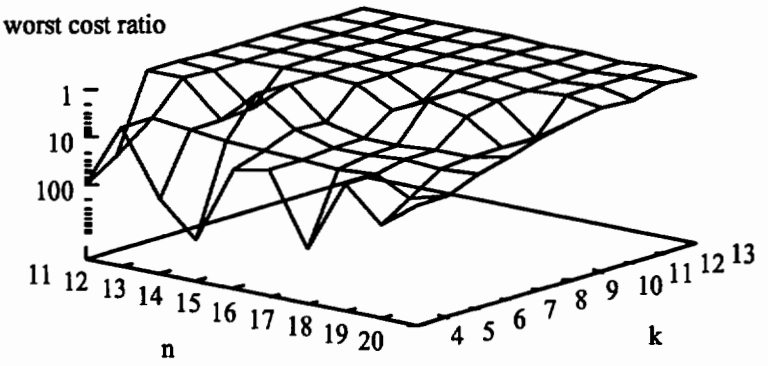
As with the measurements reported in Figure 9.1(a), we again need to be concerned about the relationship between the measured frequency with which a global minimum is found and the actual intrinsic probability of this event. However, we shall not consider this issue in detail here. We simply observe informally that the law of large numbers makes large discrepancies between the measured frequencies and actual probabilities highly unlikely.

9.5 Approximate Optima

The foregoing analysis focused on the division of plan space into watersheds, and on the amount of probability concentrated in the watershed that holds the global minimum. This analysis implicitly assumed that when seeking to optimize a query, we would not be satisfied with any solution short of the global minimum—otherwise we would not have



(a) Median cost ratio



(b) Worst-case cost ratio

Figure 9.2: Goodness of approximate optima, expressed as ratios of plan costs to optimal plan cost

swept aside the other local minima without first stopping to consider *how different* they were from the global minimum. But if it should turn out that the other local minima are all quite close to the global minimum, then holding out for the global minimum alone, and refusing to settle for a close approximation, might be a stance without rational justification.

Figure 9.2 graphs statistics that bear on the acceptability of local minima that are not necessarily global minima. Figure 9.2(a) shows, for each combination of n and k, the *median* cost of the plans returned by 1000 invocations of *bushwhack*, expressed as a ratio to the optimal cost. Analogously, Figure 9.2(b) shows the *worst-case* plan cost over 1000 invocations, again expressed as a ratio to the optimal cost. Once again the vertical axes

are upside-down; thus, the plateau at the top rear of Figure 9.2(b) represents combinations of n and k for which *bushwhack* yields only good plans, while the downward slope closer to the viewer represents combinations where, in the worst case, the cost of a plan returned by *bushwhack* may exceed the true optimum by a large factor.

To understand the significance of the median cost ratio, first consider the meaning of a median in the present context. By the definition of *median*, half of all runs of *bushwhack* (for fixed n and k and a fixed query) yield a cost less than or equal to the median cost, while the other half yield a cost greater than or equal to the median cost. Then the median cost ratio is simply the ratio of the median cost to the global minimum.

In Figure 9.2(a) we see median cost ratios that run very close to unity; the largest median cost ratio, at $n = 20$ and $k = 4$, is 1.74, while most of the other values depicted in the figure are within a few percent of unity. According to Swami's classification, an approximate solution whose cost is within a factor of 2 of the optimum is *good*, and not merely *acceptable*. Thus, with any combination of n and k in the ranges shown in the figure, one may expect to obtain a good solution at least half the time.

A higher probability of attaining a good solution can be achieved through multiple runs of *bushwhack*. Since each individual run yields a good solution with probability at least $1/2$, a succession of j runs will yield a good solution with probability at least $1-1/2^j$. It follows that if we wish to be 99% certain of obtaining a good solution, it suffices to invoke *bushwhack* 7 times—since $1 - 1/2^7 = 1 - 1/128 > 0.99$—and take the best of the 7 solutions it yields.

The strategy just described yields good solutions with high probability regardless of the choice of k. But it offers no guarantees about the quality of the solution in those rare occurrences (i.e., fewer than one in a hundred) where the solution is not good. To get a sense of the solution quality in those cases, we turn to the worst-case cost ratios in Figure 9.2(b).

When k is very small, the worst-case cost ratio is seen to run as high as several hundred. Given the possibility that a local minimum could exceed the optimum by such a large factor, it would probably be inadvisable to take the result of a single run of *bushwhack* to be an "approximate optimum." But as k increases, the worst-case cost ratio rapidly

falls to the single digits. With larger k, then, it might not be unreasonable to treat the result of a single run as an approximate optimum. Recall that Swami defines an *acceptable* solution as one whose cost is within a factor of ten of the optimum; by this definition of "acceptable," most of the points in Figure 9.2(b) represent combinations of n and k for which a single run of *bushwhack* is guaranteed to yield an acceptable solution.

By combining our observations regarding the median cost ratio and the worst-case cost ratio, we can have the best of both worlds: a good solution with very high probability, and an acceptable solution in the residual cases. To obtain a good solution with 99% probability, we invoke *bushwhack* seven times; and to assure that the solution will be acceptable with virtually 100% probability, it is merely necessary to avoid very small values of k in at least one of the seven invocations.

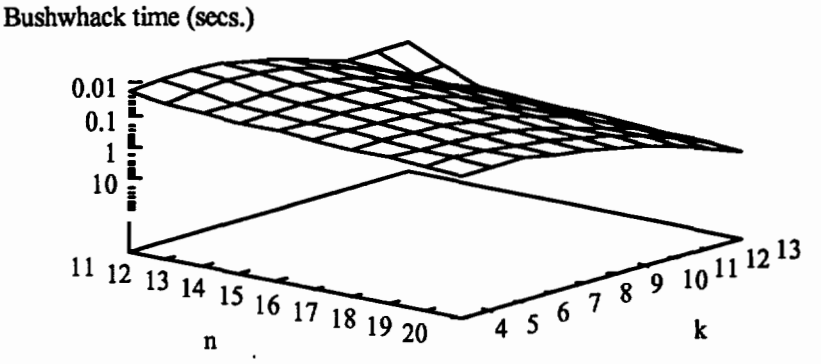
9.6 Optimization Time

In Figures 9.1 and 9.2 we examined the *quality* of the solutions produced by *bushwhack*, without regard for the amount of time it took to produce them. We now consider the *time* needed to run *bushwhack*, without regard for the quality of the solutions obtained. Subsequently we will address both issues together, and consider the trade-off between quality and time.

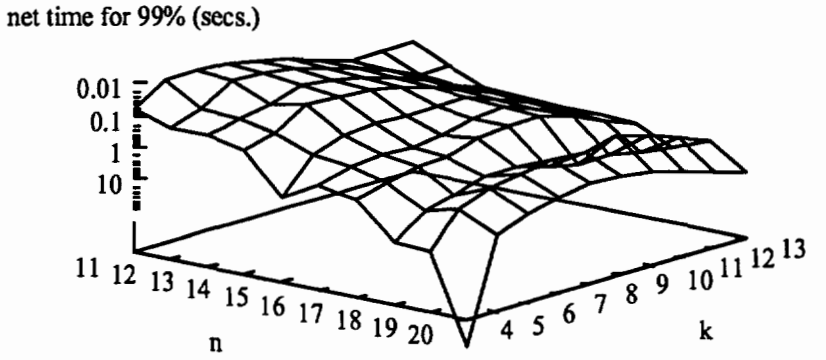
Figure 9.3(a) shows, as a function of n and k, the average CPU time consumed by a single run of *bushwhack* on the canonical *cycle* + 3 queries. Figure 9.3(b) shows the expected aggregate CPU time required to obtain an optimal plan with 99% probability; we shall discuss the latter graph in detail in Section 9.7 below. Here we comment only on the CPU time for individual runs of *bushwhack*.

Because of the stochastic character of *bushwhack*, the execution time of individual runs is variable; however, our measurements indicate that this variability is not open-ended. Only rarely does an individual run require twice the average time, and never (as far as we have seen) does its running time exceed the average by as much as a factor of three.

Our reported timings do not include garbage-collection time. Separate measurements show that the amortized execution time of the garbage collector is typically about 10%



(a) Average CPU time for one invocation of *bushwhack*



(b) Time required to obtain an optimal plan with 99% probability

Figure 9.3: Optimization times for the Stochastic Bushwhack algorithm

to 20% of the time required by *bushwhack* proper. However, in an individual run of *bushwhack*, there may be no garbage-collection activity at all, or, at the other extreme, the garbage-collection time may exceed many times over the *bushwhack* execution time if a major collection is needed.

The surface of Figure 9.3(a) is smooth and uniform except in the far rear corner. In general, execution time consistently rises with both n and k. The dog-ear in the far rear corner arises because optimization time stops increasing when k exceeds n, since *bushwhack* is equivalent to exhaustive search whenever $k \geq n$. (Bear in mind that k is the *maximum* number of pseudo-relations in the tightenings carried out inside *bushwhack*; the number of pseudo-relations may be less than k.)

For the most part, the timings shown here run far below our exhaustive-search timings from Chapter 6—sometimes by as much as three orders of magnitude. However, in a few instances, and notably in the instances where *bushwhack* reduces to exhaustive search, the present timings are an order of magnitude or more *higher*. Exhaustive search takes longer here than in our earlier measurements primarily because here the exhaustive search is carried out by code written in SML, not C. The New Jersey SML compiler generates very good code, but the kind of manipulations performed in our exhaustive-search algorithm do not show SML to best advantage. Performance of our present code also suffers to some degree from a lack of tuning. In light of these factors, the generally favorable timings seen in Figure 9.3(a) are all the more noteworthy.

Perhaps the most striking feature of Figure 9.3(a) is that growth is faster along the k-axis than along the n-axis. Since the vertical axis of the graph is logarithmic, exponential growth rates appear as straight lines. Thus, we see approximately exponential growth along the n-axis; since the times at $n = 20$ are roughly twice the corresponding times at $n = 11$, we may infer that optimization time is roughly proportional to $2^{n/9}$ (i.e., 1.08^n) for n and k in the ranges under study. By contrast, the lines that trace growth along the k-axis are not straight at all, but start out flat and then curve sharply downwards (reflecting a steep *increase* in optimization time). Growth along the k-axis thus appears to be faster than exponential. But inspection of the numbers behind the graph reveals that this appearance is somewhat deceptive—there is actually a knee in the curve. The curve does indeed start out almost flat, and then approaches an asymptote proportional to 3^n. The net effect is that across the entire range from $k = 4$ to $k = 13$, the times grow by roughly a factor of 50.

The likely explanation for the knee in the growth along the k-axis is simply that when k is less than about 8, the time taken by Blitzsplit runs is so small that it is insignificant compared to the time collectively required to identify pseudo-relations, to collapse relations and predicates, to annotate tree nodes, and so forth. But as k rises above 8, the time required to apply exhaustive search to subproblems increases so steeply that it quickly becomes the dominant component of optimization time.

9.7 Quantifying the Quality–Effort Trade-off

Let us now turn to the problem of quantifying the trade-off between quality of optimization and the time required for optimization. We have seen that for fixed n and a fixed query, larger values of k tend to yield better solutions in return for a greater optimization effort. Is the best choice of k simply a matter of judgment, or is there some objective criterion for preferring one k-value over another?

The question may be recast in the following terms. Suppose the objective of optimization is to obtain, with probability at least p, a solution whose cost is within a factor r of the optimum. The appropriate choices of p and r may very well depend on the context in which the optimized query will be used, and so we cannot presume to prescribe the best choices for these variables. But we *can* identify particular instances of p and r that are likely to be of interest. For example, some plausible choices for p might be 0.95, 0.99, 0.995, and so on, and plausible choices for r might be 1, 2, 5, and 10. (Various values of r *between* 1 and 2 would likely also be of interest.) Once p and r have been chosen, it becomes possible to perform an objective comparison of different choices for k.

9.7.1 An Optimization-Effectiveness Index

Now let us assume a fixed query, and let p_k denote, for each k, the probability that a single run of *bushwhack* parameterized by n and k will yield an optimal solution; and let t_k denote the average time required for such a run. (Thus, p_k is the quantity plotted in Figure 9.1(b), and t_k is the quantity plotted in Figure 9.3(a).) To obtain an optimal solution with probability at least p, we must run j_k repetitions of *bushwhack*, where j_k is the least positive integer such that

$$(1 - p_k)^{j_k} \leq 1 - p. \tag{9.1}$$

(The left-hand side is the probability that in j_k runs we will *fail* to find an optimal solution.) The expected total time required for these j_k runs is $T_k = j_k t_k$. Then an objective criterion for choosing k is to take the k for which T_k is least.

In solving for j_k, we first rewrite (9.1) as

$$j_k \ln(1 - p_k) \leq \ln(1 - p), \tag{9.2}$$

and then (reversing the inequality since the logarithms are negative), as

$$j_k \geq \frac{\ln(1 - p)}{\ln(1 - p_k)}. \tag{9.3}$$

The least positive integer j_k satisfying this inequality is

$$j_k = \left\lceil \frac{\ln(1 - p)}{\ln(1 - p_k)} \right\rceil, \tag{9.4}$$

and so the time required to achieve the desired probability p of obtaining an optimal solution is

$$T_k = \left\lceil \frac{\ln(1 - p)}{\ln(1 - p_k)} \right\rceil \cdot t_k. \tag{9.5}$$

To avoid a zero denominator on the right-hand side, an exception must be made when $p_k = 1$; in that case we may simply take $T_k = t_k$. (Alternatively, and with the same effect, we may replace a measured probability p_k equal to 1 with a presumed true probability of $p_k = 1 - \epsilon$ for some suitable small positive ϵ.)

9.7.2 Attaining an Optimum with 99% Probability

Here we examine the case where $p = 0.99$ and $r = 1$. In other words, for a given n, we seek the value of k that minimizes the effort that must be expended to find a plan for our test query such that the cost of this plan is optimal with probability at least 0.99.

Figure 9.3(b) plots, as a function of n and k, the values of T_k for our canonical *cycle*+3 queries, given the objective of finding a minimum with 99% probability. The arched shape of the surface shows that the effectiveness of *bushwhack* declines if k is either too small or too large. We have seen before that when k is too small, the probability of attaining a minimum becomes very low; on the other hand, when k is too large, the time required to execute *bushwhack* becomes large. The best compromise appears to be found when k is in the neighborhood of 6 to 9. Within this range, the smaller values of k appear to be more appropriate for smaller n, while the larger k are better suited to the larger n.

For example, at $n = 11$, we can obtain the value $T_k = 0.016$ sec by taking $k = 6$; at $n = 15$, we obtain $T_k = 0.14$ sec when $k = 7$; at $n = 20$, we obtain $T_k = 1.6$ sec when $k = 9$. Note, though, that the arch is rather flat at its high point; using a choice of k that is slightly too large or slightly too small does not greatly affect the value of T_k obtained.

9.7.3 The *Recursive* Bushwhack Algorithm and the "Kick"

Two aspects of the results just reported motivate possible improvements to the Stochastic Bushwhack algorithm. These improvements are likely to be especially important as we consider larger queries.

First, it appears, not surprisingly, that the value of k best suited to a given value n is roughly proportional to n. Thus, it may make more sense to parameterize the algorithm by the *ratio* of k to n than by k itself. In our studies of queries of larger numbers of relations, we shall use such a parameterization, as illustrated in Figure 9.4. The parameter *k-pct* determines k as a percentage of n. In the figure, *k-pct* varies from 24 to 60, so that, for example, when $n = 21$, k effectively varies from 5 to 13, and when $n = 30$, k varies from 7 to 18.

Second, the times needed to obtain an optimal plan with 99% probability, as illustrated above, are dramatically lower than the times needed to perform exhaustive search on problems of the same size. It is therefore tempting to make the Stochastic Bushwhack algorithm *recursive* in the following sense. Where *tighten* calls an exhaustive-search optimizer to optimize a subproblem, it can be made instead to issue a recursive call (or a series of such calls) to the Stochastic Bushwhack algorithm. With this change, we are no longer guaranteed an optimal solution to the subproblem. But we can expect to obtain a high-quality solution for a greatly reduced effort, and so there is a good chance we will come out ahead. We shall refer to this recursive variant of the algorithm as the *Recursive Bushwhack* algorithm.

Needless to say, if we wish to make the Stochastic Bushwhack algorithm recursive, we must ensure that the recursion is well-founded. That is, we must identify a base case that is not recursive, and ensure that the recursive calls eventually work their way down to the base case. We meet these requirements of well-foundedness as follows. For the base case,

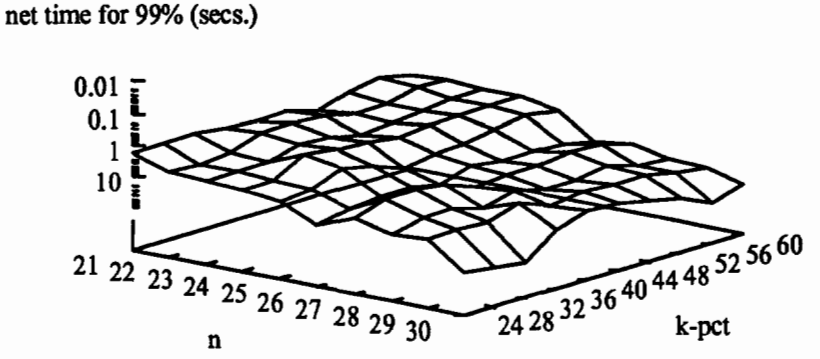


Figure 9.4: Time to obtain minimum cost with 99% probability, as function of n and *k-pct*

we adopt a somewhat arbitrary policy: *tighten* will solve a k-way subproblem by exhaustive search whenever $k \leq 10$, but otherwise by a recursive call to *bushwhack*. To ensure that we eventually obtain subproblems involving no more than 10 pseudo-relations, we rely on a parameterization of the algorithm by *k-pct* rather than k. For example, if $n = 30$ and *k-pct* $= 60$, then we obtain subproblems of size $30 \cdot 0.60 = 18$; in the recursive invocations of *bushwhack*, we thus have $n = 18$, and we obtain subproblems of size $18 * 0.60 \approx 11$; finally, at the third level of recursion, we have $n = 11$, and the resultant subproblems of size $11 \cdot 0.60 \approx 7$ are solved by exhaustive search.

On a philosophical level, the recursive strategy may come across as the moral equivalent of a perpetual-motion machine. Is it not an attempt to get something for nothing? When $n = 30$, is there any reason to expect we should get better results from a recursive algorithm with *k-pct* $= 60$, than from a non-recursive algorithm with k fixed at 7? In either case, we will be breaking the optimization problem down to 7-way subproblems, and performing exhaustive search only on these 7-way subproblems. What does the recursive strategy do that the non-recursive strategy does not do?

The sequence of 7-way subproblems that are solved may be slightly different in the two cases, but probably the more important difference is that the recursive strategy will be injecting additional randomness into the tightening process. Each recursive call to *bushwhack* will bring with it a call to *rand_tree*, which in the case $n = 30$ will randomly rearrange the positions of up to 18 relations or pseudo-relations. The immediate effect of

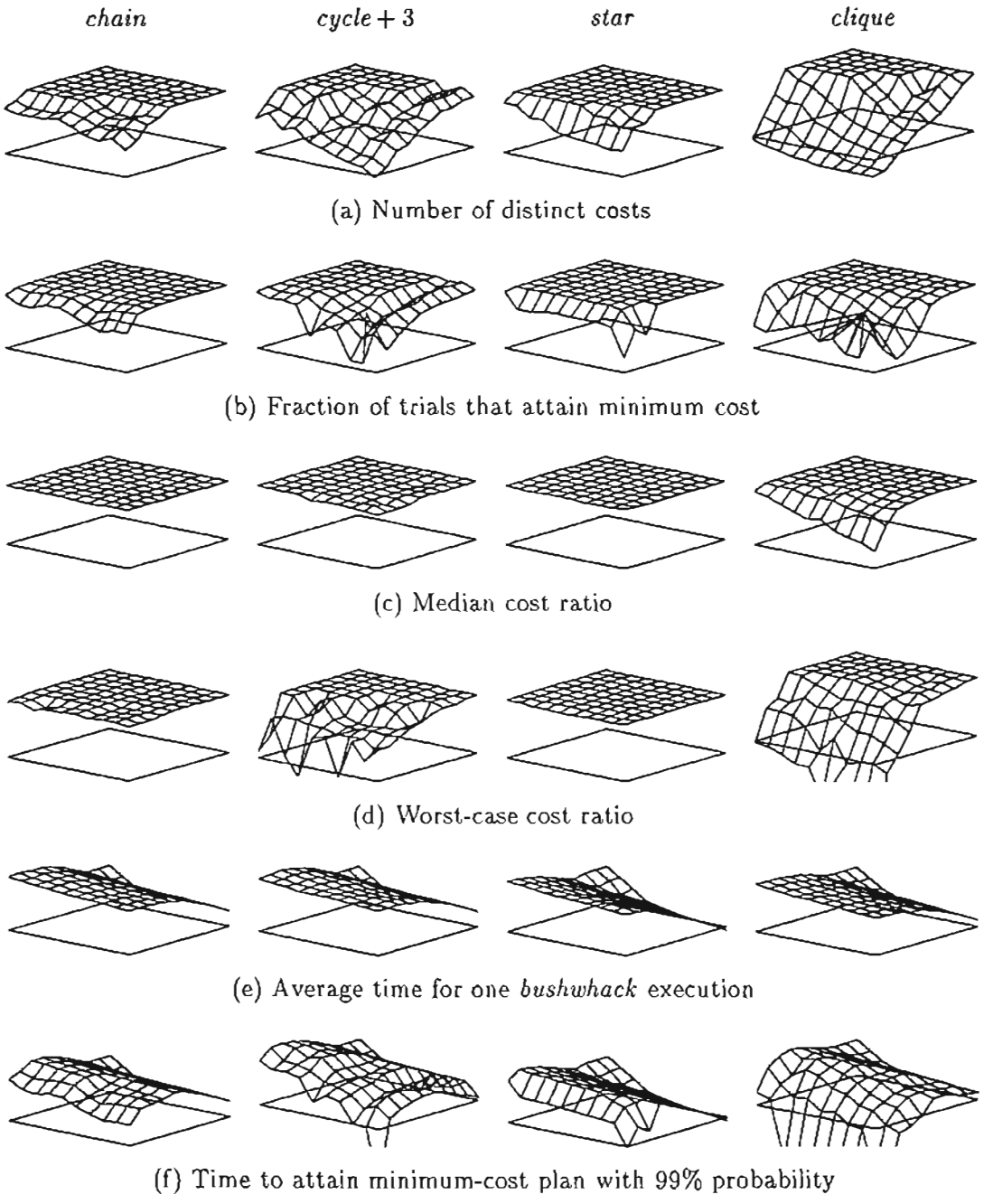
these rearrangements will usually be to degrade the plan as a whole; but they will also create opportunities for descending to new local minima that have not yet been explored, and that would not be reachable by a sequence of 7-way tightenings. In effect, the recursive calls provide the "kick" that is called for in the Chained Local Optimization technique of Martin and Otto [39] (*cf.* Section 8.1). Later in this chapter we shall present experimental results on the Recursive Bushwhack algorithm's performance.

9.8 Varying the Join Graph

Up to this point we have been examining the behavior of the Stochastic Bushwhack algorithm only for queries of the *cycle*+3 topology. In this section, we repeat the experiments of Figures 9.1, 9.2, and 9.3 for several other topologies as well. For compatibility with the earlier experiments, we continue to use the non-recursive Stochastic Bushwhack algorithm for the present. In Section 9.9, we will extend these experiments to larger joins; at that point we will switch to the Recursive Bushwhack algorithm, as the non-recursive version becomes prohibitively expensive.

Figure 9.5 gives twenty-four graphs, arranged in six rows and four columns. The six rows correspond to the six graphs that were presented in Figures 9.1, 9.2, and 9.3 for the case of the canonical *cycle* + 3 queries. Within each row of Figure 9.5, analogous information is presented for each of four distinct join-graph topologies, namely the *chain*, *cycle* + 3, *star*, and *clique*. Thus, one may scan down any column of the figure to see a summary of the behavior of the optimizer for any one of the topologies. In particular, the second column (the "*cycle* + 3" column) repeats, in reduced format, the graphs of Figures 9.1, 9.2, and 9.3.

The surfaces for the *chain* and *star* are generally flatter than for the *cycle* + 3 and *clique* topologies. This flatness reflects the fact that the chain and star topologies are acyclic, and apparently queries drawn from these topologies are easier to optimize that queries drawn from cyclic topologies. However, the star presents a difficulty that is absent in the other topologies. In the star case, the time required for *bushwhack* execution rises very fast as k increases. The probable reason for this effect is that optimal plans for stars



(a) Number of distinct costs

(b) Fraction of trials that attain minimum cost

(c) Median cost ratio

(d) Worst-case cost ratio

(e) Average time for one *bushwhack* execution

(f) Time to attain minimum-cost plan with 99% probability

Figure 9.5: Profile of Bushwhack behavior for joins of 11 to 20 relations, with k from 4 to 13 (canonical test queries)

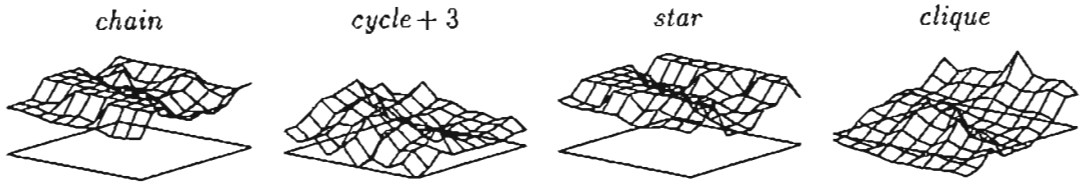
tend to be left-deep, hence deeper than bushy plans with the same number of leaves; the recursion depth of *shake_up* and *shake_down* is consequently greater as well, and a larger number of tightenings are performed on each iteration of *itshake*.

Figure 9.5(f) shows differences among the arches for the different topologies. Recall that the top of the arch represents values of k for which optimization is most effective, in the sense of yielding the best plans for a given optimization effort. Evidently the most effective k values for the easier topologies are lower than for the more difficult topologies.

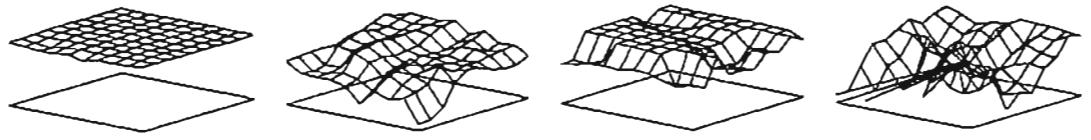
9.9 Larger Numbers of Relations

Now we consider larger values of n. Figure 9.6 profiles *bushwhack* behavior for joins of 21 to 30 relations. The overall organization of this figure is exactly the same as that of Figure 9.5. However, here we base our measurements on the *Recursive* Bushwhack algorithm, because of the huge expense of running the non-recursive version on larger joins. Accordingly, the right-hand horizontal axis in each of the present plots represents *k-pct* (as in Figure 9.4) rather than k.

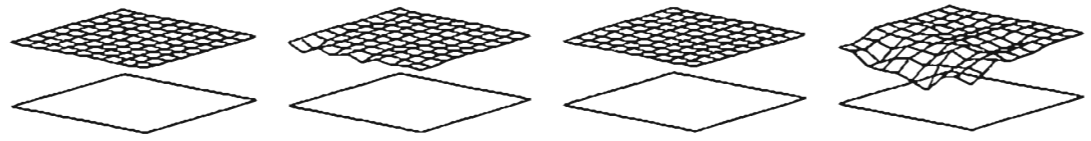
The present plots tend to be somewhat bumpier than those for the smaller joins, but otherwise they have much the same character. Because of the large sizes of the present joins, it was not feasible to verify by exhaustive search that the lowest minima found for each query were in fact the true global minima. However, extrapolation from the evidence obtained for joins of 11 to 20 relations suggests that here, too, the lowest minimum found in 1000 trials is likely to be the global minimum. For the *cycle*+3 queries, for example, we typically obtain a few tens of distinct costs, each of which accounts for several percent of plan space on average. By the same reasoning we applied in Section 9.3 above, any costs that may have been overlooked probably represent well under 1% of plan space. So it is unlikely that we are overlooking the true global optimum unless it occupies a watershed of unusually low probability—which is itself unlikely, since we have noted that, if anything, the watershed containing the global optimum tends to occupy a disproportionately *large* chunk of plan space. That tendency showed no sign of dropping off as n increased. We may expect the same to be true here. Indeed, we have further circumstantial support for



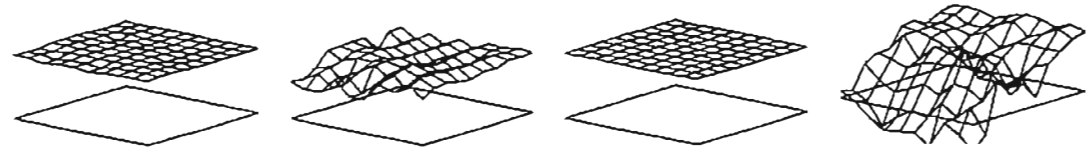
(a) Number of distinct costs



(b) Fraction of trials that attain minimum cost

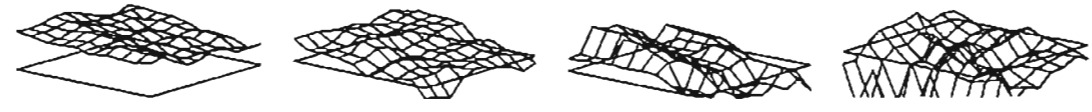


(c) Median cost ratio



(d) Worst-case cost ratio

(e) Average time for one *bushwhack* execution



(f) Time to attain minimum-cost plan with 99% probability

Figure 9.6: Profile of Recursive Bushwhack behavior for joins of 21 to 30 relations, with *k-pct* from 24 to 60 (canonical test queries)

this expectation in the fact that the fraction of the present trials that attain the minimum (for the *cycle* + 3 topology) is typically 1/10 to 1/2.

One may object that any extrapolation from the smaller joins is suspect, since our earlier observations did not involve recursion in the Bushwhack algorithm, which could subtly alter its behavioral characteristics. However, note that there is overlap between the non-recursive and recursive versions of the algorithm. For example, when $n = 21$ and *k-pct* is anywhere up to about 50, the recursive version is equivalent to the non-recursive version with k ranging up to 10. There is some overlap at larger n as well, although the amount of overlap shrinks as n increases. When $n = 30$, the overlap involves values of *k-pct* up to about 34, which again correspond to values of k up to 10. Thus, there is a triangular area in each of the plots where recursion does not come into play; this area in each plot includes the corner that lies leftmost on the printed page. In several of the star-query plots, and to some extent in all the plots of Figure 9.6, one observes a discontinuity at the edge of this triangle, where recursion kicks in. But in general the transition is fairly smooth, and suggests that the recursive version of the algorithm does not behave in a fundamentally different way from the non-recursive version.

The most encouraging information in these plots is the cost ratios in Figures 9.6(c) and (d). The median cost ratios are all quite good—never above 2 except in the case of the clique; and the same can be said for the clique as well if the smaller values of *k-pct* are avoided. Even the worst-case cost ratios are quite moderate—excluding the clique, the *worst* among them run to about 15.

The timings in Figure 9.6(f) are also quite good, but it is hard to get a sense of these timings from the scaleless plots in the figure. Figure 9.7 plots, for the canonical test queries of each topology, the amount of time needed to obtain a minimum-cost plan with 99% probability, as a function of the number of relations in the query. What appears to be the best choice of *k-pct* is applied in each case. Thus, for the chain and star queries, $k\text{-}pct = 32$; for the *cycle* + 3 queries, $k\text{-}pct = 44$; and the for the clique queries, $k\text{-}pct = 60$. The timings shown are hundreds to thousands of times lower than the timings reported by Steinbrunn [54] for stochastic join-optimization techniques applied to joins of up to

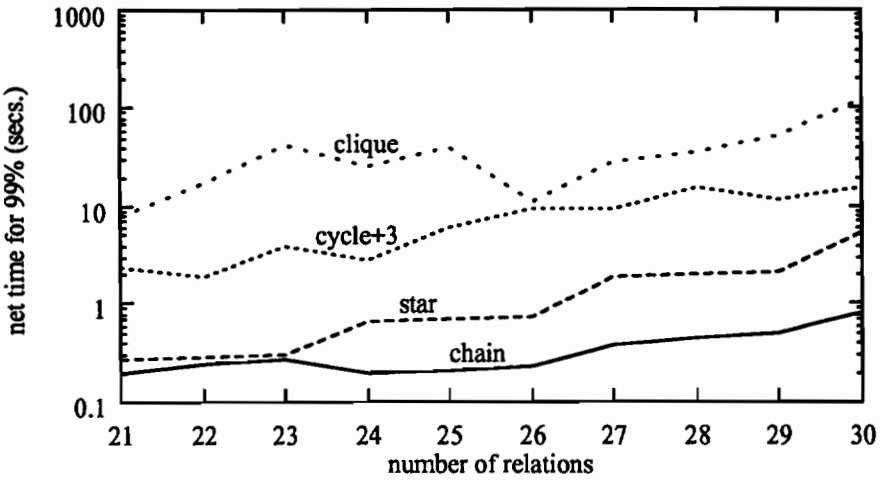


Figure 9.7: Time needed to obtain a minimal plan with 99% probability (canonical test queries)

30 relations. Moreover, the evidence presented above, combined with Steinbrunn's plan-quality analyses, suggests that the solutions we obtain by our technique are likely to be at least as good as those obtained by other techniques, and probably better in many instances.

Our timings may be unrealistically optimistic for a couple of reasons, over and above the fact that we are using an overly simplistic cost model:

- These timings assume that the algorithm is run with an appropriate choice of *k-pct*; determining an appropriate *k-pct* for an arbitrary query may not be easy. (One can safely use a value of *k-pct* that is too large, but optimization time may then rise by an order of magnitude.)

- The 99% probability of obtaining a minimum-cost solution is based on executing some number of *bushwhack* iterations that depends on the algorithm's behavior for the query at hand. Determining the appropriate number of iterations for an arbitrary query may not be easy.

Nonetheless, the fact that these timings are possible at all gives an indication of the power of the Recursive Bushwhack algorithm. Further study may reveal good, straightforward

heuristics for choosing *k-pct* and the number of *bushwhack* iterations.

9.10 Varying the Queries

All of our measurements up to this point have involved our so-called *canonical* test queries, each with mean base-relation cardinality 10^4 and variability 0.5. Here we explore the extent to which the results we obtained for that parameterization generalize to other parameterizations.

Figure 9.8 shows, in analogy to Figure 9.1(a), the number of distinct costs for 20-way *cycle* + 3 queries with different cardinality parameterizations. The plot is based on runs of *bushwhack* with $k = 8$. (Equivalent results would be obtained by running the Recursive Bushwhack algorithm with *k-pct* = 40.) The horizontal axes of the figure are as in Figure 6.1 on page 153; but relative to the plots of that figure, the present plot is rotated about the vertical axis. Thus, the *mean cardinality* axis here is the shorter of the horizontal axes, and the *variability* axis is the longer of the two.

The reason for this change in perspective becomes evident when one examines the array of plots in Figure 9.9, each of which is parameterized the same way as Figure 9.8. Whereas most of the variation in Figure 6.1 occurred along the *mean cardinality* axis, here there is greater variation along the *variability* axis. However, mean cardinality can make a big difference here, too. In particular, queries with mean cardinality 1 are the *easiest* to optimize under the Stochastic Bushwhack algorithm. The most problematical queries are those involving extremely large cardinalities.

Large variabilities seem to present a greater challenge than small variabilities except in the case of the clique. The reasons for the observed effects of variability are unknown to the author, and remain to be investigated in the future. In general, the effects are not extremely large; it would appear that the Stochastic Bushwhack algorithm is effective for a broad range of queries, and not just for our so-called canonical queries. It is possible that the poor performance we see here for some of the clique queries would be ameliorated by using a larger value for k or *k-pct*; again, this possibility remains to be investigated.

Though there is variety among the queries represented in Figure 9.9, the variety is of a

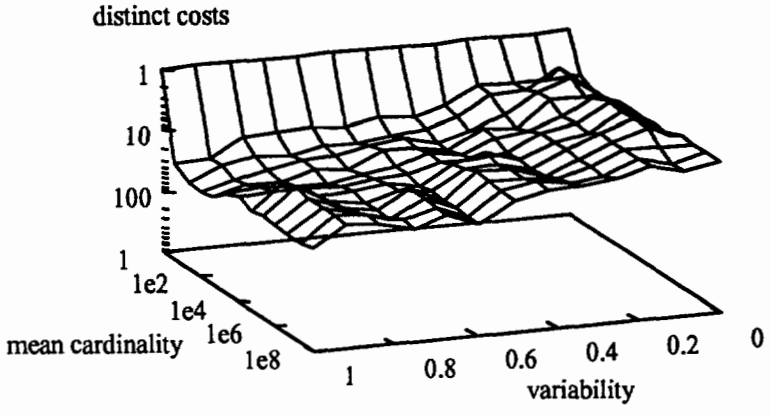


Figure 9.8: Number of distinct minima as a function of mean cardinality and variability ($n = 20, k = 8$)

restricted kind. The cardinalities in all these queries are equally spaced on a logarithmic scale, which raises the question of whether the regularity of the query construction somehow gives the Stochastic Bushwhack algorithm a better handle on the queries. (Such an effect would have been implausible in the case of the deterministic Blitzsplit algorithm, but in the present situation is difficult to reject out of hand.) To explore the possibility of such an effect, below we repeat the experiments of Figure 9.9 using queries in which the cardinalities and selectivities have been subjected to random perturbations.

We perturb the queries by multiplying each cardinality by 10^X and by raising each selectivity value to the power $1+X/5$, where X is a uniformly distributed random variable drawn from the interval $[-0.5, 0.5]$. (A different X is used to perturb each cardinality and each selectivity.) These perturbations result in very uneven spacing of the cardinalities, and reduce the tendency for the selectivities to cancel out the cardinalities. Figure 9.10 shows the behavior observed in the presence of these perturbations.[2,3] Not surprisingly, the plots show chaotic-looking variation. But the overall outlines of these plots are the

[2]The plots in Figures 9.10, 9.11, and 9.12 are based on 100 rather than 1000 trials per query.

[3]In the surfaces in Figure 9.10, each point reflects 100 applications of ***bushwhack*** to the ***same*** perturbed query. Figure 9.10 differs from our other figures in that the details of the function surfaces in Figure 9.10 depend on the random number generator used to generate the random variable X. The tests represented in the other figures also involve random number generation, but in the other figures, the effect of using a different generator would presumably be negligible.

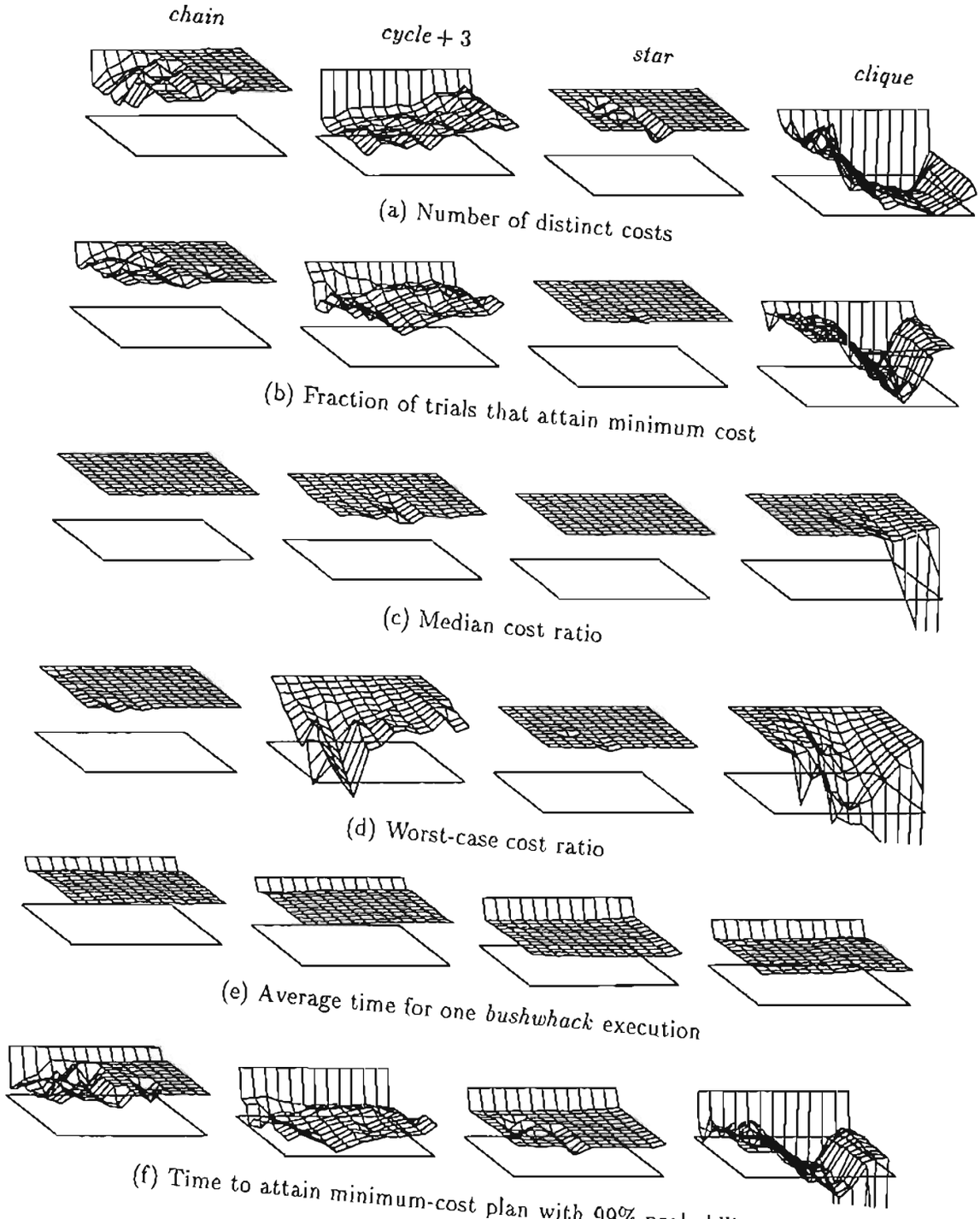


Figure 9.9: Profile of Bushwhack behavior as a function of mean cardinality and variability ($n = 20, k = 8$)

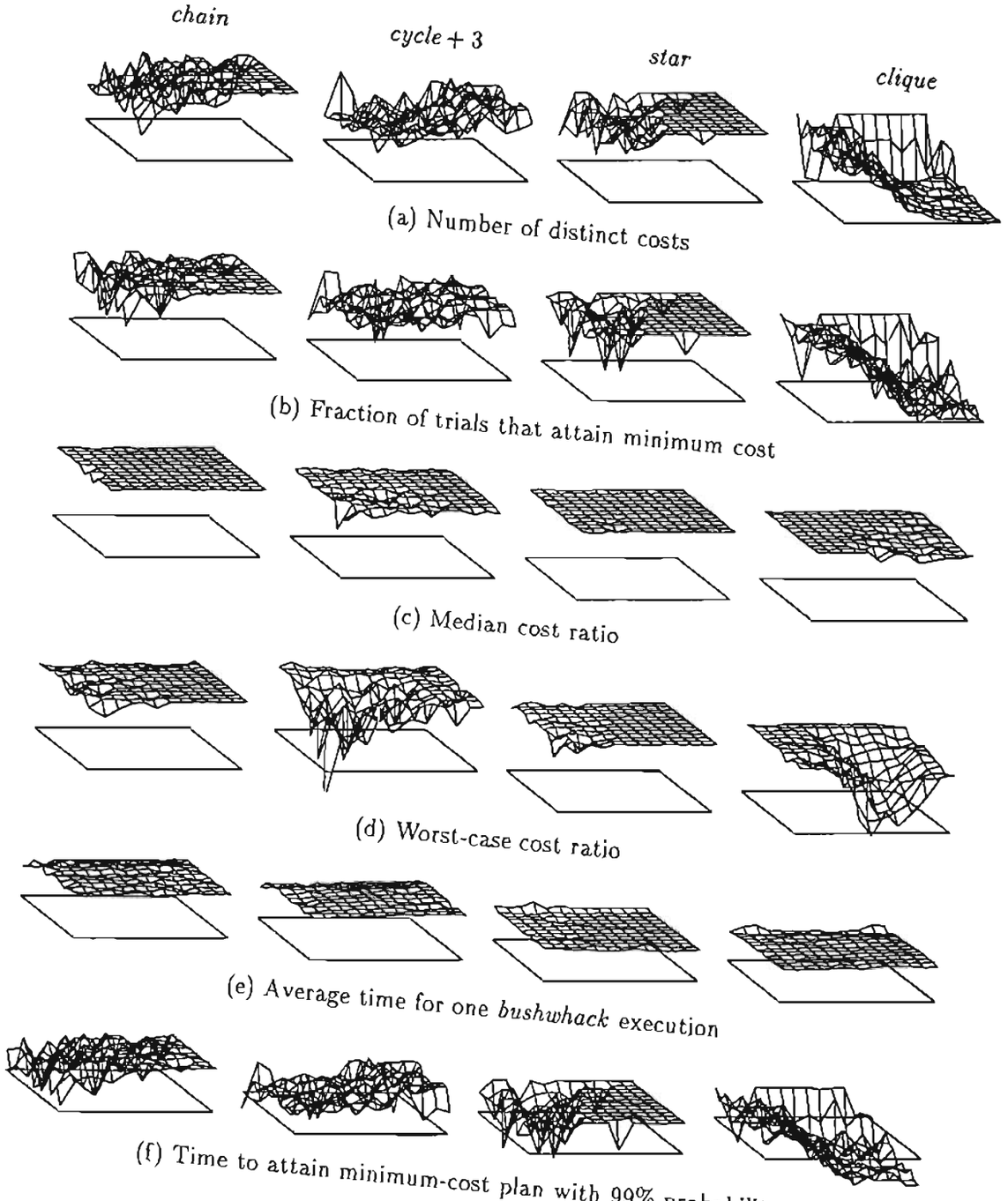


Figure 9.10: Profile of Bushwhack behavior as a function of mean cardinality and variability, with perturbations ($n = 20, k = 8$)

same as those of Figure 9.9. Evidently the behavior of the Stochastic Bushwhack algorithm does not depend critically on any particular spacing of the base-relation cardinalities or on any special relationship between these cardinalities and the predicate selectivities.

9.11 Varying the Cost Model

We have accumulated substantial evidence showing the Stochastic Bushwhack algorithm to be effective for a variety of queries. But all of this evidence has been obtained using the naive cost function κ_0. Do the observed results carry over to other cost models?

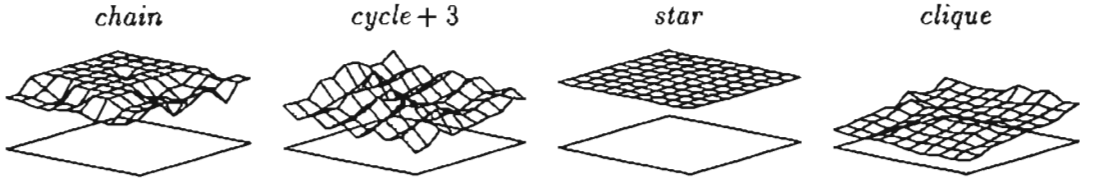
Figures 9.11 and 9.12 give a partial answer to this question. Based on measurements using the disk-nested-loops cost function κ_{dnl}, these figures show behaviors that are generally comparable to those seen in the corresponding figures for the naive cost function (Figures 9.6 and 9.9). The plots in Figures 9.11 and 9.12 are parameterized in the same way as those in Figures 9.6 and 9.9. Thus, in Figure 9.11, the horizontal axes in each plot represent n and *k-pct*, as first illustrated in Figure 9.4 on page 239. By contrast, in Figure 9.12 the horizontal axes represent *mean cardinality* and *variability*, as illustrated in Figure 9.8 on page 247.

In some respects behavior under the disk-nested-loops model is better, and in other respects worse, than under the naive model. Notably, clique queries seem to cause more trouble under the disk-nested-loops model, especially at large cardinalities. But as in the case of the naive model, it may be possible to obtain better behavior for cliques by using larger values of k or *k-pct*.

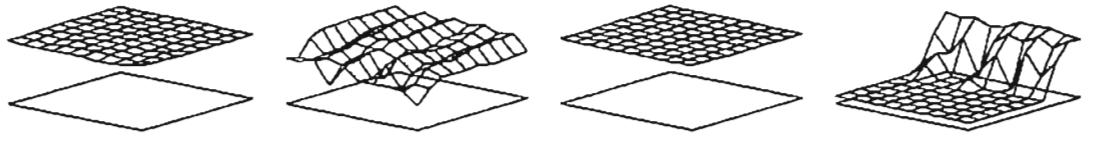
Overall, the evidence of Figures 9.11 and 9.12 suggests that behavior of the Stochastic Bushwhack algorithm is not critically dependent on the particulars of the problem formulation.

9.12 Summary and Discussion

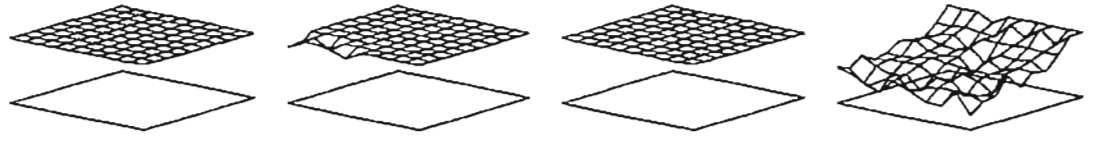
In this chapter we have conducted empirical studies of the behavior of the Stochastic Bushwhack algorithm. Partly on the basis of our preliminary results for queries of 11 to



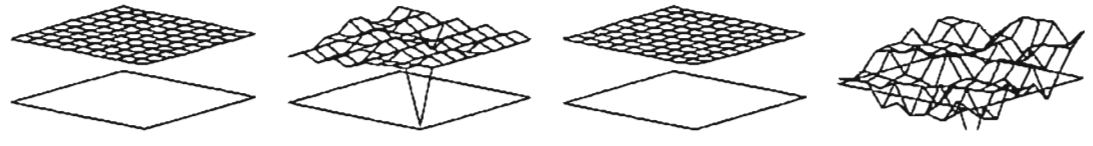
(a) Number of distinct costs



(b) Fraction of trials that attain minimum cost

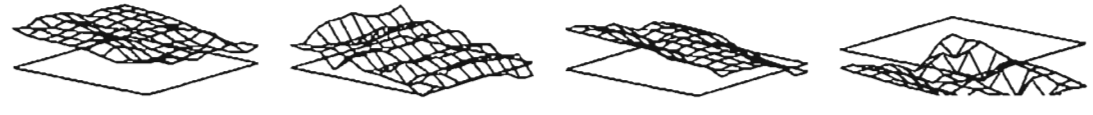


(c) Median cost ratio



(d) Worst-case cost ratio

(e) Average time for one *bushwhack* execution



(f) Time to attain minimum-cost plan with 99% probability

Figure 9.11: Profile of Recursive Bushwhack behavior for joins of 21 to 30 relations, with *k-pct* from 24 to 60 (canonical test queries, disk-nested-loops cost model)

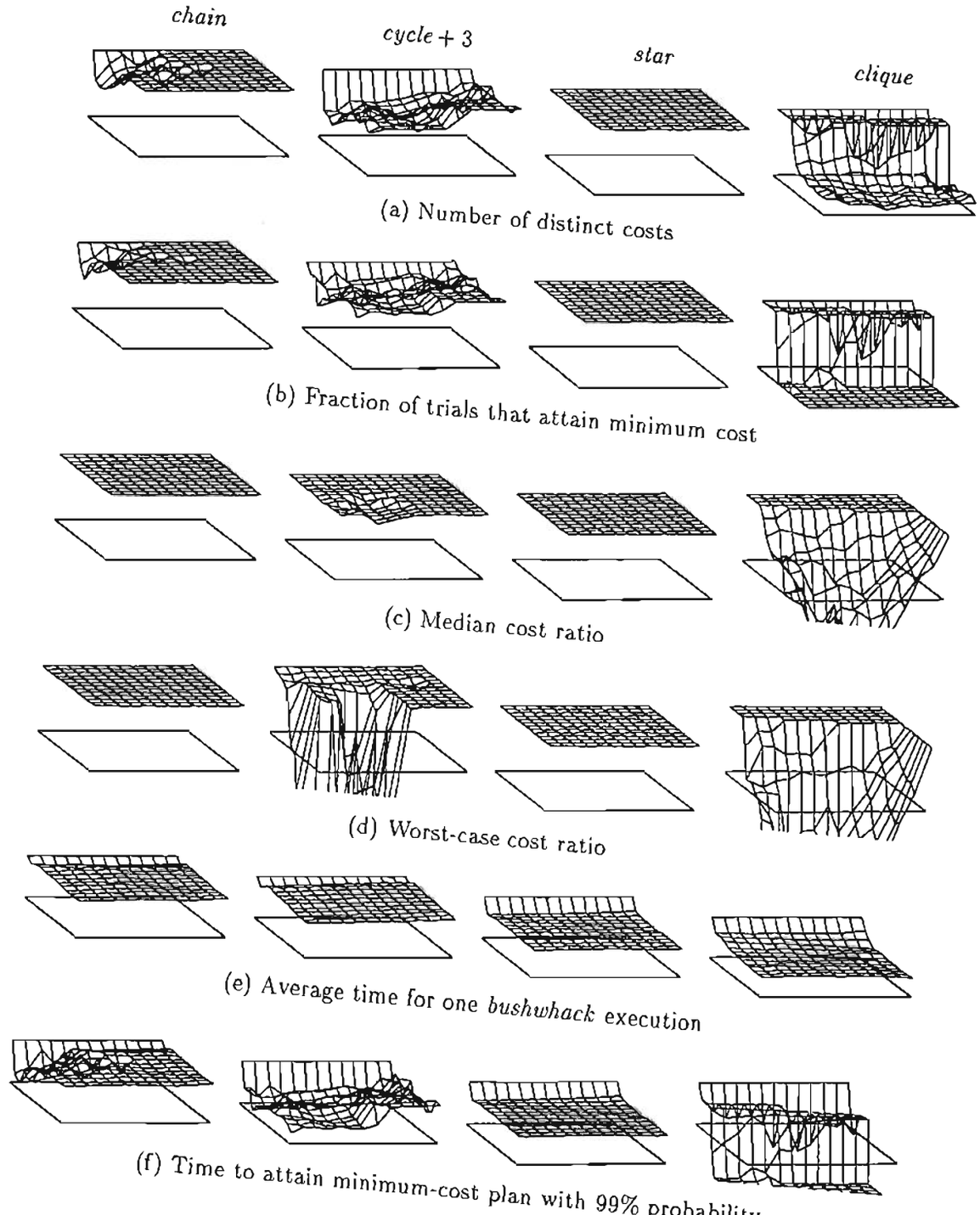


Figure 9.12: Profile of Bushwhack behavior as a function of mean cardinality and variability ($n = 20, k = 8$, disk-nested-loops cost model)

20 relations, we formulated the Recursive Bushwhack algorithm, which effectively incorporates a "kick" of the kind called for in the Chained Local Optimization technique of Martin and Otto. We then used the Recursive Bushwhack algorithm in our subsequent measurements involving queries of 21 to 30 relations.

Our measurements assess both optimization time and the quality of the solutions obtained. The evidence suggests, though not conclusively, that with iterated runs of the Recursive Bushwhack algorithm it is possible with near certainty to obtain optimal plans for join queries of at least 30 relations. The time required to obtain these plans varies with the join graph, but generally runs to seconds, not minutes, of CPU time. As such, our optimization times appear to be at least two orders of magnitude lower than those reported in Steinbrunn's survey of stochastic join-optimization techniques [54].

Our results must be treated with caution because of the simplicity of our cost models, and because of our omission of consideration of physical properties. Optimization times can be expected to increase when more realistic cost models are used and when physical properties are factored in; it is also possible that the quality of the plans obtained will decline.

However, there are intuitive grounds for believing that plan quality will remain high when different cost models are used. One would expect that in typical cases, a query plan that is good under one cost model ought to be tolerably good under other models as well; for regardless of the cost model, good plans must avoid generating large intermediate results. To a large extent, join-order optimization is a matter of balancing the cardinalities generated at one node against those generated at other nodes, and ensuring that no one node assumes too great a burden. Different cost models may assign different weights to the cardinalities involved, but one may conjecture that the consequences of these differences are likely to be relatively localized within the plan tree.

The same intuition applies to physical properties as well. For example, the sort-order of a relation may affect the cost of the join node that is its immediate parent in the plan tree, and perhaps also that of its grandparent or great-grandparent; but it is unlikely to have repercussions at far-distant nodes. Only in "borderline" cases would one expect that consideration of physical properties would justify changing the macroscopic structure of a

plan tree that was optimal under a simpler set of assumptions.

We leave validation (or refutation) of these conjectures to the future. Inasmuch as they are correct, they suggest a means of avoiding a large increase in optimization time with the introduction of sophisticated cost models and physical properties. One could first optimize using a simple cost model and without regard to physical properties, and then iteratively *tighten* the resultant plan, using the sophisticated cost model and taking physical properties into account. Such a two-step approach would be reminiscent of a proposal by Swami and Iyer [59], in which a tentative plan is first obtained under one set of assumptions, and then locally improved (if possible) by considering changes to those assumptions. Other hybrids are also imaginable. For example, the simulated-annealing phase of two-phase optimization [26] could be replaced by iterated tightening; genetic algorithms [54] could be used to recombine plans obtained in successive *bushwhack* iterations; and so on.

Despite the apparent similarity of the Stochastic Bushwhack algorithm to Swami's local-improvement technique [56], performance of the Stochastic Bushwhack algorithm—and for larger queries, the Recursive Bushwhack algorithm—seems to be far more satisfactory. Possible explanations for the performance differences lie in the following differences between the present study and Swami's:

- The Stochastic Bushwhack algorithm takes advantage of the fast dynamic programming provided by the Blitzsplit algorithm.
- The "kick" in the Recursive Bushwhack algorithm may be essential to obtain good performance for larger queries.
- Swami based his comparisons of join-optimization techniques on extremely large queries involving up to 100 relations each. It is possible that, even with the "kick," the Recursive Bushwhack algorithm is not effective for such large queries.
- Swami's studies considered only left-deep plans, and excluded plans with Cartesian products. It is conceivable that allowing Cartesian products in the initial join-processing tree speeds up convergence of *bushwhack*, even if the optimal plan does

not contain Cartesian products.

In connection with the last point, it is interesting to note that at large values of *k-pct*, the Recursive Bushwhack algorithm yielded its longest running times when presented with star queries. The optimal plans for star queries are typically left-deep; the clear implication—especially in light of the observations of Section 9.8—is that the Recursive Bushwhack algorithm works faster on bushy plans than on left-deep plans.

Changes to the algorithm might be able to ameliorate performance on left-deep plans. Our implementation discards all the dynamic programming tables constructed in subproblem optimization; retention of these tables could speed construction of subsequent tables. The savings would probably be slight for bushy trees, but might be substantial in the case of left-deep trees (since successive tightenings in the left-deep case involve many of the same relations).

Chapter 10

Conclusion

The foregoing chapters have presented evidence in support of the thesis that consideration of all Cartesian products in join-order optimization need not be prohibitively expensive. In both the deterministic Blitzsplit algorithm and the Stochastic Bushwhack algorithm, we consider all join orders without prejudice against Cartesian products. Yet our performance measurements show that under simple cost models both these algorithms outperform their more conventional counterparts by a wide margin. This margin may shrink under more complicated cost models, but based on our performance analysis (including observations on cost-function execution counts), there is reason to believe that our approach will continue to hold an advantage.

Consideration of Cartesian products is affordable because enumeration of query-plan alternatives is not the limiting factor in join-order optimization. The limiting factor appears to be the cost computations; and because the cost computations required for Cartesian products are generally negligible, there is little to be gained from their exclusion.

Progress beyond the point reached in the present work will depend on identifying good ways of containing the cost computations under realistic query-optimization conditions. Below we touch briefly on two issues related to keeping the cost computations manageable—the accommodation of physical properties, and the question of whether bottom-up or top-down dynamic programming is better suited to the avoidance of cost computations for useless subplans. Both issues are ripe areas for future work.

We also briefly address the question of extending the present work to non-relational database systems—specifically, to object-oriented systems.

10.1 Physical Properties

Suppose, as in Section 2.6.3, that the base relations of a database are distributed between machines in Tokyo and Kyoto, and that each join of a query plan may execute at either Tokyo or Kyoto. To accommodate this situation in the Blitzsplit algorithm, we must first increase the number of columns in the dynamic programming table. Instead of just having one Best Split and one Cost column, we need two of each: a Tokyo Best Split, a Tokyo Cost, a Kyoto Best Split, and a Kyoto Cost. That is, the best plan for computing the join in Tokyo will be different from the best plan for computing the join in Kyoto, and each of these best plans will have its own cost.

Now in the loop in *find_best_split*, there is more work to do than before. For a given split, there are four possibilities to consider: both the left- and right-hand sides are computed at Tokyo; both are computed at Kyoto; the left-hand side is computed at Tokyo, but the right-hand side at Kyoto; or vice versa. Each combination constitutes a different plan for computing the join in question, and each such plan must be considered as a candidate Tokyo Best Split and as a candidate Kyoto Best Split.

In the general case, the database may be distributed across b_0 cities, for some b_0. In addition, each relation and each join result may be sorted according to any of b_1 sort orders, for some b_1. There may be additional physical properties as well, giving, respectively, b_2, $\ldots$, b_{M-1} alternatives, where M is the number of properties (including location and sort order). Then on the face of it, the dynamic programming table must have $\prod_{i=0}^{M-1} b_i$ Best Split columns, and an equal number of Cost columns, to accommodate all the different combinations of the physical properties. What is worse, each iteration of the loop in *find_best_split* must consider $\left(\prod_{i=0}^{M-1} b_i\right)^2$ combinations of left-hand sides and right-hand sides.

But these horrible factors assume a naive implementation. Presumably it is possible to do better, though just how much better is not yet clear. The key to beating the naive implementation is to observe that the plans for different physical properties have a great deal in common, and that portions of the cost analysis for the different alternatives can be shared among them.

In particular, one can include in the dynamic programming table a column Min Cost that holds the minimum of the Cost columns for all the physical-property combinations. Similarly, in *find_best_split* one can maintain a *max_best_cost_so_far* that holds the maximum of the *best_cost_so_far* values for all property combinations. Then when a specific split is considered, if the sum of the Min Cost values for the left- and right-hand sides exceeds *max_best_cost_so_far*, one can immediately discard the split, without individually examining *any* of the $\left(\prod_{i=0}^{M-1} b_i\right)^2$ plans conforming to that split. This example involves sharing of the cost analysis at a large granularity, in the sense that it lumps together all the plans for a split; finer-granularity sharing is also possible, and probably advisable.

In addition, the memory requirements entailed by physical properties can be reduced by avoiding the storage of redundant information. For example, suppose that for a given join, the Tokyo Best Split and the Kyoto Best Split happen to be the same; suppose, in fact, that the best way to obtain the result of this join in Kyoto is to compute it in Tokyo, and then to ship the result to Kyoto. In such a case, it is wasteful to record identical information for the Tokyo Best Split and Kyoto Best Split, when it would suffice to store this information just once. Moreover, it is wasteful to record both a Tokyo Cost and a Kyoto Cost, since in this example the latter can be derived from the former rather easily. This kind of waste may be tolerable when there are just two cities involved, as in our example; but if there were, say, *fifty* cities involved, the waste would become more worrisome.

The key to avoiding such waste lies in recognizing the common features of the plans—and the costs—for different property combinations. The locality properties of the Blitzsplit algorithm make it especially well-suited to taking advantage of shared representations of plan information. But the details of the pertinent data structures, and the corresponding cost-analysis code, remain to be worked out.

10.2 Top-down vs. Bottom-up

McKenna and Graefe [20, 40] have argued that the top-down, memoizing style of dynamic programming used in Volcano is more efficient than bottom-up dynamic programming.

(Volcano's transformation-based plan generation may also offer an advantage in flexibility (e.g., in accommodating new query operators), but here we shall address only the efficiency issue.) The core of the argument given by McKenna and Graefe regarding efficiency is that top-down optimization is goal-driven, and can avoid pursuing subgoals that are not warranted by the top-level goal.

The argument seems plausible, but needs additional support to be convincing. In his comparisons of Volcano performance against Starburst performance, McKenna addresses only the efficiency of enumeration, showing that the two systems are roughly comparable in this regard. Whether Volcano succeeds in performing fewer cost-function executions than Starburst remains unclear.

It should be noted that inasmuch as top-down dynamic programming is preferable to bottom-up, the Blitzsplit algorithm can be changed into a top-down algorithm with only small revisions. Rather than systematically filling in the table entries for *all* sets $\mathcal{S}$, the top-down variant simply searches for a best split for the set $\mathcal{R}$ of all relations. In the course of doing so, it may need to consult other table entries; these other entries are then filled in, as needed, when they are accessed. But if a table entry is never accessed, it never needs to be filled in. Thus, the table construction becomes demand-driven.

One can also imagine variants of the Blitzsplit algorithm that cannot be neatly categorized as either bottom-up or top-down. One might fill in the dynamic programming table with incomplete information in a bottom-up fashion, and then fill it in more completely on demand.

In the early stages of the research described here, the author actually experimented *first* with a top-down variant of the Blitzsplit algorithm, only later settling on the bottom-up variant as preferable. The top-down variant did perform better in some special cases—for example, for chain-like queries of 18 relations or more—but its performance quickly degraded when it faced more difficult queries.

The author's current view is that very large queries are best optimized by stochastic methods such as the Recursive Bushwhack algorithm described in Chapter 9 above. As problem difficulty increases, stochastic methods degrade more gracefully than dynamic programming algorithms, be they bottom-up or top-down: In performing exhaustive

search, dynamic programming algorithms always guarantee optimal solutions, but demand exorbitant amounts of time and memory to solve large, difficult problems. Stochastic methods, by contrast, let go of the guarantee of optimality when it comes at too high a price. In this way they achieve a better balance between efficient query execution and efficient optimization.

For queries of more moderate size, the bottom-up variant of dynamic programming unquestionably beats top-down in speed of enumeration. Moreover, the use of techniques such as plan-cost slices permits pruning of many unnecessary cost-function executions. However, it is conceivable that a top-down strategy can prune away cost-function executions even more effectively. A systematic study of the number of cost-function executions needed by either approach remains as a challenge for future research.

10.3 Extension beyond Relational Systems

The join-optimization work described in this dissertation was an outgrowth of work on algebraic foundations for object-oriented query optimization. However, the techniques and experiments reported in the foregoing chapters have all been presented in a relational setting. It is natural to wonder what is involved in generalizing these results to object-oriented databases.

The central observation to be made in this regard concerns the algebraic foundation for join-order optimization. The commutativity and associativity of the relational join operator are responsible for giving the plan-space the shape it has. (If arbitrary Cartesian products are allowed, then predicates have no role in the topology of this space.)

It is difficult to construct a genuinely object-oriented analogue of the relational join operator without giving up commutativity and associativity.[1] But it is trivial to construct an analogue that is commutative and associative *up to isomorphism*, in the manner of a categorical product [2, 48]. That is, one can construct an object join $\overset{\circ}{\bowtie}$ such that if A, B,

[1] Analogues have been proposed that retain these properties; see, for example, the work of Shaw and Zdonik [52]. However, such analogues tend to exhibit anomalous algebraic characteristics.

and C are unordered object collections, then

$$A \overset{\circ}{\bowtie} B \cong B \overset{\circ}{\bowtie} A \tag{10.1}$$

$$(A \overset{\circ}{\bowtie} B) \overset{\circ}{\bowtie} C \cong A \overset{\circ}{\bowtie} (B \overset{\circ}{\bowtie} C). \tag{10.2}$$

Actual equality between the left- and right-hand sides is unnecessary for join reordering—it turns out that isomorphism suffices. Thus, the join-order optimization techniques discussed in this work can be applied to join expressions built with an object-join operator that obeys (10.1) and (10.2) above.

10.4 Conclusion

This dissertation has taken a fresh look at the use of dynamic programming in join-order optimization; in going over old ground, it has turned up surprising new findings, and these findings hold promise of making query optimizers more efficient. But it has also turned up new puzzles, and leaves many questions unanswered. Despite the long history of dynamic programming in join optimization, the subtler aspects of its behavior—those having to do with the amount of effort required for cost computations—are still not well understood. Further investigation of these matters could yield large rewards in the efficiency of commercial query optimizers.

This dissertation has also taken a fresh look at the idea of combining dynamic programming with randomized search in a hybrid optimizer. Even less is understood about the behavior of this kind of hybrid than about dynamic programming itself. But such hybrids appear to possess enormous potential—a potential that has yet to be tapped.

Bibliography

[1] Gennady Antoshenkov. Query processing in DEC Rdb: Major issues and future challenges. *Database Engineering*, 16(4):42–52, December 1993.

[2] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures.* The MIT Press, 1991.

[3] Roberto Bayardo, IBM Almaden Research Center, San Jose, California. Personal communication, January 1996.

[4] Keith Billings. A TPC-D model for query optimization in Cascades. Master's thesis, Portland State University. In preparation.

[5] Stavros Christodoulakis. Estimating block transfers and join sizes. In *SIGMOD '83, Proceedings of Annual Meeting, Database Week, San Jose, May 23–26, 1983*, pages 40–54, 1983.

[6] Sophie Cluet and Guido Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Database Theory—ICDT '95, 5th International Conference, Prague, Czech Republic, January 11–13, 1995, Proceedings*, volume 893 of *Lecture Notes in Computer Science*, pages 54–67. Springer-Verlag, 1995.

[7] Richard Cole, Mark J. Anderson, and Robert J. Bestgen. Query processing in the IBM Application System 400. *Database Engineering*, 16(4):19–28, December 1993.

[8] George Copeland and David Maier. Making Smalltalk a database system. In *SIGMOD '84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18–21, 1984*, pages 316–325, 1984.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Los Angeles, California, January 17–19, 1977*, pages 238–252, 1977.

[10] Scott Daniels, Goetz Graefe, Thomas Keller, David Maier, Duri Schmidt, and Bennet Vance. Query optimization in Revelation, an overview. *Database Engineering*, 14(2):58–62, June 1991.

[11] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

[12] César Galindo-Legaria, Arjan Pellenkoft, and Martin Kersten. Fast, randomized join-order selection—why use transformations? In *Proceedings of the 20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile*, pages 85–95, 1994.

[13] César A. Galindo-Legaria, Microsoft Corporation, Redmond, Washington. Personal communication, June 1996.

[14] Sumit Ganguly, Phillip B. Gibbons, Yossi Matias, and Avi Silberschatz. Bifocal sampling for skew-resistant join size estimation. In *1996 Proceedings, ACM SIGMOD International Conference on Management of Data, June 4 to 6, Montréal, Québec, Canada*, pages 271–281, 1996.

[15] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2–5, 1992*, pages 9–18, 1992.

[16] Peter Gassner, Guy Lohman, and K. Bernhard Schiefer. Query optimization in the IBM DB2 family. *Database Engineering*, 16(4):4–18, December 1993.

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

[18] Goetz Graefe. The Cascades framework for query optimization. *Database Engineering*, 18(3):19–29, September 1995.

[19] Goetz Graefe and David J. DeWitt. The Exodus optimizer generator. In *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data 1987, Annual Conference, San Francisco, May 27–29, 1987*, pages 160–172, 1987.

[20] Goetz Graefe and William J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19–23, 1993, Vienna, Austria*, pages 209–218, 1993.

[21] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, 1989.

[22] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, May 26–28, 1993*, pages 267–276, 1993.

[23] Robert V. Hogg and Allen T. Craig. *Introduction to Mathematical Statistics.* Macmillan, 3rd edition, 1970.

[24] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[25] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, September 1984.

[26] Yannis E. Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29–31, 1991*, pages 168–177, 1991.

[27] Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. In *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data 1987, Annual Conference, San Francisco, May 27–29, 1987*, pages 9–22, 1987.

[28] Navin Kabra and David J. DeWitt. Opt++: An object-oriented implementation for extensible database query optimization, 1995. Available: http://www.cs.wisc.edu/~navin/research/opt++.ps [September 26, 1997].

[29] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming.* Addison-Wesley, 2nd edition, 1973.

[30] Robert Kooi and Derek Frankforth. Query optimization in Ingres. *Database Engineering*, 5(3):2–5, September 1982.

[31] Robert Philip Kooi. *The Optimization of Queries in Relational Databases.* PhD thesis, Case Western Reserve University, 1980.

[32] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill, 2nd edition, 1991.

[33] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the Twelfth International Conference on Very Large Data Bases, Kyoto, Japan, August 25–28, 1986*, pages 128–137, 1986.

[34] Rosana S. G. Lanzelotte, Patrick Valduriez, Mohamed Zaït, and Mikal Ziane. Industrial-strength parallel query optimization: Issues and lessons. *Information Systems*, 19(4):311–330, 1994.

[35] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, May 23–25, 1990, Atlantic City, New Jersey*, pages 1–11, 1990.

[36] C. L. Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill, 1968.

[37] Guy M. Lohman, IBM Almaden Research Center, San Jose, California. Personal communication, April 1996.

[38] David Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[39] O. Martin and S. Otto. Combining simulated annealing with local search heuristics. *Annals of Operations Research*, 63:57–75, 1996.

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

[41] William J. McKenna, Louis Burger, Chi Hoang, and Melissa Truong. Eroc: A toolkit for building Neato query optimizers. In *Proceedings of the Twenty-second International Conference on Very Large Data Bases, September 3–6, 1996, Mumbai (Bombay), India*, pages 111–121, 1996.

[42] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1990.

[43] Priti Mishra and Margaret H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[44] Kiyoshi Ono and Guy M. Lohman. Extensible enumeration of feasible joins for relational query optimization. *Technical Report RJ6625*, IBM Almaden Research Center, December 1988.

[45] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases, August 13–16, 1990, Brisbane, Australia*, pages 314–325, 1990.

[46] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin Kersten. The complexity of transformation-based join enumeration. In *Proceedings of the Twenty-third International Conference on Very Large Data Bases, Athens, Greece, 26–29 August, 1997*, pages 306–315, 1997.

[47] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *SIGMOD '84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18–21, 1984*, pages 256–276, 1984.

[48] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* The MIT Press, 1991.

[49] Wolfgang Scheufele and Guido Moerkotte. On the complexity of generating optimal plans with cross products. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona, May 12–14, 1997*, pages 238–248, 1997.

[50] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *ACM-SIGMOD 1979 International Conference on Management of Data, May 30–June 1, The 57 Park Plaza Hotel, Boston, Massachusetts, Proceedings*, pages 23–34, 1979.

[51] Leonard Shapiro, David Maier, Keith Billings, Yubo Fan, Bennet Vance, Quan Wang, and Hsiao-min Wu. Safe pruning in the Columbia query optimizer. Submitted for publication, 1997. For more information, see the web site http://www.cs.pdx.edu/~len [November 4, 1997].

[52] Gail M. Shaw and Stanley B. Zdonik. An object-oriented query algebra. In *Proceedings of the Second International Workshop on Database Programming Languages, 4–8 June 1989, Salishan Lodge, Gleneden Beach, Oregon*, pages 103–112. Morgan Kaufmann, 1990.

[53] David W. Shipman. The functional data model and the data language Daplex. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[54] M. Steinbrunn. *Heuristic and Randomised Optimisation Techniques in Object-Oriented Database Systems.* Infix-Verlag, Germany, 1996. Also published as a PhD thesis, Universität Passau.

[55] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Optimizing join orders. *Technical Report MIP-9307*, Universität Passau, 1993.

[56] Arun Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon*, pages 367–376, 1989.

[57] Arun Swami. Distributions of query plan costs for large join queries. *Technical Report RJ7908*, IBM Almaden Research Center, January 1991.

[58] Arun Swami and Anoop Gupta. Optimization of large join queries. In *1988 Proceedings, SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1–3*, pages 8–17, 1988.

[59] Arun N. Swami and Balakrishna R. Iyer. A polynomial time algorithm for optimizing join queries. In *Proceedings of the Ninth International Conference on Data Engineering, April 19–23, 1993, Vienna, Austria*, pages 345–354, 1993.

[60] Bennet Vance. Presentation to group meeting of the Revelation project [10], led by Prof. David Maier at the Oregon Graduate Institute of Science & Technology, January 1994.

[61] Bennet Vance and David Maier. Rapid bushy join-order optimization with Cartesian products. In *1996 Proceedings, ACM SIGMOD International Conference on Management of Data, June 4 to 6, Montréal, Québec, Canada*, pages 35–46, 1996.

[62] Carlo Zaniolo. The database language Gem. In *SIGMOD '83, Proceedings of Annual Meeting, Database Week, San Jose, May 23–26, 1983*, pages 207–218, 1983.

Appendix A

Complexity of Join Enumeration in Starburst

This appendix shows that the worst-case time complexity of join enumeration in Starburst is $O(4^n)$, where n is the number of relations to be joined. Our complexity analysis is based on join-generation pseudo-code given by Ono and Lohman [44]. The pseudo-code in question is reproduced with minor reformatting in Figure A.1. According to Ono and Lohman, Starburst uses a *different* join-enumeration strategy when it detects chain-like join graphs—but the code shown here implements the strategy that would be used to handle a worst-case optimization problem.

Below we first briefly describe the mechanism of the Starburst join-generation pseudo-code. Then we explain our approach to analyzing the complexity of this code, and proceed to go through the details of the analysis.

A.1 The Starburst Join-Generation Mechanism

Consider a worst-case bushy optimization problem involving four relations A, B, C, and D. Let us say that these relations belong to a clique graph, so that the join of any subset of $\{A, B, C, D\}$ can be expressed without Cartesian products. We will examine how the code in Figure A.1 generates the various join expressions that could represent a subcomputation in the join of $\{A, B, C, D\}$.

Prior to the execution of the code in Figure A.1, some initialization would occur. One of the initialization steps would be to insert the singleton sets $\{A\}$, $\{B\}$, $\{C\}$, and $\{D\}$ into a collection called *qset*[1].

```
for k := 2 to n do
  for i := 1 to ⌊k/2⌋ do
    for each large_set in qset[k − i] do
      for each small_set in qset[i] do
        if feasibility criteria succeed then
          put (large_set ∪ small_set) into qset[k]
          (* Implicitly, at this point the algorithm also registers
             large_set ⋈ small_set as one of the possible joins for
             large_set ∪ small_set. *)
        end if
      end for
    end for
  end for
end for
```

Figure A.1: Starburst join-generation algorithm

The variable *qset* is actually an *array* of collections. As we have seen, *qset*[1] is a collection of singleton sets; *qset*[2] will be a collection of 2-relation sets; and the same pattern holds for arbitrary k, so that in general, *qset*[k] will be a collection of sets that each contain k relations.

The outer loop in Figure A.1 lets k run from 2 to n, and on successive iterations, the body of this outer loop fills in the collection *qset*[k] for each value of k. When $k = 2$, the code constructs 2-relation sets by considering joins of pairs of 1-relation sets from *qset*[1]. That is, i takes the value 1, and so *large_set* and *small_set* both range over the sets in *qset*[1]. When *large_set* = $\{A\}$ and *small_set* = $\{B\}$, the code implicitly constructs the join expression $A \bowtie B$ as a possible means of computing the join over $\{A, B\}$; what is explicit in the code is that it inserts $\{A, B\}$ into *qset*[2] (assuming this set is not already present in the collection). When it pairs $\{A\}$ with $\{C\}$, the code obtains a join over $\{A, C\}$ and inserts this set into *qset*[2] as well; and similarly for other pairings.

But note that the code will also attempt pairings of *large_set* with *small_set* when both are bound to $\{A\}$, and when both are bound to $\{B\}$, and so on. In these cases, the *feasibility criteria* are not met, and so (for example) the set $\{A\} \cup \{A\} = \{A\}$ will *not* be inserted into *qset*[2], and $A \bowtie A$ will *not* be registered as a possible join for the set $\{A\}$. Starburst's feasibility criteria are configurable, but the minimal criterion that must always

be met is that *large_set* and *small_set* be disjoint. On this basis, a pairing of $\{A\}$ with $\{A\}$ is rejected. No other feasibility criteria apply in the situation under consideration.

When $k = 3$, *large_set* ranges over all the 2-relation sets, while *small_set* ranges over all the 1-relation sets. It is apparent that all 3-relation sets will be obtained in this manner. When $k = 4$, the variable i first takes on the value 1, causing 3-relation sets to be paired with 1-relation sets; subsequently, with $i = 2$, the 2-relation sets are paired with 2-relation sets. Once again it is evident that all 4-relation sets will be obtained.

At the end of this process, $qset[k]$ for each k from 1 to 4 will contain all the k-relation subsets of $\{A, B, C, D\}$. The *number* of sets in $qset[k]$ will be the number of k-relation subsets of a 4-element set, or $\binom{4}{k}$.

More generally, in the optimization of a worst-case join of n relations, the number of sets in $qset[k]$ will be $\binom{n}{k}$. Note that when the algorithm of Figure A.1 is in the process of constructing $qset[k]$ for some particular k, the construction of $qset[j]$ for all $j < k$ will have already been completed.

A.2 Overview of Complexity Calculation

Now let us calculate the worst-case complexity of this algorithm. We shall assume that per-iteration loop overheads are constant, and we shall also assume that the execution time of the **if**-block (inside the loops) is bounded by a constant. Thus, to obtain the complexity of the algorithm, we shall merely count the number of iterations of the **if**-block inside all the loops. We shall refer to this iteration count as I_{total}.

To count iterations of the **if**-block, we proceed as follows. First we observe that for fixed k and fixed i, the number of iterations of the loop on *large_set* is equal to the number of sets in $qset[k-i]$, which in the worst case is $\binom{n}{k-i}$, as noted above. Similarly, the number of iterations of the loop on *small_set* is equal to the number of sets in $qset[i]$, or $\binom{n}{i}$. Note that these two innermost loops are independent of one another. Consequently, for a given k and i, the number of iterations of the **if**-block is the product of the iteration counts of the two innermost loops, or $\binom{n}{k-i}\binom{n}{i}$. Below we shall find it more convenient to write this product with its factors reversed, as $\binom{n}{i}\binom{n}{k-i}$. Thus, using the notation $I(k, i)$ to denote

the iteration count of the inner **if**-block per execution of the innermost pair of loops, we have

$$I(k,i) = \binom{n}{i}\binom{n}{k-i}. \tag{A.1}$$

Next we must step back and examine, for fixed k, the iteration count of the **if**-block per execution of the loop on i. But since i simply ranges from 1 to $\lfloor k/2 \rfloor$, it is straightforward to express this count as a summation. To take into account the effect of the two innermost loops, we must sum the value $I(k,i)$ defined above across the specified range of i values:

$$I_{i\text{-}loop}(k) = \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i) \tag{A.2}$$

$$= \sum_{i=1}^{\lfloor k/2 \rfloor} \binom{n}{i}\binom{n}{k-i}. \tag{A.3}$$

It will be easier to calculate with the approximation

$$I'_{i\text{-}loop}(k) = \sum_{i=0}^{\lfloor k/2 \rfloor} \binom{n}{i}\binom{n}{k-i}, \tag{A.4}$$

in which the lower limit of the range for i has been changed from 1 to 0. We will base our calculations on $I'_{i\text{-}loop}(k)$, bearing in mind that it is only an approximation to $I_{i\text{-}loop}(k)$.

Finally, we can obtain the total iteration count of the **if**-block by summing $I_{i\text{-}loop}$ for all the values taken by k in the outermost loop. Thus,

$$I_{total} = \sum_{k=2}^{n} I_{i\text{-}loop}(k). \tag{A.5}$$

Again, it will be easier to calculate with an approximation. We will find it convenient to sum over $I'_{i\text{-}loop}(k)$ rather than $I_{i\text{-}loop}(k)$; moreover, we will also change the lower limit of the outer sum:

$$I'_{total} = \sum_{k=0}^{n} I'_{i\text{-}loop}(k). \tag{A.6}$$

Later we will consider the amount of error introduced by simplifying the limits on both sums. For the present, our objective is to calculate I'_{total}, which will give us a good approximation to the complexity of the algorithm. Our first subgoal will be to calculate $I'_{i\text{-}loop}(k)$. Subsequently we will calculate I'_{total} itself.

A.3 Calculating $I'_{i\text{-}loop}(k)$

Here we evaluate the sum for $I'_{i\text{-}loop}(k)$ given in (A.4). As given, the sum $I'_{i\text{-}loop}(k) = \sum_{i=0}^{\lfloor k/2 \rfloor} \binom{n}{i}\binom{n}{k-i}$ is awkward to evaluate, because the upper limit of the range for i is the peculiar quantity $\lfloor k/2 \rfloor$. It is much easier to evaluate the sum $\sum_{i=0}^{k} \binom{n}{i}\binom{n}{k-i}$. It is therefore sensible to ask what is the relationship between these two sums. As it happens, the latter sum comes to almost exactly twice the sum we are seeking, because of symmetry considerations to be discussed presently. Our strategy, therefore, will be to convert the sum in $I'_{i\text{-}loop}(k)$ into an expression involving the easier sum.

The applicable symmetry considerations are the following. For each i in the range 0 to $\lfloor k/2 \rfloor$, there is a corresponding i' in the range $\lceil k/2 \rceil$ to k such that $\binom{n}{i}\binom{n}{k-i} = \binom{n}{i'}\binom{n}{k-i'}$. To wit, take $i' = k - i$; then plainly $0 \le i \le \lfloor k/2 \rfloor$ implies $\lceil k/2 \rceil \le i' \le k$, and

$$\begin{aligned}\binom{n}{i'}\binom{n}{k-i'} &= \binom{n}{k-i}\binom{n}{k-(k-i)} \\ &= \binom{n}{k-i}\binom{n}{i} \\ &= \binom{n}{i}\binom{n}{k-i}.\end{aligned}$$

Since the correspondence between i and i' is one-to-one, we may sum across the i' range instead of the i range, without fear of dropping or repeating any terms of the original sum. Thus, we have

$$\sum_{i=0}^{\lfloor k/2 \rfloor} \binom{n}{i}\binom{n}{k-i} = \sum_{i=\lceil k/2 \rceil}^{k} \binom{n}{i}\binom{n}{k-i}. \tag{A.7}$$

The left- and right-hand sides of (A.7) give us two different ways of expressing $I'_{i\text{-}loop}(k)$.

In our next step towards calculating $I'_{i\text{-}loop}(k)$, we distinguish between the cases of odd and even k.

Case k odd If k is odd, then $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$ are consecutive integers. We may use this fact in conjunction with the two formulations for $I'_{i\text{-}loop}(k)$ given by (A.7), as follows:

$$I'_{i\text{-}loop}(k) = \frac{1}{2}\left[I'_{i\text{-}loop}(k) + I'_{i\text{-}loop}(k)\right] \tag{A.8}$$

$$= \frac{1}{2}\left[\sum_{i=0}^{\lfloor k/2 \rfloor} \binom{n}{i}\binom{n}{k-i} + \sum_{i=\lceil k/2 \rceil}^{k} \binom{n}{i}\binom{n}{k-i}\right] \tag{A.9}$$

$$= \frac{1}{2}\sum_{i=0}^{k} \binom{n}{i}\binom{n}{k-i}. \tag{A.10}$$

The last sum can be reduced to closed form by way of the Vandermonde convolution [21, 29], which may be expressed as

$$\sum_{i=0}^{k} \binom{r}{i}\binom{s}{k-i} = \binom{r+s}{k}. \tag{A.11}$$

If we take $r = s = n$, then the left-hand side of (A.11) becomes the same as (A.10), except for a factor of $1/2$; hence, when k is odd,

$$I'_{i\text{-}loop}(k) = \frac{1}{2}\binom{r+s}{k} \tag{A.12}$$

$$= \frac{1}{2}\binom{2n}{k}. \tag{A.13}$$

Case k even The situation is not radically different when k is even. But when k is even, then $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$, rather than being *consecutive* integers, are *the same*: $\lfloor k/2 \rfloor = k/2 = \lceil k/2 \rceil$. Consequently, the ranges $[0, \lfloor k/2 \rfloor]$ and $[\lceil k/2 \rceil, k]$ overlap at $k/2$, and cannot simply be tacked together. What we shall do instead is to break the upper range $[\lceil k/2 \rceil, k]$ into two subranges—$[k/2, k/2]$ and $[(k/2)+1, k]$. The first of these subranges, $[k/2, k/2]$, contributes a single term that can be moved outside the summation. We will then be left with summations over the lower range $[0, k/2]$ and the upper subrange $[(k/2)+1, k]$, which are adjacent and can be combined into a single sum.

In the manipulations below, note that $\sum$ is assumed to bind more tightly than $+$, so

that $\sum A + B$ means $(\sum A) + B$, not $\sum(A + B)$. Thus, we have the following derivation.

$$I'_{i\text{-}loop}(k) = \frac{1}{2}\left[I'_{i\text{-}loop}(k) + I'_{i\text{-}loop}(k)\right] \tag{A.14}$$

$$= \frac{1}{2}\left[\sum_{i=0}^{k/2}\binom{n}{i}\binom{n}{k-i} + \sum_{i=k/2}^{k}\binom{n}{i}\binom{n}{k-i}\right] \tag{A.15}$$

$$= \frac{1}{2}\left[\sum_{i=0}^{k/2}\binom{n}{i}\binom{n}{k-i} + \sum_{i=k/2}^{k/2}\binom{n}{i}\binom{n}{k-i} + \sum_{i=k/2+1}^{k}\binom{n}{i}\binom{n}{k-i}\right] \tag{A.16}$$

$$= \frac{1}{2}\left[\sum_{i=0}^{k/2}\binom{n}{i}\binom{n}{k-i} + \binom{n}{k/2}\binom{n}{k-k/2} + \sum_{i=k/2+1}^{k}\binom{n}{i}\binom{n}{k-i}\right] \tag{A.17}$$

$$= \frac{1}{2}\left[\sum_{i=0}^{k/2}\binom{n}{i}\binom{n}{k-i} + \binom{n}{k/2}^2 + \sum_{i=k/2+1}^{k}\binom{n}{i}\binom{n}{k-i}\right] \tag{A.18}$$

$$= \frac{1}{2}\left[\sum_{i=0}^{k/2}\binom{n}{i}\binom{n}{k-i} + \sum_{i=k/2+1}^{k}\binom{n}{i}\binom{n}{k-i}\right] + \frac{1}{2}\binom{n}{k/2}^2 \tag{A.19}$$

$$= \frac{1}{2}\sum_{i=0}^{k}\binom{n}{i}\binom{n}{k-i} + \frac{1}{2}\binom{n}{k/2}^2 \tag{A.20}$$

$$= \frac{1}{2}\binom{2n}{k} + \frac{1}{2}\binom{n}{k/2}^2. \tag{A.21}$$

The general case Now let us combine the cases of odd and even k. We define χ_{even} to be the *characteristic function* for the even integers:

$$\chi_{even}(k) = \begin{cases} 0 & \text{if } k \text{ is odd,} \\ 1 & \text{if } k \text{ is even.} \end{cases} \tag{A.22}$$

By multiplying a term by this characteristic function, we make inclusion of the term conditional on k being even. Thus, (A.13) and (A.21) can be combined to give

$$I'_{i\text{-}loop}(k) = \frac{1}{2}\binom{2n}{k} + \frac{1}{2}\binom{n}{k/2}^2 \cdot \chi_{even}(k), \tag{A.23}$$

which is valid whether k is odd or even.

A.4 Calculating I'_{total}

We now turn to the calculation of I'_{total} itself. From (A.6) and (A.23) we have

$$I'_{total} = \sum_{k=0}^{n} I'_{i\text{-}loop}(k) \tag{A.24}$$

$$= \sum_{k=0}^{n} \left[\frac{1}{2}\binom{2n}{k} + \frac{1}{2}\binom{n}{k/2}^2 \cdot \chi_{even}(k) \right] \tag{A.25}$$

$$= \frac{1}{2}\sum_{k=0}^{n}\binom{2n}{k} + \frac{1}{2}\sum_{k=0}^{n}\binom{n}{k/2}^2 \cdot \chi_{even}(k). \tag{A.26}$$

Consider the sum $\sum_{k=0}^{n}\binom{2n}{k}$. This sum is most easily solved by the same technique we used in calculating the sum in $I'_{i\text{-}loop}$—namely, by taking advantage of a symmetry property to extend the range of the summation. Recall that in general, $\binom{r}{r-k} = \binom{r}{k}$. Now observe that if k is in the range 0 to n, and if we define $k' = 2n - k$, then k' will be in the range n to $2n$, and

$$\binom{2n}{k'} = \binom{2n}{2n-k} = \binom{2n}{k}.$$

It follows that

$$\sum_{k=0}^{n}\binom{2n}{k} = \sum_{k=n}^{2n}\binom{2n}{k}, \tag{A.27}$$

and hence that

$$\sum_{k=0}^{n}\binom{2n}{k} = \frac{1}{2}\left[\sum_{k=0}^{n}\binom{2n}{k} + \sum_{k=n}^{2n}\binom{2n}{k}\right] \tag{A.28}$$

$$= \frac{1}{2}\left[\sum_{k=0}^{n}\binom{2n}{k} + \sum_{k=n+1}^{2n}\binom{2n}{k} + \binom{2n}{n}\right] \tag{A.29}$$

$$= \frac{1}{2}\sum_{k=0}^{2n}\binom{2n}{k} + \frac{1}{2}\binom{2n}{n}, \tag{A.30}$$

which, by a basic property of binomial coefficients [36], simplifies to

$$\frac{1}{2}\cdot 2^{2n} + \frac{1}{2}\binom{2n}{n} \tag{A.31}$$

$$= \frac{1}{2}\cdot 4^{n} + \frac{1}{2}\binom{2n}{n}. \tag{A.32}$$

Substituting (A.32) into the first term of (A.26), we obtain

$$I'_{total} = \frac{1}{2}\left[\frac{1}{2}\cdot 4^n + \frac{1}{2}\binom{2n}{n}\right] + \frac{1}{2}\sum_{k=0}^{n}\binom{n}{k/2}^2 \cdot \chi_{even}(k) \tag{A.33}$$

$$= \frac{1}{4}\cdot 4^n + \frac{1}{4}\binom{2n}{n} + \frac{1}{2}\sum_{k=0}^{n}\binom{n}{k/2}^2 \cdot \chi_{even}(k) \tag{A.34}$$

$$= 4^{n-1} + \frac{1}{4}\binom{2n}{n} + \frac{1}{2}\sum_{k=0}^{n}\binom{n}{k/2}^2 \cdot \chi_{even}(k). \tag{A.35}$$

Now we turn to the sum $\sum_{k=0}^{n}\binom{n}{k/2}^2 \cdot \chi_{even}(k)$. Yet again a symmetry property will prove useful. As before, if k is in the range 0 to n, and $k' = 2n - k$, then k' is in the range n to $2n$, and k' is even if and only if k is even, so $\chi_{even}(k') = \chi_{even}(k)$. Moreover, when k' and k are even, then

$$\binom{n}{k'/2} = \binom{n}{(2n-k)/2} = \binom{n}{n-k/2} = \binom{n}{k/2}.$$

Thus,

$$\sum_{k=0}^{n}\binom{n}{k/2}^2 \cdot \chi_{even}(k) \tag{A.36}$$

$$= \frac{1}{2}\left[\sum_{k=0}^{n}\binom{n}{k/2}^2 \cdot \chi_{even}(k) + \sum_{k=n}^{2n}\binom{n}{k/2}^2 \cdot \chi_{even}(k)\right] \tag{A.37}$$

$$= \frac{1}{2}\left[\sum_{k=0}^{n}\binom{n}{k/2}^2 \cdot \chi_{even}(k) + \sum_{k=n+1}^{2n}\binom{n}{k/2}^2 \cdot \chi_{even}(k) + \binom{n}{n/2}^2 \cdot \chi_{even}(n)\right] \tag{A.38}$$

$$= \frac{1}{2}\sum_{k=0}^{2n}\binom{n}{k/2}^2 \cdot \chi_{even}(k) + \frac{1}{2}\binom{n}{n/2}^2 \cdot \chi_{even}(n). \tag{A.39}$$

Since the terms of the sum in (A.39) are nonzero only for even k, in effect the sum ranges over $k = 0, 2, 4, \ldots, 2n$; equivalently, if we let $j = k/2$, then the sum ranges over $j = 0, 1, 2, \ldots, n$. Changing the index variable from k to j, and replacing all occurrences of k with $2j$, we obtain

$$\frac{1}{2}\sum_{j=0}^{n}\binom{n}{2j/2}^2 \cdot \chi_{even}(2j) + \frac{1}{2}\binom{n}{n/2}^2 \cdot \chi_{even}(n) \tag{A.40}$$

$$= \frac{1}{2}\sum_{j=0}^{n}\binom{n}{j}^2 + \frac{1}{2}\binom{n}{n/2}^2 \cdot \chi_{even}(n). \tag{A.41}$$

Now recall the Vandermonde convolution, equation (A.11) above. In the special case where r, s, and k are all equal to n, (A.11) becomes

$$\sum_{i=0}^{n} \binom{n}{i}\binom{n}{n-i} = \binom{n+n}{n}. \tag{A.42}$$

Changing the index variable from i to j, then using the fact that $\binom{n}{n-j} = \binom{n}{j}$, and simplifying, we obtain

$$\sum_{j=0}^{n} \binom{n}{j}^2 = \binom{2n}{n}. \tag{A.43}$$

Then (A.41) may be simplified using (A.43) to give

$$\frac{1}{2}\binom{2n}{n} + \frac{1}{2}\binom{n}{n/2}^2 \cdot \chi_{even}(n). \tag{A.44}$$

In (A.44), we finally have a closed-form expression for $\sum_{k=0}^{n} \binom{n}{k/2}^2 \cdot \chi_{even}(k)$. Substituting (A.44) for the sum in (A.35), and then simplifying, we obtain

$$I'_{total} = 4^{n-1} + \frac{1}{4}\binom{2n}{n} + \frac{1}{2}\left[\frac{1}{2}\binom{2n}{n} + \frac{1}{2}\binom{n}{n/2}^2 \cdot \chi_{even}(n)\right] \tag{A.45}$$

$$= 4^{n-1} + \frac{1}{4}\binom{2n}{n} + \frac{1}{4}\binom{2n}{n} + \frac{1}{4}\binom{n}{n/2}^2 \cdot \chi_{even}(n) \tag{A.46}$$

$$= 4^{n-1} + \frac{1}{2}\binom{2n}{n} + \frac{1}{4}\binom{n}{n/2}^2 \cdot \chi_{even}(n). \tag{A.47}$$

We have in (A.47) a good approximation to the worst-case complexity of join enumeration in Starburst. We now show that the 4^{n-1} term dominates (though not by much), and hence that the complexity is $O(4^n)$. Using Stirling's approximation for the factorial [29],

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n, \tag{A.48}$$

we can rewrite $\binom{2n}{n}$ as follows:

$$\binom{2n}{n} = \frac{(2n)!}{n!(2n-n)!} \tag{A.49}$$

$$= \frac{(2n)!}{(n!)^2} \tag{A.50}$$

$$\approx \frac{\sqrt{2\pi(2n)}\left(\frac{2n}{e}\right)^{2n}}{\left[\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\right]^2} \tag{A.51}$$

$$= \frac{\sqrt{2}\sqrt{2\pi n}}{(\sqrt{2\pi n})^2} \cdot \frac{(2^{2n} \cdot n^{2n})/e^{2n}}{n^{2n}/e^{2n}} \tag{A.52}$$

$$= \frac{\sqrt{2}}{\sqrt{2\pi n}} \cdot 2^{2n} \tag{A.53}$$

$$= \frac{1}{\sqrt{\pi n}} \cdot 4^n. \tag{A.54}$$

Analogously, when n is even,

$$\binom{n}{n/2} \approx \frac{1}{\sqrt{\pi n/2}} \cdot 4^{n/2}, \tag{A.55}$$

and so

$$\binom{n}{n/2}^2 \approx \left[\frac{1}{\sqrt{\pi n/2}} \cdot 4^{n/2}\right]^2 \tag{A.56}$$

$$= \frac{1}{\pi n/2} \cdot 4^n \tag{A.57}$$

$$= \frac{2}{\pi n} \cdot 4^n. \tag{A.58}$$

Incorporating the approximations given by (A.54) and (A.58) into (A.47), we obtain

$$I'_{total} = 4^{n-1} + \frac{1}{2}\binom{2n}{n} + \frac{1}{4}\binom{n}{n/2}^2 \cdot \chi_{even}(n) \tag{A.59}$$

$$\approx 4^{n-1} + \frac{1}{2}\left(\frac{1}{\sqrt{\pi n}} \cdot 4^n\right) + \frac{1}{4}\left(\frac{2}{\pi n} \cdot 4^n\right) \cdot \chi_{even}(n) \tag{A.60}$$

$$= 4^{n-1} + \frac{1}{2\sqrt{\pi n}} \cdot 4^n + \frac{1}{2\pi n} \cdot 4^n \cdot \chi_{even}(n). \tag{A.61}$$

The leading term dominates, and as n grows into the teens, the contributions of the remaining two terms become relatively small.

A.5 Correction for Extraneous Terms

To simplify our calculations, in Section A.2 above we altered the lower limits of the summations involved in the foregoing complexity analysis. The effect of these alterations is to *overestimate* the complexity of Starburst join generation. We now examine the amount by which we overestimated, and show that it is much less significant than even the lower-order terms in (A.61).

The correct sum would have been

$$I_{total} = \sum_{k=2}^{n} I_{i\text{-}loop}(k) = \sum_{k=2}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i), \tag{A.62}$$

whereas the sum we actually used was

$$I'_{total} = \sum_{k=0}^{n} I'_{i\text{-}loop}(k) = \sum_{k=0}^{n} \sum_{i=0}^{\lfloor k/2 \rfloor} I(k,i). \tag{A.63}$$

The difference $I'_{total} - I_{total}$ is then

$$\sum_{k=0}^{n} \sum_{i=0}^{\lfloor k/2 \rfloor} I(k,i) - \sum_{k=2}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i) \tag{A.64}$$

$$= \sum_{k=0}^{n} \sum_{i=0}^{\lfloor k/2 \rfloor} I(k,i) - \Big[0\Big] - \sum_{k=2}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i) \tag{A.65}$$

$$= \sum_{k=0}^{n} \sum_{i=0}^{\lfloor k/2 \rfloor} I(k,i) - \Big[\sum_{k=0}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i) - \sum_{k=0}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i)\Big] - \sum_{k=2}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i) \tag{A.66}$$

$$= \sum_{k=0}^{n} \sum_{i=0}^{\lfloor k/2 \rfloor} I(k,i) - \sum_{k=0}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i) + \sum_{k=0}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i) - \sum_{k=2}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i) \tag{A.67}$$

$$= \sum_{k=0}^{n} \Big[\sum_{i=0}^{\lfloor k/2 \rfloor} I(k,i) - \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i)\Big] + \Big[\sum_{k=0}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i) - \sum_{k=2}^{n} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i)\Big] \tag{A.68}$$

$$= \sum_{k=0}^{n} \sum_{i=0}^{0} I(k,i) + \sum_{k=0}^{1} \sum_{i=1}^{\lfloor k/2 \rfloor} I(k,i) \tag{A.69}$$

$$= \sum_{k=0}^{n} I(k,0) + \Big[\sum_{i=1}^{\lfloor 0/2 \rfloor} I(0,i) + \sum_{i=1}^{\lfloor 1/2 \rfloor} I(1,i) \Big] \tag{A.70}$$

$$= \sum_{k=0}^{n} \binom{n}{0} \binom{n}{k} + \big[0 + 0 \big] \tag{A.71}$$

$$= \sum_{k=0}^{n} \binom{n}{k} \tag{A.72}$$

$$= 2^n. \tag{A.73}$$

When $n \geq 6$, this 2^n error-term does not exceed a few percent of the value given by (A.61). Thus we conclude that the worst-case complexity of join enumeration in Starburst is indeed $O(4^n)$.

Appendix B

Implementation of Blitzsplit Algorithm in C

```
#include <stdio.h>

#define MAX_N           18
#define COST_LIMIT      1E35


typedef int relname;
typedef int setrep;

typedef struct {
        float   cardinality;
} rel_data_entry;


double  t_cardinality[1 << MAX_N];    /* ---------------------- */
setrep  t_best_lhs[1 << MAX_N];       /* "table" (in 3 pieces) */
float   t_cost[1 << MAX_N];           /* ---------------------- */


void init_singleton(relname, rel_data_entry *);
void compute_properties(setrep);
void find_best_split(setrep);
void find_best_split1(setrep, setrep);
void find_best_split2(setrep, setrep, setrep, setrep);
```

```
#define least_subset(set)       ((set) & -(set))
#define next_subset(subset,set) (((subset) - (set)) & (set))


int blitzsplit(int n, rel_data_entry *rel_data)
{
        relname R;
        setrep  s;
        int     tabsize = 1 << n;

        if (n < 1 || n > MAX_N) {
                printf("blitzsplit:  bad input\n");
                exit(1);
        }

        for (R = 0; R < n; R++)
                init_singleton(R, rel_data);

        for (s = 1; s < tabsize; s++)
                if (least_subset(s) != s) {
                        compute_properties(s);
                        find_best_split(s);
                }

        return (t_cost[tabsize - 1] < COST_LIMIT) ? 0 : -1;
                        /* 0 = OK, -1 = cost limit exceeded */
}


void init_singleton(relname R, rel_data_entry *rel_data)
{
        setrep s = 1 << R;

        t_cardinality[s] = rel_data[R].cardinality;
        t_best_lhs[s] = R; /* kludge for output */
        t_cost[s] = 0.0;
}
```

```
void compute_properties(setrep s)
{
        setrep  u = least_subset(s),    v = s - u;

        t_cardinality[s] = t_cardinality[u] * t_cardinality[v];
}

void find_best_split(setrep s)
{
        setrep  u = least_subset(s),    v = s - u,
                w = least_subset(v),    z = v - w;

        if (t_cardinality[s] >= COST_LIMIT) {
                t_best_lhs[s] = v;      /* dummy plan */
                t_cost[s] = COST_LIMIT;
        }
        else if (z == 0)
                find_best_split1(s, v);
        else
                find_best_split2(s, v, w, least_subset(z) - w);
}



/* Macros for find_best_split1 and find_best_split2 */

#define intcost(i)      (((int *) t_cost)[i])
#define intbest         (*((int *) (&best_cost_so_far)))
#define loop_body \
        rhs = s - lhs; \
        if (intcost(rhs) < intbest && intcost(lhs) < intbest) { \
                operand_cost = t_cost[lhs] + t_cost[rhs]; \
                if (operand_cost < best_cost_so_far) { \
                        best_cost_so_far = operand_cost; \
                        best_lhs = lhs; \
                } \
        }
```

```
void find_best_split1(setrep s, setrep v)
{
        setrep  lhs, rhs, best_lhs = v;
        float   operand_cost, best_cost_so_far = COST_LIMIT;

        lhs = 0;
        while ((lhs = next_subset(lhs,v)) != 0) {
                loop_body
        }
        t_best_lhs[s] = best_lhs;
        t_cost[s] = best_cost_so_far + t_cardinality[s];
}

void find_best_split2(setrep s, setrep v, setrep d1, setrep d2)
{
        setrep  lhs, rhs, best_lhs = v;
        float   operand_cost, best_cost_so_far = COST_LIMIT;

        lhs = t_best_lhs[v];              /* Heuristic to     */
        loop_body       lhs += s - v;     /* reduce initial   */
        loop_body                         /* best_cost_so_far */

        lhs = d1;
        loop_body       lhs += d2;
        loop_body       lhs += d1;
        loop_body

        while ((lhs = next_subset(lhs,v)) != 0) {
                loop_body       lhs += d1;
                loop_body       lhs += d2;
                loop_body       lhs += d1;
                loop_body
        }
        t_best_lhs[s] = best_lhs;
        t_cost[s] = best_cost_so_far + t_cardinality[s];
}
```

Appendix C

Parameterization of Test Queries

This appendix gives details on the parameterization of the test queries used in our performance measurements in Chapters 6, 7, and 9. We first discuss the four dimensions of parameterization of our basic tests. We then give additional details about our cost-function computations, and illustrate two principles that can be applied to the implementation of a cost model to reduce costing effort.

C.1 The Four Dimensions of Parameterization

In all measurements in Chapters 6 and 7, the number of base relations n was held fixed at 15; but in the measurements in Chapter 9, n varies. Because $n = 15$ is an important special case, in the following descriptions of our parameterizations, we use the example of $n = 15$ (where applicable) for illustration. Regardless of the value of n, we shall assume that the relations R_0, R_1, and so on, are numbered in order of increasing cardinality. Thus, R_0 is smallest, and R_{n-1} is largest.

Our four basic dimensions of parameterization are the *mean cardinality* of the base relations, the *variability* of the base relations, the *join graph*, and the *cost model.*

C.1.1 Mean Cardinality

We define the mean base-relation cardinality to be the geometric mean

$$(\prod_{i=0}^{n-1} |R_i|)^{1/n}.$$

For example, if $n = 15$ and relations R_0 through R_{14} have the respective cardinalities

$$10^0, 10^1, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, 10^8, 10^9, 10^{10}, 10^{11}, 10^{12}, 10^{13}, 10^{14}, \tag{C.1}$$

then the mean base-relation cardinality would be

$$(10^0 \cdot 10^1 \cdot 10^2 \cdot 10^3 \cdot 10^4 \cdot 10^5 \cdot 10^6 \cdot 10^7 \cdot 10^8 \cdot 10^9 \cdot 10^{10} \cdot 10^{11} \cdot 10^{12} \cdot 10^{13} \cdot 10^{14})^{1/15} \tag{C.2}$$

$$= (10^{105})^{1/15} \tag{C.3}$$

$$= 10^7. \tag{C.4}$$

C.1.2 Variability

The *variability* ranges from 0 to 1, with 0 indicating *no* variability in the base-relation cardinalities (i.e., all $|R_i|$ are equal), and with 1 indicating maximal variability subject to the constraints that the smallest relation cardinality be at least 1, and that the cardinalities of relations R_0 through R_{n-1} be equally spaced on a logarithmic scale. In general,

$$|R_0| = (\textit{mean cardinality})^{1-\textit{variability}},$$

and the remaining R_i are such that $|R_i|/|R_{i-1}|$ is constant.

For example, the cardinality sequence illustrated in (C.1) above is obtained when the mean cardinality is 10^7 and the variability is 1. Observe that in that case,

$$|R_0| = (\textit{mean cardinality})^{1-\textit{variability}} = (10^7)^{1-1} = (10^7)^0 = 1,$$

and $|R_i|/|R_{i-1}| = 10$ for $i = 1, \ldots, 14$.

C.1.3 Join Graph

Our measurements use four kinds of join graphs: a chain graph, an augmented cycle graph that we refer to as a "*cycle* + 3" graph, a star graph, and a clique. We describe first the topologies of these graphs, and then we describe the assignment of selectivity values to the edges of the graphs.

The Graph Topologies

When $n = 15$, our chain graphs have the following predicate connections:

$$R_0\text{–}R_8\text{–}R_1\text{–}R_9\text{–}R_2\text{–}R_{10}\text{–}R_3\text{–}R_{11}\text{–}R_4\text{–}R_{12}\text{–}R_5\text{–}R_{13}\text{–}R_6\text{–}R_{14}\text{–}R_7.$$

More generally, the pattern of connections is as follows:

$$R_0\text{—}R_{\lceil n/2\rceil}\text{—}R_1\text{—}R_{\lceil n/2\rceil+1}\text{—}R_2\text{—}R_{\lceil n/2\rceil+2}\text{— } \ldots .$$

The last relation in the chain is R_{n-1} if n is even, and $R_{\lfloor n/2\rfloor}$ if n is odd.

Recall that the relations R_0, R_1, and so on, are numbered in order of increasing cardinality. Thus the cardinalities along the chain occur in a kind of sawtooth pattern; but overall the cardinalities are higher at the right-hand end of the chain than at the left-hand end. This choice of chain configurations was more or less arbitrary. (Experiments with other configurations have given similar performance results.)

The "*cycle* + 3" topology is similar to the chain, but adds several predicates that connect the two ends of the chain. Specifically, when $n = 15$, the "*cycle* + 3" topology takes the form illustrated in Figure C.1. The graph in the figure may be thought of as a chain that has been wrapped around on top of itself so that the last relation in the chain (R_7) lies directly above the first relation in the chain (R_0). The chain has then been augmented with the following predicate connections: R_0–R_7, R_8–R_{14}, R_1–R_6, and R_9–R_{13}.

In the general case as well, a "*cycle* + 3" graph consists of a chain that has been augmented with four additional predicates, just as in Figure C.1. In each such graph, the

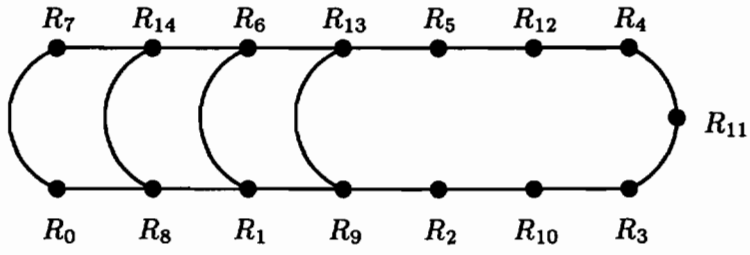


Figure C.1: The "*cycle* + 3" join-graph topology for $n = 15$

first additional predicate connects the first and last relations in the chain, thus changing the chain into a cycle. Then three more predicates are added: one between the second and second-to-last relations in the chain; one between the third and third-to-last relations in the chain; and one between the fourth and fourth-to-last relations in the chain. (Note that the "*cycle* + 3" topology makes sense only for $n \geq 9$.)

In our star graphs, the hub of the star is always the relation of largest cardinality. Thus, the star graphs have predicate connections between the hub R_{n-1} and each other relation.

Cliques have predicate connections between every pair of relations.

Assignment of Selectivities

In all our graphs, the selectivity of the predicate (if any) connecting R_i and R_j is computed as

$$\mu^{1/k} \cdot |R_i|^{-1/k_i} \cdot |R_j|^{-1/k_j}, \tag{C.5}$$

where μ is the mean base-relation cardinality (*cf.* Section C.1.1 above), k is the total number of predicates, and k_i is the number of predicates incident on R_i.

These selectivities always yield a query-result cardinality of μ, for the following reason. The query-result cardinality is the product of all relation cardinalities and all predicate selectivities. Consider a particular relation R_l for some l. This relation contributes a factor of $|R_l|$ to the result cardinality; but the k_l predicates incident on R_l each contribute a factor of $|R_l|^{-1/k_l}$, so together these predicates contribute a factor of $|R_l|^{-1}$, and thus cancel out the relation's contribution of $|R_l|$. Meanwhile, each predicate contributes a factor of $\mu^{1/k}$, and since there are k predicates altogether, these factors combine to yield $(\mu^{1/k})^k = \mu$.

Let us now consider an example that illustrates the assignment of predicate selectivities, and the effect these selectivities have on the query-result cardinality. Because selectivity assignments are somewhat complicated, we give an example with $n = 3$ rather than $n = 15$. Suppose that R_0, R_1, and R_2 have cardinalities 100^0, 100^1, and 100^2, respectively. Thus, $\mu = 100$. Suppose also that the join graph has edge connections R_0–R_2

and R_1–R_2. (This graph may be viewed either as the chain R_0–R_2–R_1 or as the star with hub R_2.)

Then the total number of predicates k is 2. The number of predicates k_0 incident on R_0 is 1, and similarly for the number of predicates k_1 incident on R_1. But the number of predicates k_2 incident on R_2 is 2.

Hence, by (C.5) above, the predicate R_0–R_2 has selectivity

$$\mu^{1/k} \cdot |R_0|^{-1/k_0} \cdot |R_2|^{-1/k_2} \tag{C.6}$$

$$= 100^{1/2} \cdot (100^0)^{-1/1} \cdot (100^2)^{-1/2} \tag{C.7}$$

$$= 10 \cdot 1^{-1} \cdot 100^{-1} \tag{C.8}$$

$$= 10^{-1}, \tag{C.9}$$

and the predicate R_1–R_2 has selectivity

$$\mu^{1/k} \cdot |R_1|^{-1/k_1} \cdot |R_2|^{-1/k_2} \tag{C.10}$$

$$= 100^{1/2} \cdot (100^1)^{-1/1} \cdot (100^2)^{-1/2} \tag{C.11}$$

$$= 10 \cdot 100^{-1} \cdot 100^{-1} \tag{C.12}$$

$$= 10^{-3}. \tag{C.13}$$

It follows that the join of R_0 and R_2 has cardinality $|R_0| \cdot |R_2| \cdot 10^{-1} = 100^0 \cdot 100^2 \cdot 10^{-1} = 10^3$, and that the join of R_1 and R_2 has cardinality $|R_1| \cdot |R_2| \cdot 10^{-3} = 100^1 \cdot 100^2 \cdot 10^{-3} = 10^3$. Thus we see that our selectivity assignments tend to have an equalizing effect on intermediate-result cardinalities.

The query-result cardinality is the product of all cardinalities and all selectivities, which in this case works out to

$$|R_0| \cdot |R_1| \cdot |R_2| \cdot 10^{-1} \cdot 10^{-3} \tag{C.14}$$

$$= 100^0 \cdot 100^1 \cdot 100^2 \cdot 10^{-1} \cdot 10^{-3} \tag{C.15}$$

$$= 100. \tag{C.16}$$

In other words, the query-result cardinality is equal to the mean base-relation cardinality. The same holds in all our test queries.

C.1.4 Cost Model

Our cost models are drawn from the study by Steinbrunn et al. [55]. (More recent versions of that study [54] do not use the same cost models.) We use three cost models: a *naive* cost model, a *sort-merge* cost model, and a *disk-nested-loops* cost model.

The cost function κ_0 of the naive cost model (*cf.* Section 2.6.2) is defined as

$$\kappa_0(R_{out}, R_{lhs}, R_{rhs}) = |R_{out}|. \tag{C.17}$$

The cost function κ_{sm} for the sort-merge cost model is defined as

$$\kappa_{sm}(R_{out}, R_{lhs}, R_{rhs}) = |R_{lhs}| \cdot (1 + \log |R_{lhs}|) + |R_{rhs}| \cdot (1 + \log |R_{rhs}|). \tag{C.18}$$

The logarithms in this cost function reflect the fact that sorting a relation consisting of N tuples has complexity $N \log N$. Note that the presence of the logarithm in the cost-function definition makes this cost function relatively time-consuming to compute. In Section C.2 below we discuss a technique for avoiding performance degradation from the logarithm computation.

Our disk-nested-loops model is based on that of Steinbrunn et al., but is formulated differently here:

$$\kappa_{dnl}(R_{out}, R_{lhs}, R_{rhs}) = 2 \cdot |R_{out}|/K + |R_{lhs}| \cdot |R_{rhs}|/K^2(M-1) + \min(|R_{lhs}|, |R_{rhs}|)/K, \tag{C.19}$$

where K is the blocking factor of relation records per disk block (we make the simplifying assumption that K is a constant), and M is the number of disk blocks that can be held in main memory. In our measurements, we arbitrarily set $K = 10$ and $M = 100$. (In separate tests, we have found that actually computing the relation widths and blocking factors, rather than taking K to be constant, has little effect on the performance graphs.)

C.2 Details of Cost-Function Computation

As noted in Chapter 3, efficient computation of the cost function κ is critical to overall performance of the Blitzsplit algorithm. It was for this reason that in Figure 3.2 we provided

that a cost function κ could be decomposed into a split-independent component κ^{out} and a split-dependent component κ^{split}. The algorithmic improvement illustrated in Figure 3.4 sought to reduce the execution frequency of κ^{split}; but even with this improvement, it is desirable to decompose a cost function κ into components κ^{out} and κ^{split} in such a way as to minimize the computation entailed by κ^{split}.

In this section we discuss cost-function decomposition in greater detail, and illustrate two principles of cost-function transformation that can be applied to reduce costing effort under some cost models.

C.2.1 Decomposition of Cost Functions

Each of the three cost models we use in our benchmarks makes a different assumption about the sources of join cost.

- In the naive model, with the cost function κ_0 defined on page 53 and again in (C.17) above, the output cardinality is the sole determinant of join cost. Thus, as noted in Chapter 3, this cost function can be decomposed as

$$\kappa_0^{out}(R_{out}) = |R_{out}| \tag{C.20}$$

$$\kappa_0^{split}(R_{out}, R_{lhs}, R_{rhs}) = 0. \tag{C.21}$$

  Because κ_0^{split} entails no computation at all, costs become extremely cheap to compute under this model.

- In the sort-merge model (κ_{sm}), defined by (C.18) above, cost is determined solely by the cardinalities of the join *inputs*. The cost actually has two summands—one that depends only on the left input, and one that depends only on the right input. Since no part of κ_{sm} depends solely on the join result R_{out}, direct implementation of this model must necessarily place the entire cost computation, including two logarithm calculations, in the split-dependent cost-function component κ^{split}:

$$\kappa_{sm}^{out}(R_{out}) = 0 \tag{C.22}$$

$$\kappa_{sm}^{split}(R_{out}, R_{lhs}, R_{rhs}) = |R_{lhs}| \cdot (1 + \log |R_{lhs}|) + |R_{rhs}| \cdot (1 + \log |R_{rhs}|). \tag{C.23}$$

- The cost function of the disk-nested-loops model (κ_{dnl}), defined by (C.19) above, has three summands, one of which is proportional to the *product* of the input cardinalities. This summand in particular is split-dependent and must necessarily be computed in κ^{split}. (The final summand of κ_{dnl} must also be computed in κ^{split}.)

$$\kappa^{out}_{dnl}(R_{out}) = 2 \cdot |R_{out}|/K \tag{C.24}$$

$$\kappa^{split}_{dnl}(R_{out}, R_{lhs}, R_{rhs}) = |R_{lhs}| \cdot |R_{rhs}|/K^2(M-1) + \min(|R_{lhs}|, |R_{rhs}|)/K. \tag{C.25}$$

On the surface, the sort-merge model is the most computation-intensive of the three. However, it turns out that it is actually the product in the nested-loops model that causes the most difficulty—it is generally hard to avoid computing that product. But a minor transformation, described next, makes the sort-merge model quite tractable.

C.2.2 Transformation of a Class of Cost Functions

We now consider a transformation that reduces the effort required to optimize under a cost model such as the sort-merge model κ_{sm}. We present the transformation in generality, and then apply it specifically to our sort-merge cost model.

Let κ be a cost function with a decomposition of the form

$$\kappa(R_{out}, R_{lhs}, R_{rhs}) = \kappa^{out}(R_{out}) + \kappa^{split}(R_{out}, R_{lhs}, R_{rhs}), \tag{C.26}$$

and consider the case where κ^{split} in turn has a nontrivial decomposition as

$$\kappa^{split}(R_{out}, R_{lhs}, R_{rhs}) = \kappa^{input}(R_{lhs}) + \kappa^{input}(R_{rhs}) + \kappa^{residual}(R_{out}, R_{lhs}, R_{rhs}). \tag{C.27}$$

Under these conditions we may define a new cost function

$$\kappa_*(R_{out}, R_{lhs}, R_{rhs}) = \kappa^{out}(R_{out}) + \kappa^{input}(R_{out}) + \kappa^{residual}(R_{out}, R_{lhs}, R_{rhs}), \tag{C.28}$$

which is *not equivalent* to κ, but which may nonetheless be substituted for κ without changing the outcome of optimization.

We shall defend this claim in a moment; but first let us consider why such a substitution is desirable. Recall the execution counts that motivated our decomposition of a cost function κ into a split-independent component κ^{out} and a split-dependent component κ^{split}. We expect to require about 2^n evaluations of κ^{out}, and somewhere between $(\ln 2/2)n2^n$ and 3^n evaluations of κ^{split} (*cf.* Chapter 3). Hence our aim in the decomposition is to reduce the net costing effort by performing as much of the cost computation as possible in κ^{out}, and as little as possible in κ^{split}.

Introducing the new function κ_* assists us in this goal. Observe that κ_*, as defined in (C.28) above, can be rewritten as

$$\kappa_*(R_{out}, R_{lhs}, R_{rhs}) = \kappa_*^{out}(R_{out}) + \kappa_*^{split}(R_{out}, R_{lhs}, R_{rhs}), \tag{C.29}$$

where

$$\kappa_*^{out}(R_{out}) = \kappa^{out}(R_{out}) + \kappa^{input}(R_{out}) \tag{C.30}$$

and

$$\kappa_*^{split}(R_{out}, R_{lhs}, R_{rhs}) = \kappa^{residual}(R_{out}, R_{lhs}, R_{rhs}). \tag{C.31}$$

In this decomposition we have a split-independent component κ_*^{out} that is *more* complicated than κ^{out}, while the split-dependent component κ_*^{split} is *less* complicated than κ^{split}. In other words, by substituting κ_* for κ, we shift effort away from the critical split-dependent component of the cost computation.

C.2.3 Justification for the Transformation

There remains the question of why this substitution is legitimate. Intuitively, the transformation of κ into κ_* involves a change in the way costs are assigned to the nodes of a join-processing tree. Consider a node that computes $E_0 \bowtie E_1$; the child nodes of this node compute E_0 and E_1. Under the cost function κ, the cost components $\kappa^{input}([\![E_0]\!])$ and $\kappa^{input}([\![E_1]\!])$ are charged to the node for $E_0 \bowtie E_1$. Under the cost function κ_*, these components are instead charged to the child nodes—$\kappa^{input}([\![E_0]\!])$ is charged to the node for E_0, and $\kappa^{input}([\![E_1]\!])$ is charged to the node for E_1. Seen this way, the difference between

κ and κ_* boils down to technical details of cost accounting: The same costs are being charged, but under different headings.

However, this intuitive view is an oversimplification. Let us examine the relationship between κ and κ_* with greater precision. Suppose $cost(E)$ is defined as on page 53; i.e.,

$$cost(R) = 0 \tag{C.32}$$

$$cost(E_0 \bowtie E_1) = cost(E_0) + cost(E_1) + \kappa([\![E_0 \bowtie E_1]\!], [\![E_0]\!], [\![E_1]\!]). \tag{C.33}$$

Next suppose that $cost_*(E)$ is defined analogously, so that

$$cost_*(R) = 0 \tag{C.34}$$

$$cost_*(E_0 \bowtie E_1) = cost_*(E_0) + cost_*(E_1) + \kappa_*([\![E_0 \bowtie E_1]\!], [\![E_0]\!], [\![E_1]\!]). \tag{C.35}$$

Our intuitive argument might lead us to conjecture that $cost(E)$ and $cost_*(E)$ are equal, but in fact this equality does not hold—the intuitive view disregarded the boundary cases where a node has no parent or no children. Instead, the relationship between $cost(E)$ and $cost_*(E)$ is as follows:

$$cost_*(E) - cost(E) = \kappa^{input}([\![E]\!]) - \sum_{R \text{ in } E} \kappa^{input}([\![R]\!]), \tag{C.36}$$

where the sum subscript "R in E" indicates that the sum ranges over all relation names R that appear as leaves of the expression E. A straightforward proof by structural induction demonstrates the truth of (C.36). The intuitive interpretation of (C.36) is that $cost_*(E)$ charges the topmost node of E as if it were the input to some other node; on the other hand, $cost_*(E)$ *fails* to charge the leaf nodes R in E for the costs they entail when they serve as inputs to other nodes.

Although $cost_*(E)$ and $cost(E)$ are not algebraically equivalent, they are equivalent for the purposes of optimization because *the difference between them is independent of the structure of* E. Suppose, for example, that E and E' are two expressions that both compute the join of some set of relations $\mathcal{S}$. In other words, $[\![E]\!]$ and $[\![E']\!]$ both give the join of $\mathcal{S}$, and hence are equal; moreover, R in E and R in E' both range over exactly the

relations in $\mathcal{S}$. It follows that

$$\kappa^{input}([\![E]\!]) - \sum_{R \text{ in } E} \kappa^{input}([\![R]\!]) = \kappa^{input}([\![E']\!]) - \sum_{R \text{ in } E'} \kappa^{input}([\![R]\!]), \tag{C.37}$$

and hence, by (C.36), that

$$cost_*(E) - cost(E) = cost_*(E') - cost(E'). \tag{C.38}$$

Equivalently, we have

$$cost_*(E) - cost_*(E') = cost(E) - cost(E'). \tag{C.39}$$

Now suppose E is an optimal expression for joining $\mathcal{S}$ under cost function κ. Then if E' is any other expression for joining $\mathcal{S}$, it must be the case that $cost(E) - cost(E') \leq 0$. From (C.39) it follows that $cost_*(E) - cost_*(E') \leq 0$; i.e., $cost_*(E) \leq cost_*(E')$. We see, then, that E is also optimal according to cost function κ_*.

C.2.4 Application to Sort-Merge Cost Model

The cost function κ_{sm} for our sort-merge cost model was defined in (C.18) as follows:

$$\kappa_{sm}(R_{out}, R_{lhs}, R_{rhs}) = |R_{lhs}| \cdot (1 + \log |R_{lhs}|) + |R_{rhs}| \cdot (1 + \log |R_{rhs}|). \tag{C.40}$$

It is apparent that (C.40) has the form

$$\kappa_{sm}(R_{out}, R_{lhs}, R_{rhs}) = \kappa_{sm}^{out}(R_{out}) + \kappa_{sm}^{split}(R_{out}, R_{lhs}, R_{rhs}) \tag{C.41}$$

with

$$\kappa_{sm}^{split}(R_{out}, R_{lhs}, R_{rhs}) = \kappa_{sm}^{input}(R_{lhs}) + \kappa_{sm}^{input}(R_{rhs}) + \kappa_{sm}^{residual}(R_{out}, R_{lhs}, R_{rhs}), \tag{C.42}$$

provided that we define

$$\kappa_{sm}^{out}(R) = 0 \tag{C.43}$$

$$\kappa_{sm}^{input}(R) = |R| \cdot (1 + \log |R|) \tag{C.44}$$

$$\kappa_{sm}^{residual}(R_{out}, R_{lhs}, R_{rhs}) = 0. \tag{C.45}$$

Applying the transformation of Section C.2.2 above, we obtain the alternative cost function

$$\kappa_{sm*}(R_{out}, R_{lhs}, R_{rhs}) = |R_{out}| \cdot (1 + \log |R_{out}|), \tag{C.46}$$

which decomposes into the following split-independent and split-dependent components:

$$\kappa^{out}_{sm*}(R_{out}) = |R_{out}| \cdot (1 + \log |R_{out}|) \tag{C.47}$$

$$\kappa^{split}_{sm*}(R_{out}, R_{lhs}, R_{rhs}) = 0. \tag{C.48}$$

It is this transformed version of the sort-merge cost model that we use in our benchmarks.

C.2.5 Generalization of the Transformation

Our transformation of the sort-merge cost function may be regarded as "cheating," in the sense that even a slight change in the definition of the cost function would have rendered the transformation inapplicable. But the two principal benefits of the transformation are generally applicable even if the transformation itself is not:

- The transformation yields a nonzero split-independent cost component, which promotes pruning of cost computations, as discussed in Chapter 7. In any realistic cost model, there should be some split-independent cost—albeit a small one—associated with the generation of join-output tuples. If a cost model does not include a component that is proportional to output cardinality, it needs to be reformulated.

- After the transformation, logarithms need to be computed just 2^n times, rather than the $(\ln 2)n2^n$ times one would expect from the original cost-function formulation—or, in the worst case, $2 \cdot 3^n$ times. But it is generally the case that if the cost function has a subexpression that depends on only one of the join inputs, then that subexpression need be evaluated just 2^n times. That is, the result of such a subexpression can be memoized in the dynamic programming table, and hence computed just once for each distinct set of relations.

Our transformation combines these two benefits; and because it does so without reliance on memoization, it gives a particularly convenient means of obtaining the desired effects.

But at the same time, it illustrates two principles of cost-function computation that can and should be applied in the formulation of any cost model.

Biographical Note

Bennet Vance was born in Hanover, New Hampshire in 1954. He graduated *summa cum laude* from Yale University in 1976 with distinction in the major in mathematics, and received a master's degree in computer science from Stanford University in 1981. He is the author or co-author of several papers on query optimization, including a paper on join-order optimization that he presented at the 1996 SIGMOD conference. Currently employed at the IBM Almaden Research Center, he has spent thirteen years in the software industry as a programmer, designer, consultant, and researcher. His interests include query processing, functional programming, and software engineering. He is a member of Phi Beta Kappa and the ACM.