CHAPTER 18

Optimization

18.1 INTRODUCTION

Optimization represents both a challenge and an opportunity for relational systems: a challenge, because optimization is *required* if the system is to achieve acceptable performance: an opportunity, because it is precisely one of the strengths of such systems that relational expressions are at a sufficiently high semantic level that optimization is feasible in the first place. In a nonrelational system, by contrast, where user requests are expressed at a lower semantic level, any "optimization" has to be done manually by the human user ("optimization" in quotes, because the term is usually taken to mean *automatic* optimization); in other words, it is the *user* in such a system who decides what low-level operations are needed and what sequence they need to be executed in. And if the user makes a bad decision, there is nothing the system can do to improve matters. Note too the implication that the user must have some programming expertise; this fact by itself puts the system out of reach for many who could otherwise benefit from it.

The advantage of automatic optimization is not just that users do not have to worry about how best to state their queries (i.e., how to phrase their requests in order to get the best performance). The fact is, there is a real possibility that the optimizer might actually do better than a human user. There are several reasons for this state of affairs, the following among them:

1. A good optimizer will have a wealth of information available to it that human users typically do not have. To be specific, it will know certain statistical information (cardinality information and the like), such as:
   - The number of distinct values of each type
   - The number of tuples currently appearing in each base relvar
   - The number of distinct values currently appearing in each attribute in each base relvar
   - The number of times each such value occurs in each such attribute

   and so on. (All of this information will typically be kept in the system catalog—see Section 18.5.) As a result, the optimizer should be able to make a more accurate assessment of the efficiency of any given strategy for implementing a particular request, and thus be more likely to choose the most efficient implementation.
2. Furthermore, if the database statistics change over time, then a different choice of strategy might become desirable; in other words, reoptimization might be required. In a relational system, reoptimization is trivial—it simply involves a reprocessing of the original relational request by the system optimizer. In a nonrelational system, by contrast, reoptimization involves rewriting the program, and very likely will not be done at all.
3. Next, the optimizer is a *program*, and therefore by definition much more patient than a typical human user. The optimizer is quite capable of considering literally hundreds of different implementation strategies for a given request, whereas it is extremely unlikely that a human user would ever consider more than three or four (at least in any depth).
4. Last, the optimizer can be regarded in a sense as embodying the skills and services of "the best" human programmers. As a consequence, it has the effect of making those skills and services available to everybody—which means it is making an otherwise scarce set of resources available to a wide range of users, in an efficient and cost-effective manner.

All of the above should serve as evidence in support of the claim made at the beginning of this section to the effect that optimizability—that is, the fact that relational requests are optimizable—is in fact a strength of relational systems.

The overall purpose of the optimizer, then, is to choose an efficient strategy for evaluating a given relational expression. In this chapter we briefly describe some of the fundamental principles and techniques involved in that process. Following an introductory motivating example in Section 18.2, Section 18.3 gives an overview of how optimizers work, and Section 18.4 then elaborates on one very important aspect of the process, *expression transformation* (also known as *query rewrite*). Section 18.5 discusses the question of *database statistics*. Next, Section 18.6 describes one specific approach to optimization, called *query decomposition*, in some detail. Section 18.7 then addresses the question of how the relational operators (join and so on) are actually implemented, and briefly considers the use of the statistics discussed in Section 18.5 to perform cost estimation. Finally, Section 18.8 presents a summary of the entire chapter.

One final introductory remark: It is usual to refer to this topic as *query* optimization specifically. This term is slightly misleading, however, inasmuch as the expression to be optimized—the "query"—might have arisen in some context other than interactive interrogation of the database (in particular, it might be part of an update operation instead of a query *per se*). What is more, the term *optimization* itself is somewhat of an overclaim, since there is usually no guarantee that the implementation strategy chosen is truly optimal in any measurable sense; it might in fact be so, but usually all that is known for sure is that the "optimized" strategy is an *improvement* on the original unoptimized version. (In certain cases, however, it might be possible to claim legitimately that the chosen strategy is indeed optimal in a very specific sense; see, for example, reference [18.30]. See also Appendix A.)

18.2 A MOTIVATING EXAMPLE

We begin with a simple example—an elaboration of one already discussed briefly in Chapter 7, Section 7.6—that gives some idea of the dramatic improvements that are possible. The query is "Get names of suppliers who supply part P2." An algebraic formulation of this query is:

```
( ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

Suppose the database contains 100 suppliers and 10,000 shipments, of which only 50 are for part P2. Assume for simplicity that relvars S and SP are represented directly on the disk as two separate stored files, with one stored record per tuple. Then, if the system were simply to evaluate the expression as stated—that is, without any optimization at all—the sequence of events would be as follows:

1. *Join SP and S (over S#):* This step involves reading the 10,000 shipments; reading each of the 100 suppliers 10,000 times (once for each of the 10,000 shipments); constructing an intermediate result consisting of 10,000 joined tuples; and writing those 10,000 joined tuples back out to the disk. (For the sake of the example, we assume there is no room for this intermediate result in main memory.)
2. *Restrict the result of Step 1 to just the tuples for part P2:* This step involves reading the 10,000 joined tuples back into memory again, but produces a result consisting of only 50 tuples, which we assume is small enough to be kept in main memory.
3. *Project the result of Step 2 over SNAME:* This step produces the desired final result (50 tuples at most, which can stay in main memory).

The following procedure is equivalent to the one just described, in the sense that it necessarily produces the same final result, but is clearly much more efficient:

1. *Restrict SP to just the tuples for part P2:* This step involves reading 10,000 tuples but produces a result consisting of only 50 tuples, which we assume will be kept in main memory.
2. *Join the result of Step 1 to S (over S#):* This step involves reading the 100 suppliers (once only, not once per P2 shipment) and produces a result of 50 tuples again (still in main memory).
3. *Project the result of Step 2 over SNAME* (same as Step 3 before): The desired final result (50 tuples at most) stays in main memory.

The first of these two procedures involves a total of 1,030,000 tuple I/O's, whereas the second involves only 10,100. It is clear, therefore, that if we take "number of tuple I/O's" as our performance measure, then the second procedure is a little over 100 times better than the first. It is also clear that we would like the implementation to use the second procedure rather than the first! *Note:* In practice, it is *page* I/O's that matter, not tuple I/O's, so let us assume for simplicity that each stored tuple occupies its own page.

So we see that a very simple change in the execution algorithm—doing a restriction and then a join, instead of a join and then a restriction—has produced a dramatic (hundredfold) improvement in performance. And the improvement would be more dramatic still if shipments were *indexed* or *hashed* on P#—the number of shipment tuples read in Step 1 would be reduced from 10,000 to just 50, and the new procedure would then be nearly 7,000 times better than the original. Likewise, if suppliers were also indexed or hashed on S#, the number of supplier tuples read in Step 2 would be reduced from 100 to 50, so that the procedure would now be over 10,000 times better than the original. What this means is, if the original unoptimized query took three hours to run, the final version will run in a fraction over *one second.* And of course numerous further improvements are possible.

The foregoing example, simple though it is, should be sufficient to show the need for optimization and the kinds of improvement that are possible. In the next section, we will present an overview of a systematic approach to the optimization task; in particular, we will show how the overall problem can be divided into a series of more or less independent subproblems. That overview provides a convenient framework within which individual optimization strategies and techniques such as those discussed in subsequent sections can be explained and understood.

18.3 AN OVERVIEW OF QUERY PROCESSING

We can identify four broad stages in query processing, as follows (refer to Fig. 18.1):

1. Cast the query into internal form.
2. Convert to canonical form.
3. Choose candidate low-level procedures.
4. Generate query plans and choose the cheapest.

We now proceed to amplify each stage.

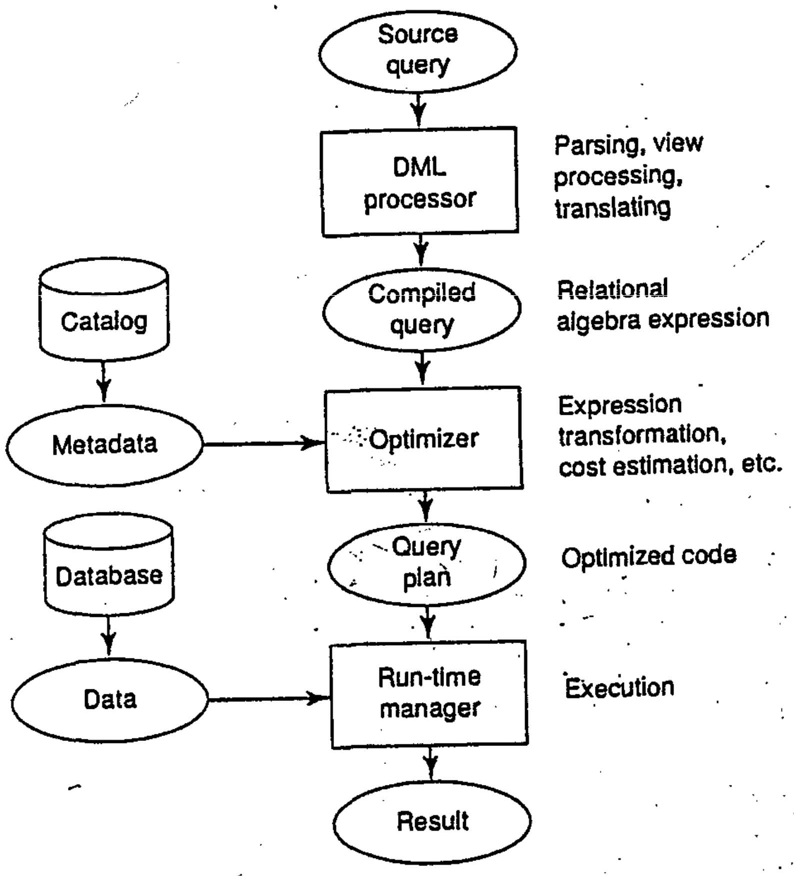

Fig. 18.1 Query processing overview

Stage 1: Cast the Query into Internal Form

The first stage involves the conversion of the original query into some internal representation that is more suitable for machine manipulation, thus eliminating purely external considerations (such as quirks of the concrete syntax of the query language under consideration) and paving the way for subsequent stages in the overall process. *Note:* View processing—that is, the process of replacing references to views by the applicable view-defining expressions—is also performed during this stage.

An obvious question is: What formalism should the internal representation be based on? Whatever formalism is chosen, it must be rich enough to represent all possible queries in the external query language. It should also be as neutral as possible, in the sense that it should not prejudice subsequent choices. The internal form typically chosen is some kind of abstract syntax tree or query tree. For example, Fig. 18.2 shows a possible query tree representation for the example from Section 18.2 ("Get names of suppliers who supply part P2").

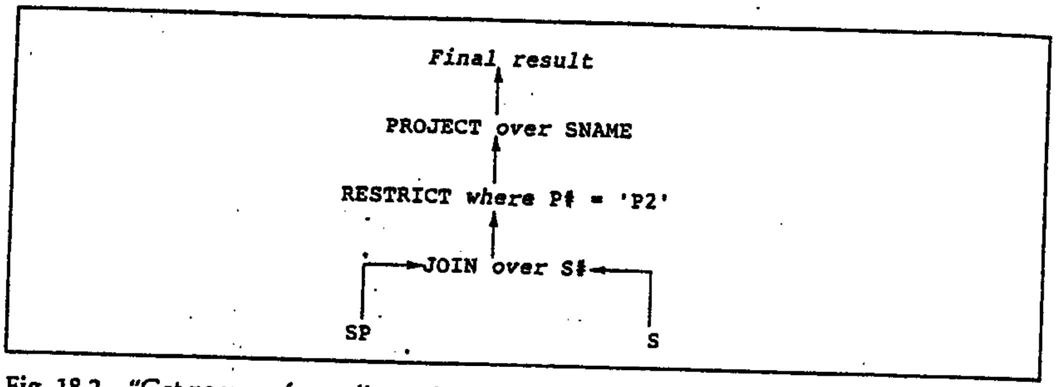

Fig. 18.2 "Get names of suppliers who supply part P2" (query tree)

For our purposes, however, it is more convenient to assume that the internal representation is one of the formalisms we are already familiar with: namely, the relational algebra or the relational calculus. A query tree such as that of Fig. 18.2 can be regarded as just an alternative, encoded representation of some expression in one of those two formalisms. To fix our ideas, we assume here that the formalism is the algebra specifically. Thus, we will assume henceforth that the internal representation of the query of Fig. 18.2 is precisely the algebraic expression shown earlier:[1]

```
( ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

Stage 2: Convert to Canonical Form

In this stage, the optimizer performs a number of optimizations that are "guaranteed to be good," regardless of the actual data values and physical access paths that exist in the stored database. The point is, relational languages typically allow all but the simplest of queries to be expressed in a variety of ways that are at least superficially distinct. In SQL, for example, even a query as simple as "Get names of suppliers who supply part P2" can be expressed in literally dozens of different ways[2]—not counting trivial variations like replacing $A = B$ by $B = A$ or p AND q by q AND p. And the performance of a query really ought not to depend on the particular way the user chose to write it. The next step in processing the query is therefore to convert the internal representation into some equivalent *canonical form* (see the next paragraph), with the objective of eliminating such superficial distinctions and—more important—finding a representation that is more efficient than the original in some way.

[1] In fact, mapping the original query into a relational algebra equivalent is exactly what some commercial SQL optimizers do.

[2] We observe, however, that the SQL language is exceptionally prone to this problem (see Exercise 8.12 in Chapter 8, also reference [4.19]). Other languages (e.g., the algebra or the calculus) typically do not provide quite so many different ways of doing the same thing. This unnecessary "flexibility" on the part of SQL actually makes life harder for the *implementer*—not to mention the user—because it makes the optimizer's job much more difficult.

A note regarding "canonical form": The notion of canonical form is central to many branches of mathematics and related disciplines. It can be defined as follows. Given a set *Q* of objects (say queries) and a notion of equivalence among those objects (say the notion that queries *q1* and *q2* are equivalent if and only if they are guaranteed to produce the same result), subset *C* of *Q* is said to be a set of canonical forms for *Q* under the stated definition of equivalence if and only if every object *q* in *Q* is equivalent to just one object *c* in *C*. The object *c* is said to be the *canonical form* for the object *q*. All "interesting" properties that apply to the object *q* also apply to its canonical form *c;* thus, it is sufficient to study just the small set *C*, not the large set *Q*, in order to prove a variety of "interesting" results.

To revert to the main thread of our discussion: In order to transform the output from Stage 1 into some equivalent but more efficient form, the optimizer makes use of certain **transformation rules** or laws. Here is an example of such a law: The expression

```
( A JOIN B ) WHERE restriction on A
```

can be transformed into the equivalent but more efficient expression

```
( A WHERE restriction on A ) JOIN B
```

We have already discussed this transformation briefly in Chapter 7, Section 7.6; in fact, it was the one we were using in our introductory example in Section 18.2, and that example showed clearly why such a transformation is desirable. Many more such laws are discussed in Section 18.4.

Stage 3: Choose Candidate Low-Level Procedures

Having converted the internal representation of the query into some more desirable form, the optimizer must then decide how to execute the transformed query represented by that converted form. At this stage considerations such as the existence of indexes or other physical access paths, distribution of data values, physical clustering of stored data, and so on, come into play (note that we paid no heed to such matters in Stages 1 and 2).

The basic strategy is to consider the query expression as specifying a series of "low-level" operations,[3] with certain interdependencies among them. An example of such an interdependency is the following: The code to perform a projection will typically require its input tuples to be sorted into some sequence, to allow it to perform duplicate elimination, which means that the immediately preceding operation in the series must produce its output tuples in that same sequence.

Now, for each possible low-level operation (and probably for various common combinations of such operations also), the optimizer will have available to it a set of predefined **implementation procedures**. For example, there will be a set of procedures for implementing the restriction operation: one for the case where the restriction is an equality comparison, one where the restriction attribute is indexed, one where it is hashed, and

[3] Level is clearly a relative concept! The operators referred to as "low-level" in this context are basically the operators of the relational algebra (join, restrict, summarize, etc.), which are more usually regarded as *high*-level.

so on. Examples of such procedures are given in Section 18.7 (see also references [18.7, 18.12]).

Each procedure will also have a (parameterized) cost formula associated with it, indicating the cost—typically in terms of disk I/O's, though some systems take CPU[4] utilization and other factors into account also—of executing that procedure. These cost formulas are used in Stage 4 (see the next subsection). References [18.7–18.12] discuss and analyze the cost formulas for a number of different implementation procedures under a variety of different assumptions. See also Section 18.7.

Next, therefore, using information from the catalog regarding the current state of the database (existence of indexes, current cardinalities, etc.), together with the pertinent interdependency information, the optimizer will choose one or more candidate procedures for implementing each of the low-level operations in the query expression. This process is sometimes referred to as access path selection (see reference [18.33]). *Note:* Actually, reference [18.33]) uses the term *access path selection* to cover both Stage 3 and Stage 4, not just Stage 3. Indeed, it might be difficult in practice to make a clean separation between the two—Stage 3 does flow more or less seamlessly into Stage 4.

Stage 4: Generate Query Plans and Choose the Cheapest

The final stage in the optimization process involves the construction of a set of candidate query plans, followed by a choice of the best (i.e., cheapest) of those plans. Each query plan is built by combining a set of candidate implementation procedures, one such procedure for each of the low-level operations in the query. Note that there will normally be many possible plans—probably embarrassingly many—for any given query. In practice, in fact, it might not be a good idea to generate all possible plans, since there will be combinatorially many of them, and the task of choosing the cheapest might well become prohibitively expensive in itself; some heuristic technique for keeping the generated set within reasonable bounds is highly desirable, if not essential (but see reference [18.53]). "Keeping the set within bounds" is usually referred to as *reducing the search space*, because it can be regarded as reducing the range ("space") of possibilities to be examined ("searched") by the optimizer to manageable proportions.

Choosing the cheapest plan obviously requires a method for assigning a cost to any given plan. Basically, the cost for a given plan is just the sum of the costs of the individual procedures that make up that plan, so what the optimizer has to do is evaluate the cost formulas for those individual procedures. The problem is, those cost formulas will depend on the size of the relation(s) to be processed; since all but the simplest queries involve the generation of intermediate results during execution (at least conceptually), the optimizer might therefore have to estimate the size of those intermediate results in order to evaluate the formulas. Unfortunately, those sizes tend to be highly dependent on actual data values. As a consequence, accurate cost estimation can be a difficult problem. References [18.2, 18.3] discuss some approaches to that problem and give references to other research in the area.

[4] CPU stands for central processing unit.

18.4 EXPRESSION TRANSFORMATION

In this section we describe some transformation laws or rules that might be useful in Stage 2 of the optimization process. Producing examples to illustrate the rules and deciding exactly why they might be useful are both left (in part) as exercises.

Of course, you should understand that, given a particular expression to transform, the application of one rule might generate an expression that can then be transformed in accordance with some other rule. For example, it is unlikely that the original query will have been directly expressed in such a way as to require two successive projections—see the second rule in the subsection "Restrictions and Projections" immediately following—but such an expression might arise internally as the result of applying certain other transformations. (An important case in point is provided by *view processing;* consider, for example, the query "Get all cities in view V," where view V is defined as the projection of suppliers on S# and CITY.) In other words, starting from the original expression, the optimizer will apply its transformation rules repeatedly until it finally arrives at an expression that it judges—according to some built-in set of heuristics—to be "optimal" for the query under consideration.

Restrictions and Projections

Here first are some transformation rules involving restrictions and projections only:

1. A sequence of restrictions on the same relation can be transformed into a single ("ANDed") restriction on that relation. For example, the expression

   ```
   ( A WHERE p1 ) WHERE p2
   ```

   is equivalent to the expression

   ```
   A WHERE p1 AND p2
   ```

   This transformation is desirable because the original formulation implies two passes over *A*, while the transformed version requires just one.

2. In a sequence of projections against the same relation, all but the last can be ignored. For example, the expression

   ```
   ( A { acl1 } ) { acl2 }
   ```

   (where *acl1* and *acl2* are commalists of attribute names) is equivalent to the expression

   ```
   A { acl2 }
   ```

   Of course, *acl2* must be a subset of *acl1* for the original expression to make sense in the first place.

3. A restriction of a projection can be transformed into a projection of a restriction. For example, the expression

   ```
   ( A { acl } ) WHERE p
   ```

   is equivalent to the expression

   ```
   ( A WHERE p ) { acl }
   ```

It is generally a good idea to do restrictions before projections, because the effect of the restriction will be to reduce the size of the input to the projection, and hence to reduce the amount of data that might need to be sorted for duplicate elimination purposes.

Distributivity

The transformation rule used in the example in Section 18.2 (transforming a join followed by a restriction into a restriction followed by a join) is actually a special case of a more general law called the *distributive* law. In general, the monadic operator *f* is said to distribute over the dyadic operator O if and only if

```
f ( A O B ) ≡ f ( A ) O f ( B )
```

for all *A* and *B*. In ordinary arithmetic, for example, SQRT (square root, assumed nonnegative) distributes over multiplication, because

```
SQRT ( A * B ) ≡ SQRT ( A ) * SQRT ( B )
```

for all *A* and *B*. Thus, an arithmetic expression optimizer can always replace either of these expressions by the other when doing arithmetic expression transformation. As a counterexample, SQRT does not distribute over addition, because the square root of *A* + *B* is not equal to the sum of the square roots of *A* and *B*, in general.

In relational algebra, restriction distributes over union, intersection, and difference. It also distributes over join, if and only if the restriction condition consists, at its most complex, of two simple restriction conditions[5] ANDed together, one for each of the two join operands. In the case of the example in Section 18.2, this requirement was indeed satisfied—in fact, the condition was a simple restriction condition on just one of the operands—and so we could use the distributive law to replace the overall expression by a more efficient equivalent. The net effect was that we were able to "do the restriction early." Doing restrictions early is usually a good idea, because it serves to reduce the number of tuples to be scanned in the next operation in sequence, and probably reduces the number of tuples in the output from that next operation too.

Here are a couple more specific cases of the distributive law, this time involving projection. First, projection distributes over union and intersection but not difference:

```
( A UNION B ) { acl } ≡ A { acl } UNION B { acl }
( A INTERSECT B ) { acl } ≡ A { acl } INTERSECT B { acl }
```

A and *B* here must be of the same type, of course.

Second, projection also distributes over join, as long as the projection retains all of the join attributes, thus:

```
( A JOIN B ) { acl } ≡ ( A { acl1 } ) JOIN ( B { acl2 } )
```

Here *acl1* is the union of the join attributes and those attributes of *acl* that appear in *A* only, and *acl2* is the union of the join attributes and those attributes of *acl* that appear in *B* only.

[5] See Chapter 7, Section 7.4, subsection "Restrict," for an explanation of the term *simple restriction condition*.

These laws can be used to "do projections early," which again is usually a good idea for reasons similar to those given previously for restrictions.

Commutativity and Associativity

Two more important general laws are the laws of *commutativity* and *associativity*. First, the dyadic operator ○ is said to be commutative if and only if

```
A ○ B  ≡  B ○ A
```

for all *A* and *B*. In arithmetic, for example, multiplication and addition are commutative, but division and subtraction are not. In relational algebra, union, intersection, and join are all commutative, but difference and division are not. So, for example, if a query involves a join of two relations *A* and *B*, the commutative law means it makes no logical difference which of *A* and *B* is taken as the "outer" relation and which the "inner." The system is therefore free to choose (say) the smaller relation as the "outer" one in computing the join (see Section 18.7).

Turning to associativity: The dyadic operator ○ is said to be associative if and only if

```
A ○ ( B ○ C )  ≡  ( A ○ B ) ○ C
```

for all *A*, *B*, *C*. In arithmetic, multiplication and addition are associative, but division and subtraction are not. In relational algebra, union, intersection, and join are all associative, but difference and division are not. So, for example, if a query involves a join of three relations *A*, *B*, and *C*, the associative and commutative laws together mean it makes no logical difference in which order the relations are joined. The system is thus free to decide which of the various possible sequences is most efficient.

Idempotence and Absorption

Another important general law is the law of *idempotence*. The dyadic operator ○ is said to be idempotent if and only if

```
A ○ A  ≡  A
```

for all *A*. As might be expected, the idempotence property can also be useful in expression transformation. In relational algebra, union, intersection, and join are all idempotent, but difference and division are not.

Union and intersection also satisfy the following useful absorption laws:

```
A UNION ( A INTERSECT B )  ≡  A

A INTERSECT ( A UNION B )  ≡  A
```

Computational Expressions

It is not just relational expressions that are subject to transformation laws. For instance, we have already indicated that certain transformations are valid for arithmetic expressions. Here is a specific example: The expression

```
A * B + A * C
```

can be transformed into

```
A * ( B + C )
```

by virtue of the fact that "*" distributes over "+". A relational optimizer needs to know about such transformations because it will encounter such expressions in the context of the extend and summarize operators.

Note, incidentally, that this example illustrates a slightly more general form of distributivity. Earlier, we defined distributivity in terms of a *monadic* operator distributing over a *dyadic* operator; in the case at hand, however, "*" and "+" are both *dyadic* operators. In general, the dyadic operator δ is said to distribute over the dyadic operator $\bigcirc$ if and only if

$$A \,\delta\, (B \bigcirc C) \equiv (A \,\delta\, B) \bigcirc (A \,\delta\, C)$$

for all *A*, *B*, *C* (in the arithmetic example, take δ as "*" and $\bigcirc$ as "+").

Boolean Expressions

We turn now to *boolean* expressions. Suppose *A* and *B* are attributes of two distinct relations. Then the boolean expression

```
A > B AND B > 3
```

is clearly equivalent to (and can therefore be transformed into) the following:

```
A > B AND B > 3 AND A > 3
```

The equivalence is based on the fact that the comparison operator ">" is transitive. Note that this transformation is certainly worth making, because it enables the system to perform an additional restriction (on *A*) before doing the greater-than join implied by the comparison "*A* > *B*". To repeat a point made earlier, doing restrictions early is generally a good idea; having the system infer additional "early" restrictions, as here, is also a good idea. *Note:* This technique is implemented in several commercial products, including, for example, DB2 (where it is called "predicate transitive closure") and Ingres.

Here is another example: The expression

```
A > B OR ( C = D AND E < F )
```

can be transformed into

```
( A > B OR C = D ) AND ( A > B OR E < F )
```

by virtue of the fact that OR distributes over AND. This example illustrates another general law—*viz.*, any boolean expression can be transformed into an equivalent expression in what is called **conjunctive normal form (CNF)**. A CNF expression is an expression of the form

```
C1 AND C2 AND ... AND Cn
```

where each of *C1*, *C2*, ..., *Cn* is, in turn, a boolean expression (called a conjunct) that involves no ANDs. The advantage of CNF is that a CNF expression is true only if every conjunct is true; equivalently, it is false if any conjunct is false. Since AND is commutative (*A* AND *B* is the same as *B* AND *A*), the optimizer can evaluate the individual conjuncts in any order it likes; in particular, it can do them in order of increasing difficulty (easiest first). As soon as it finds one that is false, the whole process can stop. Furthermore, in a parallel-processing system, it might even be possible to evaluate all of the conjuncts in parallel [18.56–18.58]. Again, as soon as one is found that is false, the whole process can stop.

It follows from this subsection and its predecessor that the optimizer needs to know how general properties such as distributivity apply not only to relational operators such as join, but also to comparison operators such as ">", boolean operators such as AND and OR, arithmetic operators such as "+", and so on.

Semantic Transformations

Consider the following expression:

```
( SP JOIN S ) { P# }
```

The join here is a *foreign-to-matching-candidate-key join;* it matches a foreign key in SP with a corresponding candidate key in S. It follows that every SP tuple does join to some S tuple, and every SP tuple therefore does contribute a P# value to the overall result. In other words, there is no need to do the join!—the expression can be simplified to just:

```
SP { P# }
```

Note carefully, however, that this transformation is valid *only* because of the semantics of the situation. In general, each of the operands in a join will include some tuples that have no counterpart in the other (and hence some tuples that do not contribute to the overall result), and transformations such as the one just illustrated will not be valid. In the case at hand, however, every tuple of SP does have a counterpart in S, because of the integrity constraint (actually a referential constraint) that says every shipment must have a supplier, and so the transformation is valid after all.

A transformation that is valid only because a certain integrity constraint is in effect is called a **semantic transformation** [18.25], and the resulting optimization is called a **semantic optimization**. Semantic optimization can be defined as the process of transforming a specified query into another, qualitatively different, query that is nevertheless guaranteed to produce the same result as the original one, thanks to the fact that the data is guaranteed to satisfy a certain integrity constraint.

It is important to understand that, in principle, *any integrity constraint whatsoever* can be used in semantic optimization (the technique is not limited to referential constraints as in the example). For instance, suppose the suppliers-and-parts database is subject to the constraint "All red parts are stored in London," and consider the query:

Get suppliers who supply only red parts and are located in the same city as at least one of the parts they supply.

This is a fairly complex query! By virtue of the integrity constraint, however, it can be transformed into the much simpler form:

Get London suppliers who supply only red parts.

Note: As far as this writer is aware, few commercial products currently do much by way of semantic optimization. In principle, however, such optimization could provide significant performance improvements—much greater improvements, very likely, than are obtained by any of today's more traditional optimization techniques. For further discussion of the semantic optimization idea, see references [18.13], [18.26–18.28], and (especially) [18.25].

Concluding Remarks

In closing this section, we emphasize the fundamental importance of the relational *closure* property to everything we have been discussing. Closure means we can write nested expressions, which means in turn that a single query can be represented by a single expression instead of a multi-statement procedure; thus, no flow analysis is necessary. Also, those nested expressions are recursively defined in terms of subexpressions, which permits the optimizer to adopt a variety of divide-and-conquer evaluation tactics (see Section 18.6). What is more, the various general laws—distributivity and so on—would not even begin to make sense in the absence of closure.

18.5 DATABASE STATISTICS

Stages 3 and 4 of the overall optimization process, the "access path selection" stages, make use of the *database statistics* stored in the catalog (see Section 18.7 for more details on how those statistics are used). For purposes of illustration, we summarize in this section, with little further comment, some of the major statistics maintained by two commercial products, DB2 and Ingres. Here first are some of the principal statistics kept by DB2:[6]

- For each base table:
  - Cardinality
  - Number of pages occupied by this table
  - Fraction of "table space" occupied by this table
- For each column of each base table:
  - Number of distinct values in this column
  - Second highest value in this column
  - Second lowest value in this column
  - For indexed columns only, the ten most frequently occurring values in this column and the number of times they occur

[6] Since they are SQL systems, DB2 and Ingres use the terms *table* and *column* in place of *relvar* and *attribute;* in this section, therefore, so do we. Also, note that both products effectively assume that base tables map directly to stored tables.

- For each index:
  - An indication of whether this is a "clustering index" (i.e., an index that is used to cluster logically related data physically on the disk)
  - If it is, the fraction of the indexed table that is still in clustering sequence
  - Number of leaf pages in this index
  - Number of levels in this index

Note: The foregoing statistics are not updated "in real time" (i.e., every time the database is updated), because of the overhead such an approach would entail. Instead, they are updated, selectively, by means of a special system utility called RUNSTATS, which is executed on demand by the DBA (e.g., after a database reorganization). An analogous remark applies to most other commercial products (though not all), including in particular Ingres (see the next paragraph), where the utility is called OPTIMIZEDB.

Here then are some of the principal Ingres statistics. *Note:* In Ingres, an index is regarded as just a special kind of stored table; thus, the statistics shown here for base tables and columns can be kept for indexes too.

- For each base table:
  - Cardinality
  - Number of primary pages for this table
  - Number of overflow pages for this table
- For each column of each base table:
  - Number of distinct values in this column
  - Maximum, minimum, and average value for this column
  - Actual values in this column and the number of times they occur

18.6 A DIVIDE-AND-CONQUER STRATEGY

As mentioned at the end of Section 18.4, relational expressions are recursively defined in terms of subexpressions, and this fact allows the optimizer to adopt a variety of divide-and-conquer strategies. Note that such strategies are likely to be especially attractive in a parallel-processing environment—in particular, in a distributed system—where different portions of the query can be executed in parallel on different processors [18.56–18.58]. In this section we examine one such strategy, called query decomposition, which was pioneered by the Ingres prototype [18.34, 18.35].

The basic idea behind query decomposition is to break down a query involving many range variables[7] into a sequence of smaller queries involving (typically) one or two such variables each, using ***detachment*** and ***tuple substitution*** to achieve the desired decomposition:

[7] Recall that the Ingres query language QUEL is calculus-based.

- **Detachment** is the process of removing a component of the query that has just one variable in common with the rest of the query.
- **Tuple substitution** is the process of substituting for one of the variables in the query a tuple at a time.

Detachment is always applied in preference to tuple substitution as long as there is a choice. Eventually, however, the query will have been decomposed via detachment into a set of smaller queries that cannot be decomposed any further using that technique, and tuple substitution must then be brought into play.

We give a single example (based on one from reference [18.34]). The query is "Get names of London suppliers who supply some red part weighing less than 25 pounds in a quantity greater than 200." Here is a QUEL formulation of this query ("Query Q0"):

```
Q0: RETRIEVE ( S.SNAME ) WHERE S.CITY   = "London"
                         AND   S.S#     = SP.S#
                         AND   SP.QTY   > 200
                         AND   SP.P#    = P.P#
                         AND   P.COLOR  = "Red"
                         AND   P.WEIGHT < 25.0
```

The (implicit) range variables here are S, P, and SP, each ranging over the base relvar with the same name.

Now, if we examine this query, we can see immediately from the last two comparisons that the only parts we are interested in are ones that are red and weigh less than 25 pounds. So we can detach the "one-variable query" (actually a projection of a restriction) involving the variable P:

```
D1: RETRIEVE INTO P' ( P.P# ) WHERE P.COLOR  = "Red"
                              AND   P.WEIGHT < 25.0
```

This one-variable query is detachable because it has just one variable (P itself) in common with the rest of the query. Since it links up to the rest of the original query via the attribute P# (in the comparison SP.P# = P.P#), attribute P# is what must appear in the "proto tuple" (see Chapter 8) in the detached version; that is, the detached query must retrieve exactly the part numbers of red parts weighing less than 25 pounds. We save that detached query as a query, Query D1, that retrieves its result into a temporary relvar P' (the effect of the INTO clause is to cause a new relvar P', with sole attribute P#, to be defined automatically to hold the result of executing the RETRIEVE). Finally, we replace references to P in the reduced version of Query Q0 by references to P'. Let us refer to this new reduced version as Query Q1:

```
Q1: RETRIEVE ( S.SNAME ) WHERE S.CITY = "London"
                         AND   S.S#   = SP.S#
                         AND   SP.QTY > 200
                         AND   SP.P#  = P'.P#
```

We now perform a similar process of detachment on Query Q1, detaching the one-variable query involving variable SP as Query D2 and leaving a modified version of Q1 (Query Q2):

```
D2: RETRIEVE INTO SP' ( SP.S#, SP.P# ) WHERE SP.QTY > 200

Q2: RETRIEVE ( S.SNAME ) WHERE S.CITY = "London"
                         AND   S.S#    = SP'.S#
                         AND   SP'.P# = P'.P#
```

Next we detach the one-variable query involving S:

```
D3: RETRIEVE INTO S' ( S.S#, S.SNAME ) WHERE S.CITY = "London"

Q3: RETRIEVE ( S'.SNAME ) WHERE S'.S#  = SP'.S#
                          AND   SP'.P# = P'.P#
```

Finally, we detach the two-variable query involving SP' and P':

```
D4: RETRIEVE INTO SP'' ( SP'.S# ) WHERE SP'.P# = P'.P#

Q4: RETRIEVE ( S'.SNAME ) WHERE S'.S# = SP''.S#
```

Thus, the original query Q0 has been decomposed into three one-variable queries D1, D2, and D3 (each of which is a projection of a restriction) and two two-variable queries D4 and Q4 (each of which is a projection of a join). We can represent the situation at this point by means of the tree structure shown in Fig. 18.3. That figure can be read as follows:

- Queries D1, D2, and D3 take as input relvars P, SP, and S (more precisely, the relations that are the current values of relvars P, SP, and S), respectively, and produce as output P', SP', and S', respectively.
- Query D4 then takes as input P' and SP' and produces as output SP".
- Finally, Query Q4 takes as input S' and SP" and produces as output the overall required result.

Observe now that Queries D1, D2, and D3 are completely independent of one another and can be processed in any order (possibly even in parallel). Likewise, Queries D3 and D4 can be processed in either order once Queries D1 and D2 have been processed. However, Queries D4 and Q4 cannot be decomposed any further and must be processed by tuple substitution (which really just means *brute force, index lookup,* or *hash lookup*—see Section 18.7). For example, consider Query Q4. With our usual sample data, the set of

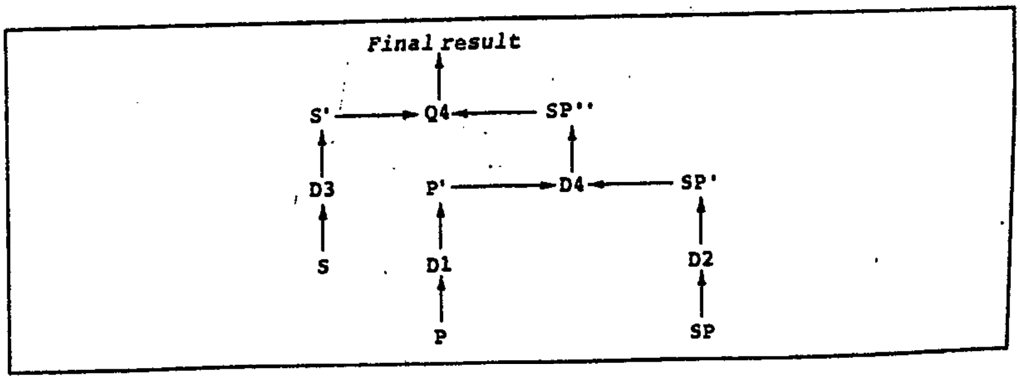

Fig. 18.3 Decomposition tree for Query Q0

supplier numbers in attribute SP".S# will be the set {S1,S2,S4}. Each of these three values will be substituted for SP".S# in turn. Query Q4 will therefore be evaluated as if it had been written as follows:

```
RETRIEVE ( S'.SNAME ) WHERE S'.S# = "S1"
                      OR    S'.S# = "S2"
                      OR    S'.S# = "S4"
```

Reference [18.34] gives algorithms for breaking the original query down into smaller queries and choosing variables for tuple substitution. It is in that latter choice that much of the actual optimization resides; reference [18.34] includes heuristics for making the cost estimates that drive the choice (Ingres will usually—but not always—choose the relation with the smallest cardinality as the one to do the substitution on). The principal objectives of the optimization process as a whole are to avoid having to build Cartesian products and to keep the number of tuples to be scanned to a minimum at each stage.

Reference [18.34] does not discuss the optimization of one-variable queries. However, information regarding that level of optimization is given in the Ingres overview paper [8.11]. Basically, it is similar to the analogous function in other systems, involving as it does the use of statistical information kept in the catalog and the choice of a particular access path (e.g., a hash or index) for scanning the data as stored. Reference [18.35] presents experimental evidence—measurements from a benchmark set of queries—that suggests that the Ingres optimization techniques sketched in this section are basically sound and in practice quite effective. Some specific conclusions from that paper are the following:

1. Detachment is the best first move.
2. If tuple substitution *must* be done first, then the best choice of variable to be substituted for is a join variable.
3. Once tuple substitution has been applied to one variable in a two-variable query, it is an excellent tactic to build an index or hash "on the fly," if necessary, on the join attribute in the other relation (Ingres in fact often applies this tactic).

18.7 IMPLEMENTING THE RELATIONAL OPERATORS

We now present a short description of some straightforward methods for implementing certain of the relational operators, join in particular. Our primary reason for including this material is simply to remove any remaining air of mystery that might possibly still surround the optimization process. The methods to be discussed correspond to what we called "low-level procedures" in Section 18.3. *Note:* More sophisticated techniques are described in the annotation to some of the references at the end of the chapter. See also Appendix A.

We assume for simplicity that tuples and relations are physically stored as such. The operators we consider are project, join, and summarize—where we take "summarize" to include both of the following cases:

1. The PER operand specifies no attributes at all ("PER TABLE_DEE").
2. The PER operand specifies at least one attribute.

Case 1 is straightforward: Basically, it involves scanning the entire relation over which the summarizing is to be done—except that, if the attribute to be aggregated happens to be indexed, it might be possible to compute the result directly from the index, without having to access the relation itself at all. For example, the expression

```
SUMMARIZE SP ADD SUM ( QTY ) AS TQ
```

can be evaluated by scanning the QTY index (assuming such an index exists) without touching the shipments *per se* at all. An analogous remark applies if SUM is replaced by COUNT or AVG (for COUNT, any index will do). As for MAX and MIN, the result can be found *in a single access* to the last index entry (for MAX) or the first (for MIN), assuming again that an index exists for the relevant attribute.

For the rest of this section we take "summarize" to mean Case 2 specifically. Here is an example of Case 2:

```
SUMMARIZE SP PER P { P# } ADD SUM ( QTY ) AS TOTQTY
```

From the user's point of view, project, join, and summarize (Case 2) are very different from one another. From an implementation point of view, however, they do have certain similarities, because in every case the system needs to group tuples on the basis of common values for specified attributes. In the case of projection, such grouping allows the system to eliminate duplicates; in the case of join, it allows it to find matching tuples; and in the case of summarize, it allows it to compute the individual aggregate values for each group. There are several techniques for performing such grouping:

1. Brute force
2. Index lookup
3. Hash lookup
4. Merge
5. Hash
6. Combinations of 1–5

Figs. 18.4–18.8 give pseudocode procedures for the case of join specifically (project and summarize are left as an exercise). The notation used in those figures is as follows: *R* and *S* are the relations to be joined; *C* is their (possibly composite) common attribute. We assume it is possible to access the tuples of each of *R* and *S* one at a time in some sequence, and we denote those tuples, in that sequence, by *R*[1], *R*[2], ..., *R*[*m*] and *S*[1], *S*[2], ..., *S*[*n*], respectively. We use the expression *R*[*i*] * *S*[*j*] to denote the joined tuple formed from the tuples *R*[*i*] and *S*[*j*]. Finally, we refer to *R* and *S* as the outer and inner relation, respectively (because they control the outer and inner loop, respectively).

Brute Force

Brute force is what might be termed "the plain case," in which all possible tuple combinations are inspected (i.e., every tuple of *R* is examined in conjunction with every tuple of *S*, as indicated in Fig. 18.4). *Note:* Brute force is often referred to as "nested loops" in the literature, but this name is misleading because nested loops are in fact involved in all of the algorithms.

```
do i := 1 to m ;                                  /* outer loop */
   do j := 1 to n ;                               /* inner loop */
      if R[i].C = S[j].C then
      add joined tuple R[i] * S[j] to result ;
   end ;
end ;
```

Fig. 18.4 Brute force

Let us examine the costs associated with the brute-force approach. *Note:* We limit our attention here to I/O costs only, although other costs (e.g., CPU costs) might also be important in practice.

First of all, the approach clearly requires a total of $m + (m * n)$ tuple reads. But what about tuple writes?—that is, what is the cardinality of the joined result? (The number of tuple writes will be equal to that cardinality if the result is written back out to the disk.)

- In the common special case of a many-to-one join (in particular, a foreign-to-matching-candidate-key join), the cardinality of the result is clearly equal to the cardinality, m or n, of whichever of R and S represents the foreign-key side of the join.
- Now consider the more general case of a many-to-many join. Let dCR be the number of distinct values of the join attribute C in relation R, and let dCS be defined analogously. If we assume *uniform value distributions* (so that any given value of C in relation R is as likely to occur as any other), then for a given tuple of R there will be n/dCS tuples of S with the same value for C as that tuple; hence, the total number of tuples in the join (i.e., the cardinality of the result) will be $(m * n)/dCS$. Or, if we start by considering a given tuple of S instead of R, the total number will be $(n * m)/dCR$; the two estimates will differ if $dCR \neq dCS$ (i.e., if there are some values of C that occur in R but not in S or *vice versa*), in which case the lower estimate is the one to use.

In practice, as stated in Section 18.2, it is *page* I/O's that matter, not tuple I/O's. Suppose, therefore, that the tuples of R and S are stored pR to a page and pS to a page, respectively (so that the two relations occupy m/pR pages and n/pS pages, respectively). Then it is easy to see that the procedure of Fig. 18.4 will involve $(m/pR) + (m * n)/pS$ page reads. Alternatively, if we interchange the roles of R and S (making S the outer relation and R the inner), the number of page reads will be $(n/pS) + (n * m)/pR$.

By way of example, suppose $m = 100$, $n = 10{,}000$, $pR = 1$, $pS = 10$. Then the two formulas evaluate to 100,100 and 1,001,000 page reads, respectively. *Conclusion:* It is desirable in the brute-force approach for the smaller relation of the two to be chosen as the outer relation (where *smaller* means "smaller number of pages").

We conclude this brief discussion of the brute-force technique by observing that it should be regarded as a worst-case procedure; it assumes that relation S is neither indexed nor hashed on the join attribute C. Experiments by Bitton *et al.* [18.6] indicate that, if that assumption is in fact valid, matters will usually be improved by constructing an index or hash on the fly and then proceeding with an index- or hash-lookup join (see the next two subsections). Reference [18.35] supports this idea, as mentioned at the end of the previous section.

Index Lookup

We now consider the case in which there is an index *X* on attribute *C* of the inner relation *S* (refer to Fig. 18.5). The advantage of this technique over brute force is that for a given tuple of the outer relation *R* we can go "directly" to the matching tuples of the inner relation *S*. The total number of tuple reads on relations *R* and *S* is thus simply the cardinality of the joined result; making the worst-case assumption that every tuple read on *S* is in fact a separate page read, the total number of page reads is thus $(m/pR) + ((m * n)/dCS)$.

```
/* assume index X on S.C */
                                              /* outer loop */
do i := 1 to m ;
   /* let there be k index entries X[1], ..., X[k] with */
   /* indexed attribute value = R[i].C                   */
   do j := 1 to k ;                           /* inner loop */
      /* let tuple of S indexed by X[j] be S[j] */
      add joined tuple R[i] * S[j] to result ;
   end ;
end ;
```

Fig. 18.5 Index lookup

If relation *S* happens to be stored in sequence by values of the join attribute *C*, however, the page-read figure reduces to $(m/pR) + ((m * n)/dCS)/pS$. Taking the same sample values as before ($m = 100$, $n = 10{,}000$, $pR = 1$, $pS = 10$), and assuming $dCS = 100$, the two formulas evaluate to 10,100 and 1,100, respectively. The difference between these two figures clearly points up the importance of keeping stored relations in a "good" physical sequence [18.7].

However, we must include the overhead for accessing the index *X* itself. The worst-case assumption is that each tuple of *R* requires an "out of the blue" index lookup to find the matching tuples of *S*, which implies reading one page from each level of the index. For an index of *x* levels, this will add an extra $m * x$ page reads to the overall page-read figure. In practice, *x* will typically be 3 or less (moreover, the top level of the index will very likely reside in main memory throughout processing, thereby reducing the page-read figure still further).

Hash Lookup

Hash lookup is similar to index lookup, except that the "fast access path" to the inner relation *S* on the join attribute *S.C* is a hash instead of an index (refer to Fig. 18.6). Derivation of cost estimates for this case is left as an exercise.

```
/* assume hash table H on S.C */

do i := 1 to m ;                                  /* outer loop */
   k := hash (R[i].C) ;
   /* let there be h tuples S[1], ..., S[h] stored at H[k] */
   do j := 1 to h ;                               /* inner loop */
      if S[j].C = R[i].C then
      add joined tuple R[i] * S[j] to result ;
   end ;
end ;
```

Fig. 18.6 Hash lookup

Merge

The merge technique assumes that the two relations *R* and *S* are both physically stored in sequence by values of the join attribute *C*. If such is in fact the case, the two relations can be scanned in physical sequence, the two scans can be synchronized, and the entire join can be done in a single pass over the data (at least, this claim is true if the join is one-to-many; it might not be quite true for the many-to-many case). Such a technique is unquestionably optimal under our stated assumptions, because every page is accessed just once (refer to Fig. 18.7). In other words, the number of page reads is just $(m/pR) + (n/pS)$. It follows that:

- Physical clustering of logically related data is one of the most critical performance factors of all: that is, it is highly desirable that data be clustered in such a way as to match the joins that are most important to the enterprise [18.7].
- In the absence of such clustering, it is often a good idea to sort either or both relations at run time and then do a merge join anyway (of course, the effect of such sorting is precisely to produce the desired clustering dynamically). This technique is referred to, logically enough, as **sort/merge** [18.8].

```
/* assume R and S are both sorted on attribute C ; */
/* following code assumes join is many-to-many ;   */
/* simpler many-to-one case is left as an exercise */

r := 1 ;
s := 1 ;
do while r ≤ m and s ≤ n ;                       /* outer loop */
   v := R[r].C ;
   do j := s by 1 while S[j].C < v ;
   end ;
   s := j ;
   do j := s by 1 while S[j].C = v ;    /* main inner loop */
      do i := r by 1 while R[i].C = v ;
         add joined tuple R[i] * S[j] to result ;
      end ;
   end ;
   s := j ;
   do i := r by 1 while R[i].C = v ;
   end ;
   r := i ;
end ;
```

Fig. 18.7 Merge (many-to-many case)

Hash

Like the merge technique just discussed, the hash technique requires a single pass over each of the two relations (refer to Fig. 18.8). The first pass builds a hash table for relation *S* on values of the join attribute *S.C*; the entries in that table contain the join-attribute value—possibly other attribute values also—and a pointer to the corresponding tuple on the disk. The second pass then scans relation *R* and applies the same hash function to the join attribute *R.C*. When an *R* tuple collides in the hash table with one or more *S* tuples, the algorithm checks to see that the values of *R.C* and *S.C* are indeed equal, and if so generates the appropriate joined tuple(s). The great advantage of this technique over the merge technique is that relations *R* and *S* do not need to be stored in any particular order, and no sorting is necessary.

As with the hash-lookup technique, we leave the derivation of cost estimates for this approach as an exercise.

```
/* build hash table H on S.C */

do j := 1 to n ;
   k := hash (S[j].C) ;
   add S[j] to hash table entry H[k] ;
end ;

/* now do hash lookup on R */
```

Fig. 18.8 Hash

18.8 SUMMARY

Optimization represents both a challenge and an opportunity for relational systems. In fact, optimizability is a strength of such systems, for several reasons: a relational system with a good optimizer might well outperform a nonrelational system. Our introductory example gave some idea of the kind of improvement that might be achievable (a factor of over 10,000 to 1 in that particular case). The four broad stages of optimization are:

1. Cast the query into some internal form (typically a **query tree** or **abstract syntax tree**, but such representations can be thought of as just an internal form of the relational algebra or relational calculus).
2. Convert to **canonical form**, using various **laws of transformation**.
3. Choose candidate **low-level procedures** for implementing the various operations in the canonical representation of the query.
4. Generate **query plans** and choose the cheapest, using **cost formulas** and knowledge of **database statistics**.

Next, we discussed the general **distributive**, **commutative**, and **associative** laws and their applicability to relational operators such as join (also their applicability to **arithmetic**, **logical**, and **comparison operators**), and we mentioned the **idempotence** and

absorption laws. We also discussed some specific transformations for the **restriction** and **projection** operators. Then we introduced the important idea of **semantic transformations**—that is, transformations based on the system's knowledge of **integrity constraints.**

By way of illustration, we sketched some of the database statistics maintained by the DB2 and Ingres products. Then we described a divide-and-conquer strategy called **query decomposition** (which was introduced with the Ingres prototype), and we mentioned that such strategies might be very attractive in a parallel-processing or distributed environment.

Finally, we examined certain **implementation techniques** for certain of the relational operators, especially **join.** We presented pseudocode algorithms for five join techniques—**brute force, index lookup, hash lookup, merge** (including **sort/merge**), and **hash**—and briefly considered the costs associated with these techniques.

We should not close this section without mentioning the fact that many of today's products do unfortunately include certain **optimization inhibitors,** which users should at least be aware of (even though there is little they can do about them, in most cases). An optimization inhibitor is a feature of the system in question that prevents the optimizer from doing as good a job as it might do otherwise (i.e., in the absence of that feature). The inhibitors in question include *duplicate rows* (see reference [6.6]), *three-valued logic* (see Chapter 19), and *SQL's implementation of three-valued logic* (see references [19.6] and [19.10]).

One last point: In this chapter, we have discussed optimization as conventionally understood and conventionally implemented; in other words, we have described "the conventional wisdom." More recently, however, a radically new approach to DBMS implementation has emerged, an approach that has the effect of invalidating many of the assumptions underlying that conventional wisdom. As a consequence, many aspects of the overall optimization process can be simplified (even eliminated entirely, in some cases), including:

- The use of *cost-based access path selection* (Stages 3 and 4 of the process)
- The use of *indexes* and other conventional access paths
- The choice between *compiling* and *interpreting* database requests
- The algorithms for *implementing the relational operators*

and many others. See Appendix A for further discussion.

EXERCISES

18.1 Some of the following pairs of expressions on the suppliers-parts-projects database are equivalent and some not. Which ones are?

```
a1. S JOIN ( ( P JOIN J ) WHERE CITY = 'London' )

a2. ( P WHERE CITY = 'London' ) JOIN ( J JOIN S )

b1. ( S MINUS ( ( S JOIN SPJ ) WHERE P# = P# ('P2') )
                { S#, SNAME, STATUS, CITY } ) { S#, CITY }

b2. S { S#, CITY } MINUS
     ( S { S#, CITY } JOIN
       ( SPJ WHERE P# = P# ('P2') ) ) { S#, CITY }
```

c1.
```
( S { CITY } MINUS P { CITY } ) MINUS J { CITY }
```
c2.
```
( S { CITY } MINUS J { CITY } )
  MINUS ( P { CITY } MINUS J { CITY } )
```
d1.
```
( J { CITY } INTERSECT P { CITY } ) UNION S { CITY }
```
d2.
```
J { CITY } INTERSECT ( S { CITY } UNION P { CITY } )
```
e1.
```
( ( SPJ WHERE S# = S# ('S1') )
                    UNION ( SPJ WHERE P# = P# ('P1') ) )
INTERSECT
( ( SPJ WHERE J# = J# ('J1') )
                    UNION ( SPJ WHERE S# = S# ('S1') ) )
```
e2.
```
( SPJ WHERE S# = S# ('S1') ) UNION
( ( SPJ WHERE P# = P# ('P1') ) INTERSECT
  ( SPJ WHERE J# = J# ('J1') ) )
```
f1.
```
( S WHERE CITY = 'London' ) UNION ( S WHERE STATUS > 10 )
```
f2.
```
S WHERE CITY = 'London' AND STATUS > 10
```
g1.
```
( S { S# } INTERSECT ( SPJ WHERE J# = J# ('J1') ) { S# } )
  UNION ( S WHERE CITY = 'London' ) { S# }
```
g2.
```
S { S# } INTERSECT ( ( SPJ WHERE J# = J# ('J1') ) { S# }
             UNION ( S WHERE CITY = 'London' ) { S# } )
```
h1.
```
( SPJ WHERE J# = J# ('J1') ) { S# }
  MINUS ( SPJ WHERE P# = P# ('P1') ) { S# }
```
h2.
```
( ( SPJ WHERE J# = J# ('J1') )
  MINUS ( SPJ WHERE P# = P# ('P1') ) ) { S# }
```
i1.
```
S JOIN ( P { CITY } MINUS J { CITY } )
```
i2.
```
( S JOIN P { CITY } ) MINUS ( S JOIN J { CITY } )
```

18.2 Show that join, union, and intersection are commutative and difference is not.

18.3 Show that join, union, and intersection are associative and difference is not.

18.4 Show that (a) union distributes over intersection; (b) intersection distributes over union.

18.5 Prove the absorption laws.

18.6 Show that (a) restriction is unconditionally distributive over union, intersection, and difference, and conditionally distributive over join; (b) projection is unconditionally distributive over union and intersection, is conditionally distributive over join, and is not distributive over difference. State the relevant conditions in the conditional cases.

18.7 Extend the transformation rules of Section 18.4 to take account of extend and summarize.

18.8 Can you find any useful transformation rules for the relational division operation?

18.9 Give an appropriate set of transformation rules for conditional expressions involving AND, OR, and NOT. An example of such a rule would be "commutativity of AND"—that is, *A* AND *B* is the same as *B* AND *A*.

18.10 Extend your answer to the previous exercise to include boolean expressions involving the quantifiers EXISTS and FORALL. An example of such a rule would be the rule given in Chapter 8 (Section 8.2) that allows an expression involving a FORALL to be converted into one involving a negated EXISTS instead.

18.11 Here is a list of integrity constraints for the suppliers-parts-projects database (extracted from the exercises in Chapter 9):

- The only legal cities are London, Paris, Rome, Athens, Oslo, Stockholm, Madrid, and Amsterdam.

- No two projects can be located in the same city.
- At most one supplier can be located in Athens at any one time.
- No shipment can have a quantity more than double the average of all such quantities.
- The highest-status supplier must not be located in the same city as the lowest-status supplier.
- Every project must be located in a city in which there is at least one supplier of that project.
- There must exist at least one red part.
- The average supplier status must be greater than 19.
- Every London supplier must supply part P2.
- At least one red part must weigh less than 50 pounds.
- Suppliers in London must supply more different kinds of parts than suppliers in Paris.
- Suppliers in London must supply more parts in total than suppliers in Paris.

And here are some sample queries against that database:

a. Get suppliers who do not supply part P2.
b. Get suppliers who do not supply any project in the same city as the supplier.
c. Get suppliers such that no supplier supplies fewer kinds of parts.
d. Get Oslo suppliers who supply at least two distinct Paris parts to at least two distinct Stockholm projects.
e. Get pairs of colocated suppliers who supply pairs of colocated parts.
f. Get pairs of colocated suppliers who supply pairs of colocated projects.
g. Get parts supplied to at least one project only by suppliers not in the same city as that project.
h. Get suppliers such that no supplier supplies more kinds of parts.

Use the integrity constraints to transform these queries into simpler forms (still in natural language, however; you are not asked to answer this exercise *formally*).

18.12 Investigate any DBMS that might be available to you. What expression transformations does that system perform? Does it perform any semantic transformations?

18.13 Try the following experiment: Take a simple query, say "Get names of suppliers who supply part P2," and state that query in as many different ways as you can think of in whatever query language is available to you (probably SQL). Create and populate a suitable test database, run the different versions of the query, and measure the execution times. If those times vary significantly, you have empirical evidence that the optimizer is not doing a very good job of expression transformation. Repeat the experiment with several different queries. If possible, repeat it with several different DBMSs also. *Note:* Of course, the different versions of the query should all give the same result. If not, you have probably made a mistake—or it might be an optimizer bug; if so, report it to the vendor!

18.14 Investigate any DBMS that might be available to you. What database statistics does that system maintain? How are they updated—in real time, or via some utility? If the latter, what is the utility called? How frequently is it run? How selective is it, in terms of the specific statistics that it can update on any specific execution?

18.15 We saw in Section 18.5 that among the database statistics maintained by DB2 are the second highest and second lowest value for each column of each base table. Why the *second* highest and lowest, do you think?

18.16 Several commercial products allow the user to provide hints to the optimizer. In DB2, for example, the specification OPTIMIZE FOR *n* ROWS on an SQL cursor declaration means the user expects to retrieve no more than *n* rows via the cursor in question (i.e., to execute FETCH against that cursor no more than *n* times). Such a specification can sometimes cause the optimizer to choose an access path that is more efficient, at least for the case where the user does in fact execute the FETCH no more than *n* times. Do you think such hints are a good idea? Justify your answer.

18.17 Devise a set of implementation procedures for the restriction and projection operations (along the lines of the procedures sketched for join in Section 18.7). Derive an appropriate set of cost formulas for those procedures. Assume that page I/O's are the only quantity of interest; that is, do not attempt to include CPU or other costs in your formulas. State and justify any other assumptions you make.

18.18 Read Appendix A and discuss.

REFERENCES AND BIBLIOGRAPHY

The field of optimization is huge, and growing all the time; the following list represents a relatively small selection from the vast literature on this subject. It is divided into groups, as follows:

- References [18.1–18.6] provide introductions to, or overviews of, the general optimization problem.
- References [18.7–18.14] are concerned with the efficient implementation of specific relational operations, such as join or summarize.
- References [18.15–18.32] describe a variety of techniques based on expression transformation, as discussed in Section 18.4 (in particular, references [18.25–18.28] consider *semantic* transformations).
- References [18.33–18.43] discuss the techniques used in System R, DB2, and Ingres, and the general problem of optimizing queries involving SQL-style nested subqueries.
- References [18.44–18.62] address a miscellaneous set of techniques, tricks, directions for future research, and so forth (in particular, references [18.55–18.58] consider the impact on optimization of parallel-processing techniques).

Note: Publications on optimization in distributed database and decision support systems are deliberately excluded. See Chapters 21 and 22, respectively.

18.1 Won Kim, David S. Reiner, and Don S. Batory (eds.): *Query Processing in Database Systems.* New York, N.Y.: Springer-Verlag (1985).

This book is an anthology of papers on the general topic of query processing (not just optimization). It consists of an introductory survey paper by Jarke, Koch, and Schmidt (similar but not identical to reference [18.2]), followed by groups of papers that discuss query processing in a variety of contexts: distributed databases, heterogeneous systems, view updating (reference [10.8] is the sole paper in this section), nontraditional applications (e.g., CAD/CAM), multi-statement optimization (see reference [18.47]), database machines, and physical database design.

18.2 Matthias Jarke and Jürgen Koch: "Query Optimization in Database Systems." *ACM Comp. Surv. 16,* No. 2 (June 1984).

An excellent early tutorial. The paper gives a general framework for query processing, much like the one in Section 18.3 of the present chapter, but based on the relational calculus rather

than the algebra. It then discusses a large number of optimization techniques within that framework: syntactic and semantic transformations, low-level operation implementation, and algorithms for generating query plans and choosing among them. An extensive set of syntactic transformation rules for calculus expressions is given. A lengthy bibliography (not annotated) is also included; note, however, that the number of papers on the subject published since 1984 is probably an order of magnitude greater than the number prior to that time.

The paper also briefly discusses certain other related issues: the optimization of higher-level query languages (i.e., languages that are more powerful than the algebra or calculus), optimization in a distributed-database environment, and the role of database machines with respect to optimization.

18.3 Götz Graefe: "Query Evaluation Techniques for Large Databases." *ACM Comp. Surv. 25*, No. 2 (June 1993).

Another excellent tutorial, more recent than reference [18.2], with an extensive bibliography. To quote the abstract: "This survey provides a foundation for the design and implementation of query execution facilities . . . It describes a wide array of practical query evaluation techniques . . . including iterative execution of complex query evaluation plans, the duality of sort- and hash-based set-matching algorithms, types of parallel query execution and their implementation, and special operators for emerging database application domains." Recommended.

18.4 Frank P. Palermo: "A Data Base Search Problem," in Julius T. Tou (ed.), *Information Systems: COINS IV*. New York, N.Y.: Plenum Press (1974).

One of the very earliest papers on optimization (in fact, a classic). Starting from an arbitrary expression of the relational calculus, the paper first uses Codd's reduction algorithm to reduce that expression to an equivalent algebraic expression (see Chapter 8), and then introduces a number of improvements on that algorithm, among them the following:

- No tuple is ever retrieved more than once.
- Unnecessary values are discarded from a tuple as soon as that tuple is retrieved—"unnecessary values" being either values of attributes not referenced in the query or values used solely for restriction purposes. This process is equivalent to projecting the relation over the "necessary" attributes, and thus not only reduces the space required for each tuple but also reduces the number of tuples that need to be retained (in general).
- The method used to build up the result is based on a least-growth principle, so that result tends to grow slowly. This technique has the effect of reducing both the number of comparisons involved and the amount of intermediate storage required.
- An efficient technique is employed in the construction of joins, involving (a) the dynamic factoring out of values used in join terms (such as S.S# = SP.S#) into *semijoins*, which are effectively a kind of dynamically constructed secondary index—note that Palermo's semijoins are not the same thing as the semijoins of Chapter 7, *q.v.*—and (b) the use of an internal representation of each join called an *indirect join*, which makes use of internal tuple IDs to identify the tuples that participate in the join. These techniques are designed to reduce the amount of scanning needed in the construction of the join, by ensuring for each join term that the tuples concerned are logically ordered on the values of the join attributes. They also permit the dynamic determination of the best sequence in which to access the required relations.

18.5 Meikel Poess and Chris Floyd: "New TPC Benchmarks for Decision Support and Web Commerce," *ACM SIGMOD Record 29*, No. 4 (December 2000).

TPC stands for the *Transaction Processing Council*, which is an independent body that has produced several industry-standard benchmarks over the years. *TPC-C* (which is modeled after an order/entry system) is a benchmark for measuring OLTP performance. *TPC-H* and *TPC-R* are decision support benchmarks: they are designed to measure performance on *ad hoc* queries (*TPC-H*) and planned reports (*TPC-R*), respectively. *TPC-W* is designed to measure performance in an e-commerce environment. See *http://www.tpc.org* for further information, including numerous actual benchmark results.

18.6 Dina Bitton, David J. DeWitt, and Carolyn Turbyfill: "Benchmarking Database Systems: A Systematic Approach." Proc. 9th Int. Conf. on Very Large Data Bases, Florence, Italy (October/November 1983).

The first paper to describe what is now usually called "the Wisconsin benchmark" (since it was developed by the authors of the paper at the University of Wisconsin). The benchmark defines a set of relations with precisely specified attribute values, and then measures the performance of certain precisely specified algebraic operations on those relations (for example, various projections, involving different degrees of duplication in the attributes over which the projections are taken). It thus represents a systematic test of the effectiveness of the optimizer on those fundamental operations.

18.7 M. W. Blasgen and K. P. Eswaran: "Storage and Access in Relational Databases," *IBM Sys. J. 16*, No. 4 (1977).

Several techniques for handling queries involving restriction, projection, and join operations are compared on the basis of their cost in disk I/O. The techniques in question are basically those implemented in System R [18.33].

18.8 T. H. Merrett: "Why Sort/Merge Gives the Best Implementation of the Natural Join." *ACM SIGMOD Record 13*, No. 2 (January 1983).

Presents a set of intuitive arguments to support the position statement of the title. The argument is essentially that:

a. The join operation itself will be most efficient if the two relations are each sorted on values of the join attribute (because in that case, as we saw in Section 18.7, merge is the obvious technique, and each data page will be retrieved exactly once, which is clearly optimal).

b. The cost of sorting the relations into that desired sequence, on a large enough machine, is likely to be less than the cost of any scheme for getting around the fact that they are not so sorted.

However, the author does admit that there could be some exceptions to his somewhat contentious position. For instance, one of the relations might be sufficiently small—that is, it might be the result of a previous restriction operation—that direct access to the other relation via an index or a hash could be more efficient than sorting it. References [18.9–18.11] give further examples of cases where sort/merge might not be the best technique in practice.

18.9 Giovanni Maria Sacco: "Fragmentation: A Technique for Efficient Query Processing," *ACM TODS 11*, No. 2 (June 1986).

Presents a divide-and-conquer method for performing joins by recursively splitting the relations to be joined into disjoint restrictions ("fragments") and performing a series of sequential scans on those fragments. Unlike sort/merge, the technique does not require the relations to be sorted first. The paper shows that the fragmentation technique always performs better than sort/merge in the case where sort/merge requires both relations to be sorted first, and usually performs better in the case where sort/merge requires just one relation (the larger) to be sorted first. The author claims that the technique can also be applied to other operations, such as intersection and difference.

18.10 Leonard D. Shapiro: "Join Processing in Database Systems with Large Main Memories," *ACM TODS 11*, No. 3 (September 1986).

Presents three hash join algorithms, one of which is "especially efficient when the main memory available is a significant fraction of the size of one of the relations to be joined." The algorithms work by splitting the relations into disjoint partitions (i.e., restrictions) that can be processed in main memory. The author contends that hash methods are destined to become the technique of choice, given the rate at which main-memory costs are decreasing.

18.11 M. Negri and G. Pelagatti: "Distributive Join: A New Algorithm for Joining Relations," *ACM TODS 16*, No. 4 (December 1991).

Another divide-and-conquer join method. "[The method] is based on the idea that . . . it is not necessary to sort both relations completely . . . It is sufficient to sort one completely and the other only partially, thus avoiding part of the sort effort." The partial sort breaks down the affected relation into a sequence of unsorted partitions *P1, P2, ..., Pn* (somewhat as in Sacco's method [18.9], except that Sacco uses hashing instead of sorting), with the property that MAX(*Pi*) < MIN(*P*(*i*+1)) for *i* = 1, 2, ..., *n*−1. The paper claims that this method performs better than sort/merge.

18.12 Götz Graefe and Richard L. Cole: "Fast Algorithms for Universal Quantification in Large Databases," *ACM TODS 20*, No. 2 (June 1995).

The universal quantifier FORALL is not directly supported in SQL and is therefore not directly implemented in current commercial DBMSs either, yet it is extremely important in formulating a wide class of queries. This paper describes and compares "three known algorithms and one recently proposed algorithm for relational division, [which is] the algebra operator that embodies universal quantification," and shows that the new algorithm runs "as fast as hash (semi-)join evaluates existential quantification over the same relations" (slightly reworded). The authors conclude among other things that FORALL should be directly supported in the user language because most optimizers "do not recognize the rather indirect formulations available in SQL."

18.13 David Simmen, Eugene Shekita, and Timothy Malkemus: "Fundamental Techniques for Order Optimization," Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

Presents techniques for optimizing or avoiding sorts. The techniques, which rely in part on the work of Darwen [11.7], have been implemented in DB2.

18.14 Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay: "Approximate Medians and Other Quantiles in One Pass and with Limited Memory," Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

18.15 César A. Galindo-Legaria and Milind M. Joshi: "Orthogonal Optimization of Subqueries and Aggregation," Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. (May 2001).

18.16 James Miles Smith and Philip Yen-Tang Chang: "Optimizing the Performance of a Relational Algebra Database Interface," *CACM 18*, No. 10 (October 1975).

Describes the algorithms used in the "Smart Query Interface for a Relational Algebra" (SQUIRAL). The techniques used include the following:

- Transforming the original algebraic expression into an equivalent but more efficient sequence of operations, along the lines discussed in Section 18.4

- Assigning distinct operations in the transformed expression to distinct processes and exploiting concurrency and pipelining among them
- Coordinating the sort orders of the temporary relations passed between those processes
- Exploiting indexes and attempting to localize page references

This paper and reference [18.17] were probably the first to discuss expression transformations.

18.17 P. A. V. Hall: "Optimisation of a Single Relational Expression in a Relational Data Base System." *IBM J. R&D 20*, No. 3 (May 1976).

This paper describes some of the optimizing techniques used in the system PRTV [7.9]. Like SQUIRAL [18.16], PRTV begins by transforming the given algebraic expression into some more efficient form before evaluating it. A feature of PRTV is that the system does not automatically evaluate each expression as soon as it receives it; rather, it defers actual evaluation until the last possible moment (see the discussion of step-at-a-time query formulation in Chapter 7, Section 7.5). Thus, the "single relational expression" of the paper's title might actually represent an entire sequence of user operations. The optimizations described resemble those of SQUIRAL but go further in some respects. They include the following (in order of application):

- Performing restrictions as early as possible
- Combining sequences of projections into a single projection
- Eliminating redundant operations
- Simplifying expressions involving empty relations and trivial conditions
- Factoring out common subexpressions

The paper concludes with some experimental results and some suggestions for further investigations.

18.18 Matthias Jarke and Jürgen Koch: "Range Nesting: A Fast Method to Evaluate Quantified Queries," Proc. 1983 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif. (May 1983).

Defines a version of the relational calculus that permits some additional transformation rules to be applied, and presents algorithms for evaluating expressions of that calculus. (Actually, the version in question is quite close to the tuple calculus as presented in Chapter 8.) The paper describes the optimization of a certain class of expressions of the revised calculus, called "perfect nested expressions." Methods are given for converting apparently complex queries—in particular, certain queries involving FORALL—into perfect expressions. The authors show that a large subset of the queries that arise in practice correspond to perfect expressions.

18.19 Surajit Chaudhuri and Kyuseok Shim: "Including Group-By in Query Optimization." Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

18.20 A. Makinouchi, M. Tezuka, H. Kitakami, and S. Adachi: "The Optimization Strategy for Query Evaluation in RDB/V1," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

RDB/V1 was the prototype forerunner of the Fujitsu product AIM/RDB. This paper describes the optimization techniques used in that prototype and briefly compares them with the techniques used in the Ingres and System R prototypes. One particular technique seems to be novel: the use of dynamically obtained MAX and MIN values to induce additional restrictions. This technique has the effect of simplifying the process of choosing a join order and improving the performance of the joins themselves. As a simple example of the latter point, suppose suppliers and parts are to be joined over cities. First, the suppliers are sorted on CITY; during the sort, the maximum and minimum values, HIGH and LOW say, of S.CITY are determined. Then the restriction

```
LOW ≤ P.CITY AND P.CITY ≤ HIGH
```

can be used to reduce the number of parts that need to be inspected in building the join.

18.21 Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan: "Extensible Rule Based Query Rewrite Optimization in Starburst," Proc. 1992 ACM SIGMOD Int. Conf. on Management of Data, San Diego, Calif. (June 1992).

As noted in Section 18.1, "query rewrite" is expression transformation by another name. The authors claim that, rather surprisingly, commercial products do little in the way of such transformation (at least as of 1992). Be that as it may, the paper describes the expression transformation mechanism of the IBM Starburst prototype (see references [18.48], [26.19], [26.23], and [26.29, 26.30]). Suitably qualified users can add new transformation rules to the system at any time (hence the "extensible" of the paper's title).

18.22 Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan: "Magic Is Relevant," Proc. 1990 ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, N.J. (May 1990).

The infelicitous term *magic* refers to an optimization technique originally developed for use with queries—especially ones involving recursion—expressed in Datalog (see Chapter 24). The present paper extends the approach to conventional relational systems, claiming on the basis of experimental measurements that it is often more effective than traditional optimization techniques (note that the query does not have to be recursive for the approach to be applicable). The basic idea is to decompose the given query into a number of smaller queries that define a set of "auxiliary relations" (somewhat as in the query decomposition approach discussed in Section 18.6), in such a way as to filter out tuples that are irrelevant to the problem at hand. The following example (expressed in relational calculus) is based on one given in the paper. The original query is:

```
{ EX.ENAME }
     WHERE EX.JOB = 'Clerk' AND
           EX.SAL > AVG ( EY WHERE EY.DEPT# = EX.DEPT#, SAL )
```

("Get names of clerks whose salary is greater than the average for their department"). If this expression is evaluated as written, the system will scan the employees tuple by tuple and hence compute the average salary for any department that employs more than one clerk several times. A traditional optimizer might therefore break down the query into the following two smaller queries:

```
WITH ( EX.DEPT#,
       AVG ( EY WHERE
             EY.DEPT# = EX.DEPT#, SAL ) AS ASAL } AS T1 :

{ EMP.ENAME } WHERE EMP.JOB = 'Clerk' AND
                    EXISTS T1 ( EMP.DEPT# = T1.DEPT# AND
                                EMP.SALARY > T1.ASAL )
```

Now no department's average will be computed more than once, but some *irrelevant* averages will be computed—namely, those for departments that do not employ clerks.

The "magic" approach avoids both the repeated computations of the first approach and the irrelevant computations of the second, at the cost of generating extra "auxiliary" relations:

```
/* first auxiliary relation : name, department, and salary  */
/* for clerks                                               */
WITH ( { EMP.ENAME, EMP.DEPT#, EMP.SAL }
       WHERE EMP.JOB = 'Clerk' ) AS T1 :
```

```
/* second auxiliary relation : departments employing clerks */
WITH ( T1.DEPT# ) AS T2 :

/* third auxiliary relation : departments employing clerks  */
/* and corresponding average salaries                       */
WITH ( ( T2.DEPT#,
         AVG ( EMP WHERE
               EMP.DEPT# = T2.DEPT#, SAL ) AS ASAL ) ) AS T3 :

/* result relation */
( T1.ENAME ) WHERE EXISTS T3 ( T1.DEPT# = T3.DEPT# AND
                               T1.SAL > T3.ASAL )
```

The "magic" consists in determining exactly which auxiliary relations are needed.

See references [18.23, 18.24] immediately following and the "References and Bibliography" section in Chapter 24 for further references to "magic."

18.23 Inderpal Singh Mumick and Hamid Pirahesh: "Implementation of Magic in Starburst." Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data. Minneapolis. Minn. (May 1994).

18.24 Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan: "Magic Conditions," *ACM TODS 21*, No. 1 (March 1996).

18.25 Jonathan J. King: "QUIST: A System for Semantic Query Optimization in Relational Databases." Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

The paper that introduced the idea of semantic optimization (see Section 18.4). It describes a prototype implementation called QUIST ("QUery Improvement through Semantic Transformation").

18.26 Sreekumar T. Shenoy and Z. Meral Ozsoyoglu: "A System for Semantic Query Optimization," Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May/June 1987).

Extends the work of King [18.25] by introducing a scheme that dynamically selects, from a potentially very large set of integrity constraints, just those that are likely to be useful in transforming a given query. The integrity constraints considered are of two basic kinds, called *implication constraints* and *subset constraints* in the paper. Such constraints are used to transform queries by eliminating redundant restrictions and joins and introducing additional restrictions on indexed attributes. Cases in which the query can be answered from the constraints alone are also handled efficiently.

18.27 Michael Siegel, Edward Sciore, and Sharon Salveter: "A Method for Automatic Rule Derivation to Support Semantic Query Optimization," *ACM TODS 17*, No. 4 (December 1992).

As explained in Section 18.4, semantic optimization makes use of integrity constraints to transform queries. However, there are several problems associated with this idea:

- How does the optimizer know which transformations will be effective (i.e., will make the query more efficient)?
- Some integrity constraints are not very useful for optimization purposes. For example, the constraint that part weights must be greater than zero, though important for integrity purposes, is essentially useless for optimization. How does the optimizer distinguish between useful and useless constraints?
- Some conditions might be valid for some states of the database—even for most states—and hence be useful for optimization purposes, and yet not strictly be integrity constraints as such. An example might be the condition "employee age is less than or equal to 50"; though

not an integrity constraint *per se* (employees can be older than 50), it might well be the case that no current employee is in fact older than 50.

This paper describes the architecture for a system that addresses the foregoing issues.

18.28 Upen S. Chakravarthy, John Grant, and Jack Minker: "Logic Based Approach to Semantic Query Optimization," *ACM TODS 15*, No. 2 (June 1990).

To quote from the abstract: "In several previous papers [the authors have] described and proved the correctness of a method for semantic query optimization . . . This paper consolidates the major results of those papers, emphasizing the techniques and their applicability for optimizing relational queries. Additionally, [it shows] how this method subsumes and generalizes earlier work on semantic query optimization. [It also indicates] how semantic query optimization techniques can be extended to [recursive queries] and integrity constraints that contain disjunction, negation, and recursion."

18.29 Qi Cheng *et al.*: "Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database," Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland (September 1999).

18.30 A. V. Aho, Y. Sagiv, and J. D. Ullman: "Efficient Optimization of a Class of Relational Expressions," *ACM TODS 4*, No. 4 (December 1979).

The relational expressions referred to in the title of this paper involve only equality restrictions ("selections"), projections, and natural joins: so-called *SPJ-expressions*. SPJ-expressions correspond to calculus queries in which the *<bool exp>* in the WHERE clause involves only equality comparisons, ANDs, and existential quantifiers. The paper introduces *tableaus* as a means of symbolically representing SPJ-expressions. A tableau is a rectangular array, in which columns correspond to attributes and rows to conditions: specifically, to *membership conditions*, which state that a certain (sub)tuple exists in a certain relation. Rows are logically connected by the appearance of common symbols in the rows concerned. For example, the tableau

S#	STATUS	CITY	P#	COLOR	
	f1				
b1	*f1*	London			— suppliers
b1			*b2*		— shipments
			b2	Red	— parts

represents the query "Get status (*f1*) of suppliers (*b1*) in London who supply some red part (*b2*)." The top row of the tableau lists all attributes mentioned in the query, the next row is the "summary" row (corresponding to the proto tuple in a calculus query or the final projection in an algebraic query), and the remaining rows (as already stated) represent membership conditions. We have tagged those rows in the example to indicate the relevant relations (or relvars, rather). Notice that the "*b*"s refer to bound variables and the "*f*"s to free variables: the summary row contains only "*f*"s.

Tableaus represent another candidate for a canonical formalism for queries (see Section 18.3), except that they are not general enough to represent all possible relational expressions. (In fact, they can be regarded as a syntactic variation on Query-By-Example, one that is however strictly less powerful than QBE.) The paper gives algorithms for reducing any tableau to another, semantically equivalent tableau in which the number of rows is reduced to a minimum. Since the number of rows (not counting the top two, which are special) is one more than the number of joins in the corresponding SPJ-expression, the converted tableau represents an optimal form of the query—optimal, in the very specific sense that the number of joins is minimized. (In the example, however, the number of joins is already the minimum possible for the query, and such optimization has no effect.) The minimal tableau can then be converted if desired into some other representation for subsequent additional optimization.

The idea of minimizing the number of joins has applicability to queries formulated in terms of join views (in particular, queries formulated in terms of a "universal relation"—see the "References and Bibliography" section in Chapter 13). For example, suppose the user is presented with a view V that is defined as the join of suppliers and shipments over S#, and the user issues the query:

```
V { P# }
```

A straightforward view-processing algorithm would convert this query into the following:

```
( SP JOIN S ) { P# }
```

As pointed out in Section 18.4, however, the following query produces the same result, and does not involve a join (i.e., the number of joins has been minimized):

```
SP { P# }
```

Note therefore that, since the algorithms for tableau reduction given in the paper take into account any explicitly stated functional dependencies among the attributes, those algorithms provide a limited example of a *semantic* optimization technique.

18.31 Y. Sagiv and M. Yannakakis: "Equivalences Among Relational Expressions with the Union and Difference Operators," *JACM 27*, No. 4 (October 1980).

Extends the ideas of reference [18.30] to include queries that make use of union and difference operations.

18.32 Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv: "Query Optimization by Predicate Move-Around." Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

18.33 P. Griffiths Selinger *et al.*: "Access Path Selection in a Relational Database System," Proc. 1979 ACM SIGMOD Int. Conf.on Management of Data, Boston, Mass. (May/June 1979).

This seminal paper discusses the optimization techniques used in System R. A query in System R is an SQL statement and thus consists of a set of "SELECT - FROM - WHERE" blocks (*query blocks*), some of which might be nested inside others. The System R optimizer first decides on an order in which to execute those query blocks; it then seeks to minimize the total cost of the query by choosing the cheapest implementation for each individual block. Note that this strategy (choosing block order first, then optimizing individual blocks) means that certain possible query plans will never be considered; in effect, it amounts to a technique for "reducing the search space" (see the remarks on this subject near the end of Section 18.3). *Note:* In the case of nested blocks, the optimizer effectively just follows the nested order as specified by the user—that is, the innermost block will be executed first, loosely speaking. See references [18.37–18.43] for criticism and further discussion of this strategy.

For a given query block, there are basically two cases to consider (the first of which can be regarded as a special case of the second):

1. For a block that involves just a restriction and/or projection of a single relation, the optimizer uses statistical information from the catalog, together with formulas (given in the paper) for estimating intermediate result sizes and low-level operation costs, to choose a strategy for performing that restriction and/or projection.
2. For a block that involves two or more relations to be joined together, with (probably) local restrictions and/or projections as well, the optimizer (a) treats each individual relation as in

Case 1 and (b) decides on a sequence for performing the joins. The two operations *a* and *b* are not independent of one another; for example, a given strategy—using a certain index, say—for accessing an individual relation *A* might well be chosen precisely because it produces tuples of *A* in the order in which they are needed to perform a subsequent join of *A* with some other relation *B*.

Joins are implemented by sort/merge, index lookup, or brute force. The paper stresses the point that, in evaluating (for example) the nested join (*A* JOIN *B*) JOIN *C*, it is not necessary to compute the join of *A* and *B* in its entirety before computing the join of the result and *C*; on the contrary, as soon as any tuple of *A* JOIN *B* has been produced, it can immediately be passed to the process that joins such tuples with tuples of *C*. Thus, it might never be necessary to materialize the relation "*A* JOIN *B*" in its entirety at all. (This general ***pipelining*** idea was discussed briefly in Chapter 3, Section 3.2. See also references [18.16] and [18.58].)

The paper also includes a few observations on the cost of optimization. For a join of two relations, the cost is said to be approximately equal to the cost of between 5 and 20 database retrievals, a negligible overhead if the optimized query will subsequently be executed a large number of times. (Note that System R is a compiling system—in fact, it pioneered the compiling approach—and hence an SQL statement might be optimized once and then executed many times, perhaps many thousands of times.) Optimization of complex queries is said to require "only a few thousand bytes of storage and a few tenths of a second" on an IBM System 370 Model 158. "Joins of eight tables have been optimized in a few seconds."

18.34 Eugene Wong and Karel Youssefi: "Decomposition—A Strategy for Query Processing," *ACM TODS 1*, No. 3 (September 1976).

18.35 Karel Youssefi and Eugene Wong: "Query Processing in a Relational Database Management System," Proc. 5th Int. Conf. on Very Large Data Bases, Rio de Janeiro, Brazil (September 1979).

18.36 Lawrence A. Rowe and Michael Stonebraker: "The Commercial Ingres Epilogue," in reference [8.10].

"Commercial Ingres" is the product that grew out of the "University Ingres" prototype. Some of the differences between the University and Commercial Ingres optimizers are as follows:

1. The University optimizer used "incremental planning"—that is, it decided what to do first, did it, decided what to do next on the basis of the size of the result of the previous step, and so on. The Commercial optimizer decides on a complete plan before beginning execution, based on estimates of intermediate result sizes.
2. The University optimizer handled two-variable (i.e., join) queries by tuple substitution, as explained in Section 18.6. The Commercial optimizer supports a variety of preferred techniques for handling such queries, including in particular the sort/merge technique described in Section 18.7.
3. The Commercial optimizer uses a much more sophisticated set of statistics than the University optimizer.
4. The University optimizer did incremental planning, as noted under point 1. The Commercial optimizer does a more exhaustive search. However, the search process stops if the time spent on optimization exceeds the current best estimate of the time required to execute the query (for otherwise the overhead of doing the optimization might well outweigh the advantages).
5. The Commercial optimizer considers all possible index combinations, all possible join sequences, and "all available join methods—sort/merge, partial sort/merge, hash lookup, ISAM lookup, B-tree lookup, and brute force" (see Section 18.7).

18.37 Won Kim: "On Optimizing an SQL-Like Nested Query," *ACM TODS 7*, No. 3 (September 1982).

See the annotation to reference [18.41].

18.38 Werner Kiessling: "On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates," Proc. 11th Int. Conf. on Very Large Data Bases, Stockholm, Sweden (August 1985).

See the annotation to reference [18.41].

18.39 Richard A. Ganski and Harry K. T. Wong: "Optimization of Nested SQL Queries Revisited," Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

See the annotation to reference [18.41].

18.40 Günter von Bültzingsloewen: "Translating and Optimizing SQL Queries Having Aggregates," Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, UK (September 1987).

See the annotation to reference [18.41].

18.41 M. Muralikrishna: "Improved Unnesting Algorithms for Join Aggregate SQL Queries," Proc. 18th Int. Conf. on Very Large Data Bases, Vancouver, Canada (August 1992).

The SQL language includes the concept of a "nested subquery"—that is, a SELECT - FROM - WHERE block that is nested inside another such block, loosely speaking (see Chapter 8). This construct has caused implementers much grief. Consider the following SQL query ("Get names of suppliers who supply part P2"), which we will refer to as Query Q1:

```
SELECT S.SNAME
FROM   S
WHERE  S.S# IN
     ( SELECT SP.S#
       FROM   SP
       WHERE  SP.P# = P# ('P2') ) ;
```

In System R [18.33], this query is implemented by (a) evaluating the inner block first to yield a temporary table, T say, containing supplier *numbers* for the required suppliers, and then (b) searching table S one row at a time, and, for each such row, searching table T to see if it contains the corresponding supplier number. This strategy is likely to be quite inefficient (especially as table T will not be indexed).

Now consider the following query (Query Q2):

```
SELECT S.SNAME
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = P# ('P2') ;
```

This query is readily seen to be semantically identical to the previous one, but System R will now consider additional implementation strategies for it. In particular, if tables S and SP happen to be physically stored in supplier number sequence, it will use a merge join, which will be very efficient. And given that (a) the two queries are logically equivalent but (b) the second is more immediately susceptible to efficient implementation, the possibility of transforming queries of type Q1 into queries of type Q2 seems worth exploring. That possibility is the subject of references [18.37–18.43].

Kim [18.37] was the first to address the problem. Five types of nested queries were identified and corresponding transformation algorithms described. Kim's paper included some experimental measurements that showed that the proposed algorithms improved the performance of nested queries by (typically) one to two orders of magnitude.

Subsequently, Kiessling [18.38] showed that Kim's algorithms did not work correctly if a nested subquery (at any level) included a COUNT operator in its SELECT list (it did not properly handle the case where the COUNT argument evaluated to an empty set). The "semantic reefs" of the paper's title referred to the SQL awkwardnesses and complexities that users have to navigate around in order to get correct answers to such queries. Furthermore, Kiessling also showed that Kim's algorithm was not easy to fix ("there seems to be no uniform way to do these transformations efficiently and correctly under all circumstances").

The paper by Ganski and Wong [18.39] provides a fix to the problem identified by Kiessling, by using an *outer join* (see Chapter 19) instead of the regular inner join in the transformed version of the query. (The fix is not totally satisfactory, in the present writer's opinion, because it introduces an undesirable ordering dependence among the operators in the transformed query.) The paper also identifies a further bug in Kim's original paper, which it fixes in the same way. However, the transformations in this paper contain additional bugs of their own, some having to do with the problem of duplicate rows (a notorious "semantic reef") and others with the flawed behavior of the SQL EXISTS quantifier (see Chapter 19).

The paper by von Bültzingsloewen [18.40] represents an attempt to put the entire topic on a theoretically sound footing (the basic problem being that, as several writers have observed, the behavior—both syntactic and semantic—of SQL-style nesting and aggregation is not well understood). It defines extended versions of both the relational calculus and the relational algebra (the extensions having to do with aggregates and nulls), and proves the equivalence of those two extended formalisms (using, incidentally, a new method of proof that seems more elegant than those previously published). It then defines the semantics of SQL by mapping SQL into the extended calculus just defined. However, it should be noted that:

1. The dialect of SQL discussed, though closer to the dialect typically supported in commercial products than that of references [18.37–18.39], is still not fully orthodox; It does not include UNION, it does not directly support operators of the form "=ALL" or ">ALL" (see Appendix B), and its treatment of *unknown* truth values—see Chapter 19—is different from (actually better than) that of conventional SQL.

2. The paper omits consideration of matters having to do with duplicate elimination "for technical simplification." But the implications of this omission are not clear, given that (as already indicated) the possibility of duplicates has significant consequences for the validity or otherwise of certain transformations [6.6].

Finally, Muralikrishna [18.41] claims that Kim's original algorithm [18.37], though incorrect, can still be more efficient than "the general strategy" of reference [18.39] in some cases, and therefore proposes an alternative correction to Kim's algorithm. It also provides some additional improvements.

18.42 Lars Baekgaard and Leo Mark: "Incremental Computation of Nested Relational Query Expressions." *ACM TODS 20*, No. 2 (June 1995).

Another paper on the optimization of queries involving SQL-style subqueries, especially *correlated* ones. The strategy is (1) to convert the original query into an unnested equivalent and then (2) to evaluate the unnested version incrementally. "To support step (1), we have developed a very concise algebra-to-algebra transformation algorithm . . . The [transformed] expression makes intensive use of the [MINUS] operator. To support step (2), we present and analyze an efficient algorithm for incrementally evaluating [MINUS operations]." The term *incremental computation* refers to the idea that evaluation of a given query can make use of previously computed results.

18.43 Jun Rao and Kenneth A. Ross: "Using Invariants: A New Strategy for Correlated Queries," Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

Yet another paper on the optimization of queries involving SQL-style subqueries.

18.44 David H. D. Warren: "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France (September 1981).

Presents a view of query optimization from a rather different perspective: namely, that of formal logic. The paper reports on techniques used in an experimental database system based on Prolog. The techniques are apparently very similar to those of System R, although they were arrived at quite independently and with somewhat different objectives. The paper suggests that, in contrast to conventional query languages such as QUEL and SQL, logic-based languages such as Prolog permit queries to be expressed in such a manner as to highlight:

- What the essential components of the query are: namely, the logic goals
- What it is that interrelates those components: namely, the logic variables
- What the crucial implementation problem is: namely, the sequence in which to try to satisfy the goals

As a consequence, it is suggested that such a language is very convenient as a base for optimization. Indeed, it could be regarded as yet another candidate for the internal representation of queries originally expressed in some other language (see Section 18.3).

18.45 Yannis E. Ioannidis and Eugene Wong: "Query Optimization by Simulated Annealing," Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (May 1987).

The number of possible query plans grows exponentially with the number of relations involved in the query. In conventional commercial applications, the number of relations in a query tends to be small and so the number of candidate plans (the "search space") usually stays within reasonable bounds. In newer applications, however, the number of relations in a query can easily become quite large (see Chapter 22). Furthermore, such applications are also likely to need "global" (i.e., multi-query) optimization [18.47] and recursive query support, both of which also have the potential for increasing the search space significantly. Exhaustive search rapidly becomes out of the question in such an environment; some effective technique of reducing the search space becomes imperative.

The present paper gives references to previous work on the problems of optimization for large numbers of relations and multi-query optimization, but claims that no previous algorithms have been published for recursive-query optimization. It then presents an algorithm that it claims is suitable whenever the search space is large, and in particular shows how to apply that algorithm to the recursive case. The algorithm (called *simulated annealing* because it models the annealing process by which crystals are grown by first heating the containing fluid and then allowing it to cool gradually) is a probabilistic, hill-climbing algorithm that has successfully been applied to optimization problems in other contexts.

18.46 Arun Swami and Anoop Gupta: "Optimization of Large Join Queries," Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, Ill. (June 1988).

The general problem of determining the optimal join order in queries involving large numbers of relations (as arise in connection with, e.g., deductive database systems—see Chapter 24) is combinatorially hard. This paper presents a comparative analysis of a number of algorithms that address this problem: perturbation walk, quasi-random sampling, iterative improvement, sequence heuristic, and simulated annealing [18.45] (the names add a pleasing element of poetry to a subject that might otherwise be thought a trifle prosaic). According to that analysis, iterative improvement is superior to all of the other algorithms; in particular, simulated annealing is not useful "by itself" for large join queries.

18.47 Timos K. Sellis: "Multiple-Query Optimization," *ACM TODS 13*, No. 1 (March 1988).

Classical optimization research has focused on the problem of optimizing individual relational expressions in isolation. In future, however, the ability to optimize several distinct queries as a unit is likely to become important. One reason for this state of affairs is that what starts out as a single query at some higher level of the system might give rise to several queries at the relational level. For example, the natural language query "Is Mike well paid?" might require the execution of three separate relational queries:

- "Does Mike earn more than $75,000?"
- "Does Mike earn more than $60,000 and have less than five years of experience?"
- "Does Mike earn more than $45,000 and have less than three years of experience?"

This example illustrates the point that sets of related queries are likely to share some common subexpressions, and hence lend themselves to global optimization.

The paper considers queries involving conjunctions of restrictions and/or equijoins only. Some encouraging experimental results are included, and directions are identified for future research.

18.48 Guy M. Lohman: "Grammar-Like Functional Rules for Representing Query Optimization Alternatives." Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, Ill. (June 1988).

In some respects, a relational optimizer can be regarded as an expert system; however, the rules that drive the optimization process have historically been embedded in procedural code, not separately and declaratively stated. As a consequence, extending the optimizer to incorporate new optimization techniques has not been easy. Future database systems (see Chapter 26) will exacerbate this problem, because there will be a clear need for individual installations to extend the optimizer to incorporate, for example, support for specific user-defined data types. Several researchers have therefore proposed structuring the optimizer as a conventional expert system, with explicitly stated declarative rules.

However, this idea suffers from certain performance problems. In particular, a large number of rules might be applicable at any given stage during query processing, and determining the appropriate one might involve complex computation. The present paper describes an alternative approach (implemented in the Starburst prototype—see reference [18.21], also references [26.19], [26.23], and [26.29, 26.30]), in which the rules are stated in the form of production rules in a grammar somewhat like the grammars used to describe formal languages. The rules, called STARs (STrategy Alternative Rules), permit the recursive construction of query plans from other plans and "low-level plan operators" (LOLEPOPs), which are basic operations on relations such as join, sort, and so on. LOLEPOPs come in various *flavors;* for example, the join LOLEPOP has a sort/merge flavor, a hash flavor, and so on.

The paper claims that the foregoing approach has several advantages: The rules (STARs) are readily understandable by people who need to define new ones, the process of determining which rule to apply in any given situation is simpler and more efficient than the more traditional expert-system approach, and the extensibility objective is met.

18.49 Ryohei Nakano: "Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions," *ACM TODS 15*, No. 4 (December 1990).

As explained in Chapter 8 (Section 8.4), queries in a calculus-based language can be implemented by (a) translating the query under consideration into an equivalent algebraic expression, then (b) optimizing that algebraic expression, and finally (c) implementing that optimized expression. In this paper, Nakano proposes a scheme for combining steps *a* and *b* into a single step, thereby translating a given calculus expression directly into an *optimal* algebraic equivalent. This scheme is claimed to be "more effective and more promising . . . because it seems quite difficult to optimize complicated algebraic expressions." The translation process makes use of certain *heuristic* transformations, incorporating human knowledge regarding the equivalence of certain calculus and algebraic expressions.

18.50 Kyu-Young Whang and Ravi Krishnamurthy: "Query Optimization in a Memory-Resident Domain Relational Calculus Database System," *ACM TODS 15*, No. 1 (March 1990).

The most expensive aspect of query processing (in the specific main-memory environment assumed by this paper) is shown to be the evaluation of boolean expressions. Optimization in that environment is thus aimed at minimizing the number of such evaluations.

18.51 Johann Christoph Freytag and Nathan Goodman: "On the Translation of Relational Queries into Iterative Programs," *ACM TODS 14*, No. 1 (March 1989).

Presents methods for compiling relational expressions directly into executable code in a language such as C or Pascal. Note that this approach differs from the approach discussed in the body of the chapter, where the optimizer effectively combines *prewritten* (parameterized) code fragments to build the query plan.

18.52 Kiyoshi Ono and Guy M. Lohman: "Measuring the Complexity of Join Enumeration in Query Optimization," Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (August 1990).

Given that join is basically a dyadic operation, the optimizer has to break down a join involving n relations ($n > 2$) into a sequence of dyadic joins. Most optimizers do this in a strictly nested fashion; that is, they choose a pair of relations to join first, then a third to join to the result of joining the first two, and so on. In other words, an expression such as *A* JOIN *B* JOIN *C* JOIN *D* might be treated as, say, ((*D* JOIN *B*) JOIN *C*) JOIN *A*, but never as, say, (*A* JOIN *D*) JOIN (*B* JOIN *C*). Further, traditional optimizers are usually designed to avoid Cartesian products if at all possible. Both of these tactics can be seen as ways of "reducing the search space" (though heuristics for choosing the sequence of joins are still needed, of course).

The present paper describes the relevant aspects of the optimizer in the IBM Starburst prototype (see references [18.21], [18.48], [26.19], [26.23], and [26.29, 26.30]). It argues that both of the foregoing tactics can be inappropriate in certain situations, and hence that what is needed is an *adaptable* optimizer that can use, or be instructed to use, different tactics for different queries.

18.53 Bennet Vance and David Maier: "Rapid Bushy Join-Order Optimization with Cartesian Products," Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (June 1996).

As noted in the annotation to reference [18.52], optimizers tend to "reduce the search space" by (among other things) avoiding plans that involve Cartesian products. This paper shows that searching the entire space "is more affordable than has been previously recognized" and that avoiding Cartesian products is not necessarily beneficial (in this connection, see the discussion of "star join" in Chapter 22). According to the authors, the paper's main contributions are in (a) fully separating join-order enumeration from predicate analysis and (b) presenting "novel implementation techniques" for addressing the join-order enumeration problem.

18.54 Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis: "Parametric Query Optimization," Proc. 18th Int. Conf. on Very Large Data Bases, Vancouver, Canada (August 1992).

Consider the following query:

```
EMP WHERE SALARY > salary
```

(where *salary* is a run-time parameter). Suppose there is an index on SALARY. Then:

- If *salary* is $10,000 per month, then the best way to implement the query is to use the index (because presumably most employees will not qualify).
- If *salary* is $1,000 per month, then the best way to implement the query is by a sequential scan (because presumably most employees *will* qualify).

This example illustrates the point that some optimization decisions are best made at run time, even in a compiling system. The present paper explores the possibility of generating *sets* of query plans at compile time (each plan being "optimal" for some subset of the set of all possible values of the run-time parameters), and then choosing the appropriate plan at run time when the actual parameter values are known. It considers one specific run-time parameter in particular, the amount of buffer space available to the query in main memory. Experimental results show that the approach described imposes very little time overhead on the optimization process and sacrifices very little in terms of quality of the generated plans; accordingly, it is claimed that the approach can significantly improve query performance. "The savings in execution cost of using a plan that is specifically tailored to actual parameter values . . . could be enormous."

18.55 Navin Kabra and David J. DeWitt: "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans," Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. (June 1998).

18.56 Jim Gray: "Parallel Database Systems 101," Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif. (May 1995).

This is not a research paper but an extended abstract for a tutorial presentation. The basic idea behind parallel systems in general is to break a large problem into a lot of smaller ones that can be solved simultaneously, thereby improving performance (throughput and response time). Relational database systems in particular are highly amenable to parallelization because of the nature of the relational model: It is conceptually easy (a) to break relations down into subrelations in a variety of ways and (b) to break relational expressions down into subexpressions, again in a variety of ways. In the spirit of the title of this reference, we offer a few words on certain important parallel-database-system concepts.

First of all, the architecture of the underlying hardware will itself presumably involve some kind of parallelism. There are three principal architectures, each involving several processing units, several disk drives, and an interconnection network of some kind:

- *Shared memory:* The network allows all of the processors to access the same memory.
- *Shared disk:* Each processor has its own memory, but the network allows all of the processors to access all of the disks.
- *Shared nothing:* Each processor has its own memory and disks, but the network allows the processors to communicate with each other.

In practice, shared nothing is usually the architecture of choice, at least for large systems (the other two approaches quickly run into problems of *interference* as more and more processors are added). To be specific, shared nothing provides both linear speed-up (increasing hardware by a given factor improves response time by the same factor) and linear scale-up (increasing both hardware and data volume by the same factor keeps response time constant). *Note:* "Scale-up" is also known as scalability.

There are also several approaches to *data partitioning* (i.e., breaking a relation *r* into partitions or subrelations and assigning those partitions to *n* different processors):

- *Range partitioning:* Relation *r* is divided into disjoint partitions 1, 2, ..., *n* on the basis of values of some subset *s* of the attributes of *r* (conceptually, *r* is sorted on *s* and the result divided into *n* equal-size partitions). Partition *i* is then assigned to processor *i*. This approach is good for queries involving equality or range restrictions on *s*.

- *Hash partitioning:* Each tuple *t* of *r* is assigned to processor *i* = *h*(*t*), where *h* is some hash function. This approach is good for queries involving an equality restriction on the hashed attribute(s), also for queries that involve sequential access to the entire relation *r*.
- *Round-robin partitioning:* Conceptually, *r* is sorted in some way; the *i*th tuple in the sorted result is then assigned to processor *i* modulo *n*. This approach is good for queries that involve sequential access to the entire relation *r*.

Parallelism can apply to the execution of an individual operation (*intraoperation* parallelism), to the execution of distinct operations within the same query (*interoperation* or *intraquery* parallelism), and to the execution of distinct queries (*interquery* parallelism). Reference [18.3] includes a tutorial on all of these possibilities, and references [18.57, 18.58] discuss some specific techniques and algorithms. We remark that a parallel version of *hash join* (see Section 18.7) is particularly effective and widely used in practice.

18.57 Dina Bitton, Haran Boral, David J. DeWitt, and W. Kevin Wilkinson: "Parallel Algorithms for the Execution of Relational Database Operations," *ACM TODS 8*, No. 3 (September 1983).

Presents algorithms for implementing sort, projection, join, aggregation, and update operations in a multi-processor environment. The paper gives general cost formulas that take into account I/O, message, and processor costs, and can be adjusted to different multi-processor architectures.

18.58 Waqar Hasan and Rajeev Motwani: "Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism," Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

18.59 Donald Kossmann and Konrad Stocker: "Iterative Dynamic Programming: A New Class of Optimization Algorithms," *ACM TODS 25*, No. 1 (March 2000).

18.60 Parke Godfrey, Jarek Gryz, and Calisto Zuzarte: "Exploiting Constraint-Like Data Characterizations in Query Optimization," Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. (May 2001).

18.61 Alin Deutsch, Lucian Poppa, and Val Tannen: "Physical Data Independence, Constraints, and Optimization with Universal Plans," Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland (September 1999).

18.62 Michael Stillger, Guy Lohman, Volker Markl, and Mohtar Kandil: "LEO—DB2's LEarning Optimizer," Proc. 27th Int. Conf. on Very Large Data Bases, Rome, Italy (September 2001).