Chapter 16

The Query Compiler

We shall now take up the architecture of the query compiler and its optimizer. As we noted in Fig. 15.2, there are three broad steps that the query processor must take:

1. The query, written in a language like SQL, is *parsed*, that is, turned into a parse tree representing the structure of the query in a useful way.

2. The parse tree is transformed into an expression tree of relational algebra (or a similar notation), which we term a *logical query plan*.

3. The logical query plan must be turned into a *physical query plan*, which indicates not only the operations performed, but the order in which they are performed, the algorithm used to perform each step, and the ways in which stored data is obtained and data is passed from one operation to another.

The first step, parsing, is the subject of Section 16.1. The result of this step is a parse tree for the query. The other two steps involve a number of choices. In picking a logical query plan, we have opportunities to apply many different algebraic operations, with the goal of producing the best logical query plan. Section 16.2 discusses the algebraic laws for relational algebra in the abstract. Then, Section 16.3 discusses the conversion of parse trees to initial logical query plans and shows how the algebraic laws from Section 16.2 can be used in strategies to improve the initial logical plan.

When producing a physical query plan from a logical plan, we must evaluate the predicted cost of each possible option. Cost estimation is a science of its own, which we discuss in Section 16.4. We show how to use cost estimates to evaluate plans in Section 16.5, and the special problems that come up when we order the joins of several relations are the subject of Section 16.6. Finally, Section 16.7 covers additional issues and strategies for selecting the physical query plan: algorithm choice, and pipelining versus materialization.

16.1 Parsing and Preprocessing

The first stages of query compilation are illustrated in Fig. 16.1. The four boxes in that figure correspond to the first two stages of Fig. 15.2.

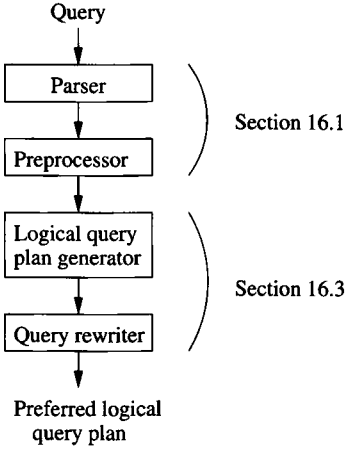

Figure 16.1: From a query to a logical query plan

In this section, we discuss parsing of SQL and give rudiments of a grammar that can be used for that language. We also discuss how to handle a query that involves a virtual view and other steps of preprocessing.

16.1.1 Syntax Analysis and Parse Trees

The job of the parser is to take text written in a language such as SQL and convert it to a *parse tree*, which is a tree whose nodes correspond to either:

1. *Atoms*, which are lexical elements such as keywords (e.g., SELECT), names of attributes or relations, constants, parentheses, operators such as + or <, and other schema elements, or

2. *Syntactic categories*, which are names for families of query subparts that all play a similar role in a query. We shall represent syntactic categories by triangular brackets around a descriptive name. For example, <Query> will be used to represent some queries in the common select-from-where form, and <Condition> will represent any expression that is a condition; i.e., it can follow WHERE in SQL.

If a node is an atom, then it has no children. However, if the node is a syntactic category, then its children are described by one of the *rules* of the grammar for the language. We shall present these ideas by example. The details of how one designs grammars for a language, and how one "parses," i.e.,

turns a program or query into the correct parse tree, is properly the subject of a course on compiling.[1]

16.1.2 A Grammar for a Simple Subset of SQL

We shall illustrate the parsing process by giving some rules that describe a small subset of SQL queries.

Queries

The syntactic category <Query> is intended to represent (some of the) queries of SQL. We give it only one rule:

```
<Query> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>
```

Symbol ::= means "can be expressed as." The syntactic categories <SelList> and <FromList> represent lists that can follow SELECT and FROM, respectively. We shall describe limited forms of such lists shortly. The syntactic category <Condition> represents SQL conditions (expressions that are either true or false); we shall give some simplified rules for this category later.

Note this rule does not provide for the various optional clauses such as GROUP BY, HAVING, or ORDER BY, nor for options such as DISTINCT after SELECT, nor for query expressions using UNION, JOIN, or other binary operators.

Select-Lists

```
<SelList> ::= <Attribute> , <SelList>
<SelList> ::= <Attribute>
```

These two rules say that a select-list can be any comma-separated list of attributes: either a single attribute or an attribute, a comma, and any list of one or more attributes. Note that in a full SQL grammar we would also need provision for expressions and aggregation functions in the select-list and for aliasing of attributes and expressions.

From-Lists

```
<FromList> ::= <Relation> , <FromList>
<FromList> ::= <Relation>
```

Here, a from-list is defined to be any comma-separated list of relations. For simplification, we omit the possibility that elements of a from-list can be expressionsa, such as joins or subqueries. Likewise, a full SQL grammar would have to allow tuple variables for relations.

[1] Those unfamiliar with the subject may wish to examine A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 2007, although the examples of Section 16.1.2 should be sufficient to place parsing in the context of the query processor.

Conditions

The rules we shall use are:

```
<Condition> ::= <Condition> AND <Condition>
<Condition> ::= <Attribute> IN ( <Query> )
<Condition> ::= <Attribute> = <Attribute>
<Condition> ::= <Attribute> LIKE <Pattern>
```

Although we have listed more rules for conditions than for other categories, these rules only scratch the surface of the forms of conditions. We have omitted rules introducing operators `OR`, `NOT`, and `EXISTS`, comparisons other than equality and `LIKE`, constant operands, and a number of other structures that are needed in a full SQL grammar.

Base Syntactic Categories

Syntactic categories <Attribute>, <Relation>, and <Pattern> are special, in that they are not defined by grammatical rules, but by rules about the atoms for which they can stand. For example, in a parse tree, the one child of <Attribute> can be any string of characters that identifies an attribute of the current database schema. Similarly, <Relation> can be replaced by any string of characters that makes sense as a relation in the current schema, and <Pattern> can be replaced by any quoted string that is a legal SQL pattern.

Example 16.1: Recall two relations from the running movies example:

```
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
```

Our study of parsing and query rewriting will center around two versions of the query "find the titles of movies that have at least one star born in 1960." We identify stars born in 1960 by asking if their birthdate (a SQL string) ends in `'1960'`, using the `LIKE` operator.

One way to ask this query is to construct the set of names of those stars born in 1960 as a subquery, and ask about each `StarsIn` tuple whether the `starName` in that tuple is a member of the set returned by this subquery. The SQL for this variation of the query is shown in Fig. 16.2.

The parse tree for the query of Fig. 16.2, according to the grammar we have sketched, is shown in Fig. 16.3. At the root is the syntactic category <Query>, as must be the case for any parse tree of a query. Working down the tree, we see that this query is a select-from-where form; the select-list consists of only the attribute `movieTitle`, and the from-list is only the one relation `StarsIn`.

The condition in the outer `WHERE`-clause is more complex. It has the form of attribute-IN-parenthesized-query. The subquery has its own singleton select- and from-lists and a simple condition involving a `LIKE` operator. □

```
SELECT movieTitle
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '%1960'
);
```

Figure 16.2: Find the movies with stars born in 1960

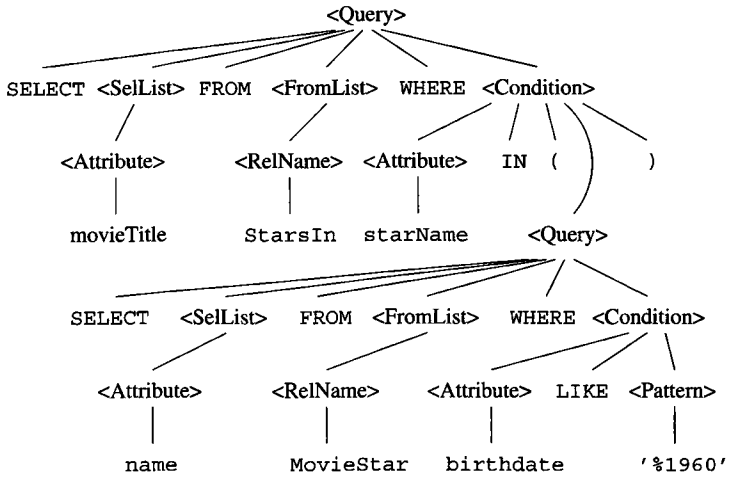

Figure 16.3: The parse tree for Fig. 16.2

Example 16.2: Now, let us consider another version of the query of Fig. 16.2, this time without using a subquery. We may instead equijoin the relations `StarsIn` and `MovieStar`, using the condition `starName = name`, to require that the star mentioned in both relations be the same. Note that `starName` is an attribute of relation `StarsIn`, while `name` is an attribute of `MovieStar`. This form of the query of Fig. 16.2 is shown in Fig. 16.4.[2]

The parse tree for Fig. 16.4 is seen in Fig. 16.5. Many of the rules used in this parse tree are the same as in Fig. 16.3. However, notice a from-list with more than one relation and two conditions connected by `AND`. □

[2]There is a small difference between the two queries in that Fig. 16.4 can produce duplicates if a movie has more than one star born in 1960. Strictly speaking, we should add `DISTINCT` to Fig. 16.4, but our example grammar was simplified to the extent of omitting that option.

```
SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name AND
    birthdate LIKE '%1960';
```

Figure 16.4: Another way to ask for the movies with stars born in 1960

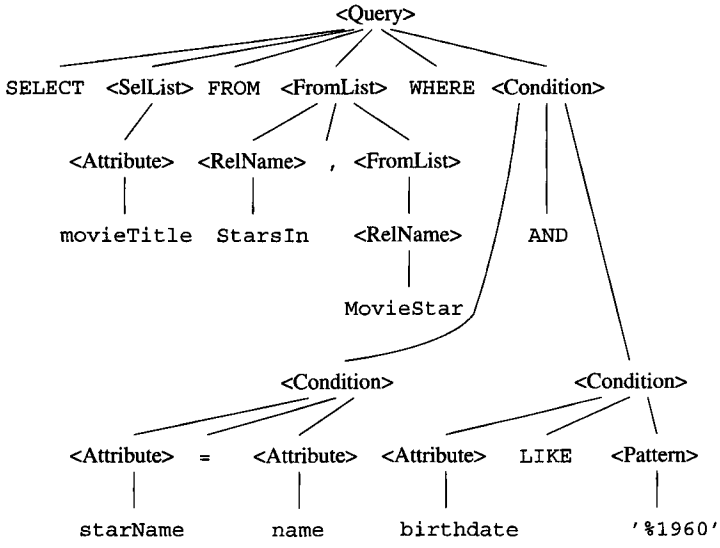

Figure 16.5: The parse tree for Fig. 16.4

16.1.3 The Preprocessor

The *preprocessor* has several important functions. If a relation used in the query is actually a virtual view, then each use of this relation in the from-list must be replaced by a parse tree that describes the view. This parse tree is obtained from the definition of the view, which is essentially a query. We discuss the preprocessing of view references in Section 16.1.4.

The preprocessor is also responsible for *semantic checking*. Even if the query is valid syntactically, it actually may violate one or more semantic rules on the use of names. For instance, the preprocessor must:

1. *Check relation uses*. Every relation mentioned in a `FROM`-clause must be a relation or view in the current schema.

2. *Check and resolve attribute uses*. Every attribute that is mentioned in the `SELECT`- or `WHERE`-clause must be an attribute of some relation in the current scope. For instance, attribute `movieTitle` in the first select-list of Fig. 16.3 is in the scope of only relation `StarsIn`. Fortunately,

`movieTitle` is an attribute of `StarsIn`, so the preprocessor validates this use of `movieTitle`. The typical query processor would at this point *resolve* each attribute by attaching to it the relation to which it refers, if that relation was not attached explicitly in the query (e.g., `StarsIn.movieTitle`). It would also check ambiguity, signaling an error if the attribute is in the scope of two or more relations with that attribute.

3. *Check types.* All attributes must be of a type appropriate to their uses. For instance, `birthdate` in Fig. 16.3 is used in a `LIKE` comparison, which requires that `birthdate` be a string or a type that can be coerced to a string. Since `birthdate` is a date, and dates in SQL normally can be treated as strings, this use of an attribute is validated. Likewise, operators are checked to see that they apply to values of appropriate and compatible types.

16.1.4 Preprocessing Queries Involving Views

When an operand in a query is a virtual view, the preprocessor needs to replace the operand by a piece of parse tree that represents how the view is constructed from base tables. The idea is illustrated in Fig. 16.6. A query Q is represented by its expression tree in relational algebra, and that tree may have some leaves that are views. We have suggested two such leaves, the views V and W. To interpret Q in terms of base tables, we find the definition of the views V and W. These definitions are also queries, so they can be expressed in relational algebra or as parse trees.

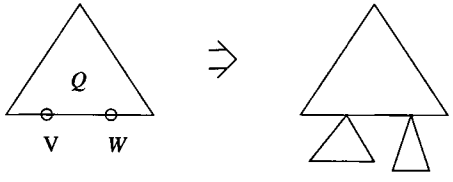

Figure 16.6: Substituting view definitions for view references

To form the query over base tables, we substitute, for each leaf in the tree for Q that is a view, the root of a copy of the tree that defines that view. Thus, in Fig. 16.6 we have shown the leaves labeled V and W replaced by the definitions of these views. The resulting tree is a query over base tables that is equivalent to the original query about views.

Example 16.3: Let us consider the view definition and query of Example 8.3. Recall the definition of view `ParamountMovies` is:

```
CREATE VIEW ParamountMovies AS
   SELECT title, year
```

```
FROM Movies
WHERE studioName = 'Paramount';
```

The tree in Fig. 16.7 is a relational-algebra expression for the query; we use relational algebra here because it is more succinct than the parse trees we have been using.

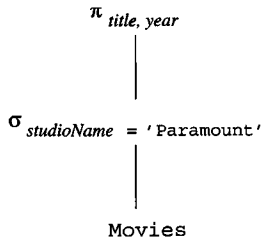

Figure 16.7: Expression tree for view `ParamountMovies`

The query of Example 8.3 is

```
SELECT title
FROM ParamountMovies
WHERE year = 1979;
```

asking for the Paramount movies made in 1979. This query has the expression tree shown in Fig. 16.8. Note that the one leaf of this tree represents the view `ParamountMovies`.

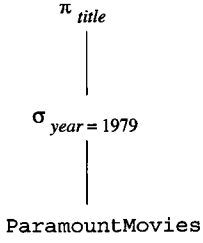

Figure 16.8: Expression tree for the query

We substitute the tree of Fig. 16.7 for the leaf `ParamountMovies` in Fig. 16.8. The resulting tree is shown in Fig. 16.9.

This tree, while the formal result of the view preprocessing, is not a very good way to express the query. In Section 16.2 we shall discuss ways to improve expression trees such as Fig. 16.9. In particular, we can push selections and projections down the tree, and combine them in many cases. Figure 16.10 is an improved representation that we can obtain by standard query-processing techniques. □

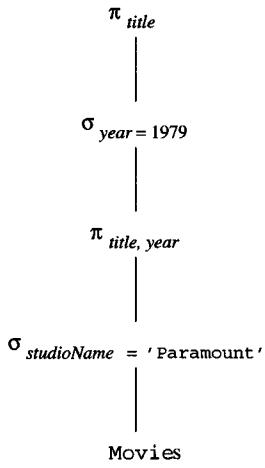

Figure 16.9: Expressing the query in terms of base tables

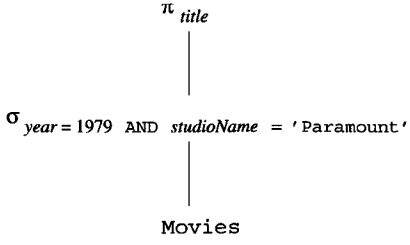

Figure 16.10: Simplifying the query over base tables

16.1.5 Exercises for Section 16.1

Exercise 16.1.1: Add to or modify the rules for <Query> to include simple versions of the following features of SQL select-from-where expressions:

a) The ability to produce a set with the `DISTINCT` keyword.

b) A `GROUP BY` clause and a `HAVING` clause.

c) Sorted output with the `ORDER BY` clause.

d) A query with no where-clause.

Exercise 16.1.2: Add to the rules for <Condition> to allow the following features of SQL conditionals:

a) Logical operators OR and NOT.

b) Comparisons other than =.

c) Parenthesized conditions.

d) EXISTS expressions.

Exercise 16.1.3: Using the simple SQL grammar exhibited in this section, give parse trees for the following queries about relations $R(a, b)$ and $S(b, c)$:

a)
```
SELECT a, c FROM R, S WHERE R.b = S.b;
```

b)
```
SELECT a FROM R WHERE b IN
    (SELECT a FROM R, S WHERE R.b = S.b);
```

16.2 Algebraic Laws for Improving Query Plans

We resume our discussion of the query compiler in Section 16.3, where we shall transform the parse tree into an expression of the extended relational algebra. Also in Section 16.3, we shall see how to apply heuristics that we hope will improve the algebraic expression of the query, using some of the many algebraic laws that hold for relational algebra. As a preliminary, this section catalogs algebraic laws that turn one expression tree into an equivalent expression tree that may have a more efficient physical query plan. The result of applying these algebraic transformations is the logical query plan that is the output of the query-rewrite phase.

16.2.1 Commutative and Associative Laws

A *commutative law* about an operator says that it does not matter in which order you present the arguments of the operator; the result will be the same. For instance, $+$ and $\times$ are commutative operators of arithmetic. More precisely, $x + y = y + x$ and $x \times y = y \times x$ for any numbers x and y. On the other hand, $-$ is not a commutative arithmetic operator: $x - y \neq y - x$.

An *associative law* about an operator says that we may group two uses of the operator either from the left or the right. For instance, $+$ and $\times$ are associative arithmetic operators, meaning that $(x + y) + z = x + (y + z)$ and $(x \times y) \times z = x \times (y \times z)$. On the other hand, $-$ is not associative: $(x - y) - z \neq x - (y - z)$. When an operator is both associative and commutative, then any number of operands connected by this operator can be grouped and ordered as we wish without changing the result. For example, $((w + x) + y) + z = (y + x) + (z + w)$.

Several of the operators of relational algebra are both associative and commutative. Particularly:

- $R \times S = S \times R$; $(R \times S) \times T = R \times (S \times T)$.
- $R \bowtie S = S \bowtie R$; $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$.
- $R \cup S = S \cup R$; $(R \cup S) \cup T = R \cup (S \cup T)$.

- $R \cap S = S \cap R$; $(R \cap S) \cap T = R \cap (S \cap T)$.

Note that these laws hold for both sets and bags. We shall not prove each of these laws, although we give one example of a proof, below.

Example 16.4: Let us verify the commutative law for $\bowtie$: $R \bowtie S = S \bowtie R$. First, suppose a tuple t is in the result of $R \bowtie S$, the expression on the left. Then there must be a tuple r in R and a tuple s in S that agree with t on every attribute that each shares with t. Thus, when we evaluate the expression on the right, $S \bowtie R$, the tuples s and r will again combine to form t.

We might imagine that the order of components of t will be different on the left and right, but formally, tuples in relational algebra have no fixed order of attributes. Rather, we are free to reorder components, as long as we carry the proper attributes along in the column headers, as was discussed in Section 2.2.5.

We are not done yet with the proof. Since our relational algebra is an algebra of bags, not sets, we must also verify that if t appears n times on the left, then it appears n times on the right, and vice-versa. Suppose t appears n times on the left. Then it must be that the tuple r from R that agrees with t appears some number of times n_R, and the tuple s from S that agrees with t appears some n_S times, where $n_R n_S = n$. Then when we evaluate the expression $S \bowtie R$ on the right, we find that s appears n_S times, and r appears n_R times, so we get $n_S n_R$ copies of t, or n copies.

We are still not done. We have finished the half of the proof that says everything on the left appears on the right, but we must show that everything on the right appears on the left. Because of the obvious symmetry, the argument is essentially the same, and we shall not go through the details here. □

We did not include the theta-join among the associative-commutative operators. True, this operator is commutative:

- $R \bowtie_C S = S \bowtie_C R$.

Moreover, if the conditions involved make sense where they are positioned, then the theta-join is associative. However, there are examples, such as the following, where we cannot apply the associative law because the conditions do not apply to attributes of the relations being joined.

Example 16.5: Suppose we have three relations $R(a, b)$, $S(b, c)$, and $T(c, d)$. The expression

$$(R \bowtie_{R.b>S.b} S) \bowtie_{a<d} T$$

is transformed by a hypothetical associative law into:

$$R \bowtie_{R.b>S.b} (S \bowtie_{a<d} T)$$

However, we cannot join S and T using the condition $a < d$, because a is an attribute of neither S nor T. Thus, the associative law for theta-join cannot be applied arbitrarily. □

Laws for Bags and Sets Can Differ

Be careful about applying familiar laws about sets to relations that are bags. For instance, you may have learned set-theoretic laws such as $A \cap_S (B \cup_S C) = (A \cap_S B) \cup_S (A \cap_S C)$, which is formally the "distributive law of intersection over union." This law holds for sets, but not for bags.

As an example, suppose bags A, B, and C were each $\{x\}$. Then $A \cap_B (B \cup_B C) = \{x\} \cap_B \{x, x\} = \{x\}$. But $(A \cap_B B) \cup_B (A \cap_B C) = \{x\} \cup_B \{x\} = \{x, x\}$, which differs from the left-hand-side, $\{x\}$.

16.2.2 Laws Involving Selection

Since selections tend to reduce the size of relations markedly, one of the most important rules of efficient query processing is to move the selections down the tree as far as they will go without changing what the expression does. Indeed early query optimizers used variants of this transformation as their primary strategy for selecting good logical query plans. As we shall see shortly, the transformation of "push selections down the tree" is not quite general enough, but the idea of "pushing selections" is still a major tool for the query optimizer.

To start, when the condition of a selection is complex (i.e., it involves conditions connected by AND or OR), it helps to break the condition into its constituent parts. The motivation is that one part, involving fewer attributes than the whole condition, may be moved to a convenient place where the entire condition cannot be evaluated. Thus, our first two laws for σ are the *splitting laws*:

- $\sigma_{C_1 \text{ AND } C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$.
- $\sigma_{C_1 \text{ OR } C_2}(R) = (\sigma_{C_1}(R)) \cup_S (\sigma_{C_2}(R))$.

However, the second law, for OR, works only if the relation R is a set. Notice that if R were a bag, the set-union would have the effect of eliminating duplicates incorrectly.

Notice that the order of C_1 and C_2 is flexible. For example, we could just as well have written the first law above with C_2 applied after C_1, as $\sigma_{C_2}(\sigma_{C_1}(R))$. In fact, more generally, we can swap the order of any sequence of σ operators:

- $\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R))$.

Example 16.6: Let $R(a, b, c)$ be a relation. Then $\sigma_{(a=1 \text{ OR } a=3) \text{ AND } b<c}(R)$ can be split as $\sigma_{a=1 \text{ OR } a=3}(\sigma_{b<c}(R))$. We can then split this expression at the OR into $\sigma_{a=1}(\sigma_{b<c}(R)) \cup \sigma_{a=3}(\sigma_{b<c}(R))$. In this case, because it is impossible for a tuple to satisfy both $a = 1$ and $a = 3$, this transformation holds regardless

of whether or not R is a set, as long as $\cup_B$ is used for the union. However, in general the splitting of an OR requires that the argument be a set and that $\cup_S$ be used.

Alternatively, we could have started to split by making $\sigma_{b<c}$ the outer operation, as $\sigma_{b<c}\big(\sigma_{a=1 \text{ OR } a=3}(R)\big)$. When we then split the OR, we would get $\sigma_{b<c}\big(\sigma_{a=1}(R) \cup \sigma_{a=3}(R)\big)$, an expression that is equivalent to, but somewhat different from the first expression we derived. □

The next family of laws involving σ allow us to push selections through the binary operators: product, union, intersection, difference, and join. There are three types of laws, depending on whether it is optional or required to push the selection to each of the arguments:

1. For a union, the selection *must* be pushed to both arguments.
2. For a difference, the selection must be pushed to the first argument and optionally may be pushed to the second.
3. For the other operators it is only required that the selection be pushed to one argument. For joins and products, it may not make sense to push the selection to both arguments, since an argument may or may not have the attributes that the selection requires. When it is possible to push to both, it may or may not improve the plan to do so; see Exercise 16.2.1.

Thus, the law for union is:

- $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$.

Here, it is mandatory to move the selection down both branches of the tree.

For difference, one version of the law is:

- $\sigma_C(R - S) = \sigma_C(R) - S$.

However, it is also permissible to push the selection to both arguments, as:

- $\sigma_C(R - S) = \sigma_C(R) - \sigma_C(S)$.

The next laws allow the selection to be pushed to one or both arguments. If the selection is σ_C, then we can only push this selection to a relation that has all the attributes mentioned in C, if there is one. We shall show the laws below assuming that the relation R has all the attributes mentioned in C.

- $\sigma_C(R \times S) = \sigma_C(R) \times S$.
- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$.
- $\sigma_C(R \bowtie_D S) = \sigma_C(R) \bowtie_D S$.
- $\sigma_C(R \cap S) = \sigma_C(R) \cap S$.

If C has only attributes of S, then we can instead write:

- $\sigma_C(R \times S) = R \times \sigma_C(S)$.

and similarly for the other three operators $\bowtie$, $\bowtie_D$, and $\cap$. Should relations R and S both happen to have all attributes of C, then we can use laws such as:

- $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie \sigma_C(S)$.

Note that it is impossible for this variant to apply if the operator is $\times$ or $\bowtie_D$, since in those cases R and S have no shared attributes. On the other hand, for $\cap$ this form of law always applies, since the schemas of R and S must then be the same.

Example 16.7: Consider relations $R(a, b)$ and $S(b, c)$ and the expression

$$\sigma_{(a=1 \text{ OR } a=3) \text{ AND } b<c}(R \bowtie S)$$

The condition $b < c$ applies only to to S, and the condition $a = 1$ OR $a = 3$ applies only to R. We thus begin by splitting the AND of the two conditions as we did in the first alternative of Example 16.6:

$$\sigma_{a=1 \text{ OR } a=3}\big(\sigma_{b<c}(R \bowtie S)\big)$$

Next, we can push the selection $\sigma_{b<c}$ to S, giving us the expression:

$$\sigma_{a=1 \text{ OR } a=3}\big(R \bowtie \sigma_{b<c}(S)\big)$$

Finally, push the first condition to R, yielding: $\sigma_{a=1 \text{ OR } a=3}(R) \bowtie \sigma_{b<c}(S)$. □

16.2.3 Pushing Selections

As was illustrated in Example 16.3, pushing a selection down an expression tree — that is, replacing the left side of one of the rules in Section 16.2.2 by its right side — is one of the most powerful tools of the query optimizer. However, when queries involve virtual views, it is sometimes necessary first to move a selection as far *up* the tree as it can go, and then push the selections down all possible branches. An example will illustrate the proper selection-pushing approach.

Example 16.8: Suppose we have the relations

```
StarsIn(title, year, starName)
Movies(title, year, length, genre, studioName, producerC#)
```

Note that we have altered the first two attributes of `StarsIn` from the usual `movieTitle` and `movieYear` to make this example simpler to follow. Define view `MoviesOf1996` by:

Some Trivial Laws

We are not going to state every true law for the relational algebra. The reader should be alert, in particular, for laws about extreme cases: a relation that is empty, a selection or theta-join whose condition is always true or always false, or a projection onto the list of all attributes, for example. A few of the many possible special-case laws:

- Any selection on an empty relation is empty.
- If C is an always-true condition (e.g., $x > 10$ OR $x \leq 10$ on a relation that forbids $x =$ NULL), then $\sigma_C(R) = R$.
- If R is empty, then $R \cup S = S$.

```
CREATE VIEW MoviesOf1996 AS
    SELECT *
    FROM Movies
    WHERE year = 1996;
```

We can ask the query "which stars worked for which studios in 1996?" by the SQL query:

```
SELECT starName, studioName
FROM MoviesOf1996 NATURAL JOIN StarsIn;
```

The view MoviesOf1996 is defined by the relational-algebra expression

$$\sigma_{year=1996}(\texttt{Movies})$$

Thus, the query, which is the natural join of this expression with StarsIn, followed by a projection onto attributes starName and studioName, has the expression shown in Fig. 16.11.

Here, the selection is already as far down the tree as it will go, so there is no way to "push selections down the tree." However, the rule $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$ can be applied "backwards," to bring the selection $\sigma_{year=1996}$ above the join in Fig. 16.11. Then, since *year* is an attribute of both Movies and StarsIn, we may push the selection down to *both* children of the join node. The resulting logical query plan is shown in Fig. 16.12. It is likely to be an improvement, since we reduce the size of the relation StarsIn before we join it with the movies of 1996. □

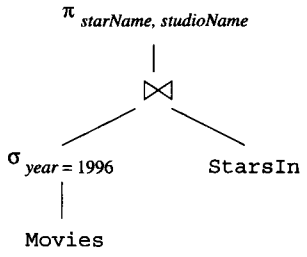

Figure 16.11: Logical query plan constructed from definition of a query and view

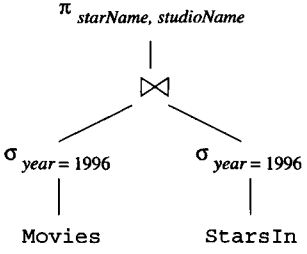

Figure 16.12: Improving the query plan by moving selections up and down the tree

16.2.4 Laws Involving Projection

Projections, like selections, can be "pushed down" through many other operators. Pushing projections differs from pushing selections in that when we push projections, it is quite usual for the projection also to remain where it is. Put another way, "pushing" projections really involves introducing a new projection somewhere below an existing projection.

Pushing projections is useful, but generally less so than pushing selections. The reason is that while selections often reduce the size of a relation by a large factor, projection keeps the number of tuples the same and only reduces the length of tuples. In fact, the extended projection operator of Section 5.2.5 can actually increase the length of tuples.

To describe the transformations of extended projection, we need to introduce some terminology. Consider a term $E \to x$ on the list for a projection, where E is an attribute or an expression involving attributes and constants. We say all attributes mentioned in E are *input* attributes of the projection, and x is an *output* attribute. If a term is a single attribute, then it is both an input and output attribute. If a projection list consists only of attributes, with no renaming or expressions other than a single attribute, then we say the projection is *simple*.

Example 16.9: Projection $\pi_{a,b,c}(R)$ is simple; a, b, and c are both its input

attributes and its output attributes. On the other hand, $\pi_{a+b\to x,\ c}(R)$ is not simple. It has input attributes a, b, and c, and its output attributes are x and c. □

The principle behind laws for projection is that:

- We may introduce a projection anywhere in an expression tree, as long as it eliminates only attributes that are neither used by an operator above nor are in the result of the entire expression.

In the most basic form of these laws, the introduced projections are always simple, although the pre-existing projections, such as L below, need not be.

- $\pi_L(R \bowtie S) = \pi_L\big(\pi_M(R) \bowtie \pi_N(S)\big)$, where M and N are the join attributes and the input attributes if L that are found among the attributes of R and S, respectively.
- $\pi_L(R \bowtie_C S) = \pi_L\big(\pi_M(R) \bowtie_C \pi_N(S)\big)$, where M and N are the join attributes (i.e., those mentioned in condition C) and the input attributes of L that are found among the attributes of R and S respectively.
- $\pi_L(R \times S) = \pi_L\big(\pi_M(R) \times \pi_N(S)\big)$, where M and N are the lists of all attributes of R and S, respectively, that are input attributes of L.

Example 16.10: Let $R(a,b,c)$ and $S(c,d,e)$ be two relations. Consider the expression $\pi_{a+e\to x,\ b\to y}(R \bowtie S)$. The input attributes of the projection are a, b, and e, and c is the only join attribute. We may apply the law for pushing projections below joins to get the equivalent expression:

$$\pi_{a+e\to x,\ b\to y}\big(\pi_{a,b,c}(R) \bowtie \pi_{c,e}(S)\big)$$

Notice that the projection $\pi_{a,b,c}(R)$ is trivial; it projects onto all the attributes of R. We may thus eliminate this projection and get a third equivalent expression: $\pi_{a+e\to x,\ b\to y}\big(R \bowtie \pi_{c,e}(S)\big)$. That is, the only change from the original is that we remove the attribute d from S before the join. □

We can perform a projection entirely before a bag union. That is:

- $\pi_L(R \cup_B S) = \pi_L(R) \cup_B \pi_L(S)$.

On the other hand, projections cannot be pushed below set unions or either the set or bag versions of intersection or difference at all.

Example 16.11: Let $R(a,b)$ consist of the one tuple $\{(1,2)\}$ and $S(a,b)$ consist of the one tuple $\{(1,3)\}$. Then $\pi_a(R \cap S) = \pi_a(\emptyset) = \emptyset$. However, $\pi_a(R) \cap \pi_a(S) = \{(1)\} \cap \{(1)\} = \{(1)\}$. □

If the projection involves some computations, and the input attributes of a term on the projection list belong entirely to one of the arguments of a join or product below the projection, then we have the option, although not the obligation, to perform the computation directly on that argument. An example should help illustrate the point.

Example 16.12: Again let $R(a,b,c)$ and $S(c,d,e)$ be relations, and consider the join and projection $\pi_{a+b\to x,\ d+e\to y}(R \bowtie S)$. We can move the sum $a+b$ and its renaming to x directly onto the relation R, and move the sum $d+e$ to S similarly. The resulting equivalent expression is

$$\pi_{x,y}\big(\pi_{a+b\to x,\ c}(R) \bowtie \pi_{d+e\to y,\ c}(S)\big)$$

One special case to handle is if x or y were c. Then, we could not rename a sum to c, because a relation cannot have two attributes named c. Thus, we would have to invent a temporary name and do another renaming in the projection above the join. For example, $\pi_{a+b\to c,\ d+e\to y}(R \bowtie S)$ could become $\pi_{z\to c,\ y}\big(\pi_{a+b\to z,\ c}(R) \bowtie \pi_{d+e\to y,\ c}(S)\big)$. □

It is also possible to push a projection below a selection.

- $\pi_L\big(\sigma_C(R)\big) = \pi_L\Big(\sigma_C\big(\pi_M(R)\big)\Big)$, where M is the list of all attributes that are either input attributes of L or mentioned in condition C.

As in Example 16.12, we have the option of performing computations on the list L in the list M instead, provided the condition C does not need the input attributes of L that are involved in a computation.

16.2.5 Laws About Joins and Products

We saw in Section 16.2.1 many of the important laws involving joins and products: their commutative and associative laws. However, there are a few additional laws that follow directly from the definition of the join, as was mentioned in Section 2.4.12.

- $R \bowtie_C S = \sigma_C(R \times S)$.
- $R \bowtie S = \pi_L\big(\sigma_C(R \times S)\big)$, where C is the condition that equates each pair of attributes from R and S with the same name, and L is a list that includes one attribute from each equated pair and all the other attributes of R and S.

In practice, we usually want to apply these rules from right to left. That is, we identify a product followed by a selection as a join of some kind. The reason for doing so is that the algorithms for computing joins are generally much faster than algorithms that compute a product followed by a selection on the (very large) result of the product.

16.2.6 Laws Involving Duplicate Elimination

The operator δ, which eliminates duplicates from a bag, can be pushed through many, but not all operators. In general, moving a δ down the tree reduces the size of intermediate relations and may therefore be beneficial. Moreover, we can sometimes move the δ to a position where it can be eliminated altogether, because it is applied to a relation that is known not to possess duplicates:

- $\delta(R) = R$ if R has no duplicates. Important cases of such a relation R include
  a) A stored relation with a declared primary key, and
  b) The result of a γ operation, since grouping creates a relation with no duplicates.
  c) The result of a set union, intersection, or difference.

Several laws that "push" δ through other operators are:

- $\delta(R \times S) = \delta(R) \times \delta(S)$.
- $\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$.
- $\delta(R \bowtie_C S) = \delta(R) \bowtie_C \delta(S)$.
- $\delta\big(\sigma_C(R)\big) = \sigma_C\big(\delta(R)\big)$.

We can also move the δ to either or both of the arguments of an intersection:

- $\delta(R \cap_B S) = \delta(R) \cap_B S = R \cap_B \delta(S) = \delta(R) \cap_B \delta(S)$.

On the other hand, δ generally cannot be pushed through the operators $\cup_B$, $-_B$, or π.

Example 16.13: Let R have two copies of the tuple t and S have one copy of t. Then $\delta(R \cup_B S)$ has one copy of t, while $\delta(R) \cup_B \delta(S)$ has two copies of t. Also, $\delta(R -_B S)$ has one copy of t, while $\delta(R) -_B \delta(S)$ has no copy of t.

Now, consider relation $T(a, b)$ with one copy each of the tuples $(1, 2)$ and $(1, 3)$, and no other tuples. Then $\delta\big(\pi_a(T)\big)$ has one copy of the tuple (1), while $\pi_a\big(\delta(T)\big)$ has two copies of (1). □

16.2.7 Laws Involving Grouping and Aggregation

When we consider the operator γ, we find that the applicability of many transformations depends on the details of the aggregate operators used. Thus, we cannot state laws in the generality that we used for the other operators. One exception is the law, mentioned in Section 16.2.6, that a γ absorbs a δ. Precisely:

- $\delta(\gamma_L(R)) = \gamma_L(R)$.

Another general rule is that we may project useless attributes from the argument should we wish, prior to applying the γ operation. This law can be written:

- $\gamma_L(R) = \gamma_L(\pi_M(R))$ if M is a list containing at least all those attributes of R that are mentioned in L.

The reason that other transformations depend on the aggregation(s) involved in a γ is that some aggregations — `MIN` and `MAX` in particular — are not affected by the presence or absence of duplicates. The other aggregations — `SUM`, `COUNT`, and `AVG` — generally produce different values if duplicates are eliminated prior to application of the aggregation.

Thus, let us call an operator γ_L *duplicate-impervious* if the only aggregations in L are `MIN` and/or `MAX`. Then:

- $\gamma_L(R) = \gamma_L(\delta(R))$ provided γ_L is duplicate-impervious.

Example 16.14: Suppose we have the relations

```
MovieStar(name, addr, gender, birthdate)
StarsIn(movieTitle, movieYear, starName)
```

and we want to know for each year the birthdate of the youngest star to appear in a movie that year. We can express this query as

```
SELECT movieYear, MAX(birthdate)
FROM MovieStar, StarsIn
WHERE name = starName
GROUP BY movieYear;
```

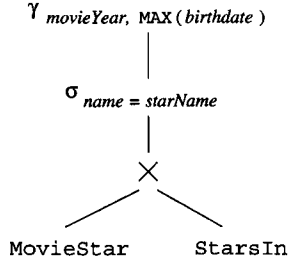

Figure 16.13: Initial logical query plan for the query of Example 16.14

An initial logical query plan constructed directly from the query is shown in Fig. 16.13. The `FROM` list is expressed by a product, and the `WHERE` clause by a selection above it. The grouping and aggregation are expressed by the γ operator above those. Some transformations that we could apply to Fig. 16.13 if we wished are:

1. Combine the selection and product into an equijoin.
2. Generate a δ below the γ, since the γ is duplicate-impervious.
3. Generate a π between the γ and the introduced δ to project onto `movieYear` and `birthdate`, the only attributes relevant to the γ.

The resulting plan is shown in Fig. 16.14.

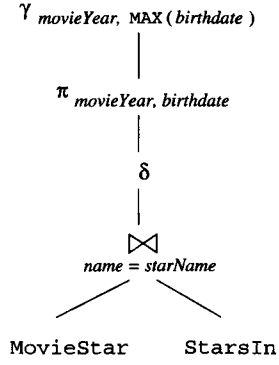

Figure 16.14: Another query plan for the query of Example 16.14

We can now push the δ below the $\bowtie$ and introduce π's below that if we wish. This new query plan is shown in Fig. 16.15. If `name` is a key for `MovieStar`, the δ can be eliminated along the branch leading to that relation. □

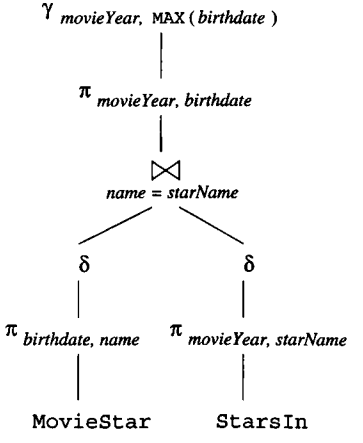

Figure 16.15: A third query plan for Example 16.14

16.2.8 Exercises for Section 16.2

Exercise 16.2.1: When it is possible to push a selection to both arguments of a binary operator, we need to decide whether or not to do so. How would the existence of indexes on one of the arguments affect our choice? Consider, for instance, an expression $\sigma_C(R \cap S)$, where there is an index on S.

Exercise 16.2.2: Give examples to show that:

a) Projection cannot be pushed below set union.

b) Projection cannot be pushed below set or bag difference.

c) Duplicate elimination (δ) cannot be pushed below projection.

d) Duplicate elimination cannot be pushed below bag union or difference.

! **Exercise 16.2.3:** Prove that we can always push a projection below both branches of a bag union.

! **Exercise 16.2.4:** Some laws that hold for sets hold for bags; others do not. For each of the laws below that are true for sets, tell whether or not it is true for bags. Either give a proof the law for bags is true, or give a counterexample.

a) $R \cup R = R$ (the idempotent law for union).

b) $R \cap R = R$ (the idempotent law for intersection).

c) $R - R = \emptyset$.

d) $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$ (distribution of union over intersection).

! **Exercise 16.2.5:** We can define $\subseteq$ for bags by: $R \subseteq S$ if and only if for every element x, the number of times x appears in R is less than or equal to the number of times it appears in S. Tell whether the following statements (which are all true for sets) are true for bags; give either a proof or a counterexample:

a) If $R \subseteq S$, then $R \cup S = S$.

b) If $R \subseteq S$, then $R \cap S = R$.

c) If $R \subseteq S$ and $S \subseteq R$, then $R = S$.

Exercise 16.2.6: Starting with an expression $\pi_L\big(R(a,b,c) \bowtie S(b,c,d,e)\big)$, push the projection down as far as it can go if L is:

a) $b + c \to x,\ c + d \to y$.

b) $a,\ b,\ a + d \to z$.

! **Exercise 16.2.7:** We mentioned in Example 16.14 that none of the plans we showed is necessarily the best plan. Can you think of a better plan?

! **Exercise 16.2.8:** The following are possible equalities involving operations on a relation $R(a,b)$. Tell whether or not they are true; give either a proof or a counterexample.

a) $\gamma_{MIN(a)\to y,\ x}\big(\gamma_{a,\ SUM(b)\to x}(R)\big) = \gamma_{y,SUM(b)\to x}\big(\gamma_{MIN(a)\to y,\ b}(R)\big)$.

b) $\gamma_{MIN(a)\to y,\ x}\big(\gamma_{a,\ MAX(b)\to x}(R)\big) = \gamma_{y,MAX(b)\to x}\big(\gamma_{MIN(a)\to y,\ b}(R)\big)$.

!! **Exercise 16.2.9:** The join-like operators of Exercise 15.2.4 obey some of the familiar laws, and others do not. Tell whether each of the following is or is not true. Give either a proof that the law holds or a counterexample.

a) $\sigma_C(R \ltimes S) = \sigma_C(R) \ltimes S$.

b) $\sigma_C(R \overset{\circ}{\bowtie} S) = \sigma_C(R) \overset{\circ}{\bowtie} S$.

c) $\sigma_C(R \overset{\circ}{\bowtie}_L S) = \sigma_C(R) \overset{\circ}{\bowtie}_L S$, where C involves only attributes of R.

d) $\sigma_C(R \overset{\circ}{\bowtie}_L S) = R \overset{\circ}{\bowtie}_L \sigma_C(S)$, where C involves only attributes of S.

e) $\pi_L(R \,\overline{\ltimes}\, S) = \pi_L(R) \,\overline{\ltimes}\, S$.

f) $(R \overset{\circ}{\bowtie} S) \overset{\circ}{\bowtie} T = R \overset{\circ}{\bowtie} (S \overset{\circ}{\bowtie} T)$.

g) $R \overset{\circ}{\bowtie} S = S \overset{\circ}{\bowtie} R$.

h) $R \overset{\circ}{\bowtie}_L S = S \overset{\circ}{\bowtie}_L R$.

i) $R \ltimes S = S \ltimes R$.

!! **Exercise 16.2.10:** While it is not precisely an algebraic law, because it involves an indeterminate number of operands, it is generally true that

$$\texttt{SUM}(a_1, a_2, \ldots, a_n) = a_1 + a_2 + \cdots + a_n$$

SQL has both a `SUM` operator and addition for integers and reals. Considering the possibility that one or more of the a_i's could be `NULL`, rather than an integer or real, does this "law" hold in SQL?

16.3 From Parse Trees to Logical Query Plans

We now resume our discussion of the query compiler. Having constructed a parse tree for a query in Section 16.1, we next need to turn the parse tree into the preferred logical query plan. There are two steps, as was suggested in Fig. 16.1.

The first step is to replace the nodes and structures of the parse tree, in appropriate groups, by an operator or operators of relational algebra. We shall suggest some of these rules and leave some others for exercises. The second step is to take the relational-algebra expression produced by the first step and to turn it into an expression that we expect can be converted to the most efficient physical query plan.

16.3.1 Conversion to Relational Algebra

We shall now describe informally some rules for transforming SQL parse trees to algebraic logical query plans. The first rule, perhaps the most important, allows us to convert all "simple" select-from-where constructs to relational algebra directly. Its informal statement:

- If we have a <Query> with a <Condition> that has no subqueries, then we may replace the entire construct — the select-list, from-list, and condition — by a relational-algebra expression consisting, from bottom to top, of:

  1. The product of all the relations mentioned in the <FromList>, which is the argument of:
  2. A selection σ_C, where C is the <Condition> expression in the construct being replaced, which in turn is the argument of:
  3. A projection π_L, where L is the list of attributes in the <SelList>.

Example 16.15: Let us consider the parse tree of Fig. 16.5. The select-from-where transformation applies to the entire tree of Fig. 16.5. We take the product of the two relations `StarsIn` and `MovieStar` of the from-list, select for the condition in the subtree rooted at <Condition>, and project onto the select-list, `movieTitle`. The resulting relational-algebra expression is Fig. 16.16.

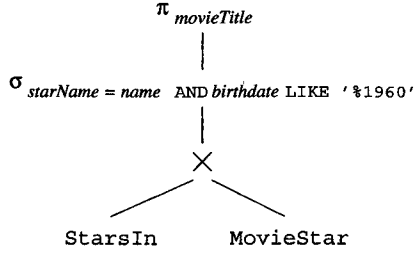

Figure 16.16: Translation of a parse tree to an algebraic expression tree

The same transformation does not apply to the outer query of Fig. 16.3. The reason is that the condition involves a subquery, a matter we defer to Section 16.3.2. However, we can apply the transformation to the subquery in

Limitations on Selection Conditions

One might wonder why we do not allow C, in a selection operator σ_C, to involve a subquery. It is conventional in relational algebra for the *arguments* of an operator — the elements that do not appear in subscripts — to be expressions that yield relations. On the other hand, *parameters* — the elements that appear in subscripts — have a type other than relations. For instance, parameter C in σ_C is a boolean-valued condition, and parameter L in π_L is a list of attributes or formulas.

If we follow this convention, then whatever calculation is implied by a parameter can be applied to each tuple of the relation argument(s). That limitation on the use of parameters simplifies query optimization. Suppose, in contrast, that we allowed an operator like $\sigma_C(R)$, where C involves a subquery. Then the application of C to each tuple of R involves computing the subquery. Do we compute it anew for every tuple of R? That would be unnecessarily expensive, unless the subquery were *correlated*, i.e., its value depends on something defined outside the query, as the subquery of Fig. 16.3 depends on the value of `starName`. Even correlated subqueries can be evaluated without recomputation for each tuple, in most cases, provided we organize the computation correctly.

Fig. 16.3. The expression of relational algebra that we get from the subquery is $\pi_{name}(\sigma_{birthdate \texttt{ LIKE '\%1960'}}(\texttt{MovieStar}))$. □

16.3.2 Removing Subqueries From Conditions

For parse trees with a <Condition> that has a subquery, we shall introduce an intermediate form of operator, between the syntactic categories of the parse tree and the relational-algebra operators that apply to relations. This operator is often called *two-argument selection*. We shall represent a two-argument selection in a transformed parse tree by a node labeled σ, with no parameter. Below this node is a left child that represents the relation R upon which the selection is being performed, and a right child that is an expression for the condition applied to each tuple of R. Both arguments may be represented as parse trees, as expression trees, or as a mixture of the two.

Example 16.16: In Fig. 16.17 is a rewriting of the parse tree of Fig. 16.3 that uses a two-argument selection. Several transformations have been made to construct Fig. 16.17 from Fig. 16.3:

1. The subquery in Fig. 16.3 has been replaced by an expression of relational algebra, as discussed at the end of Example 16.15.

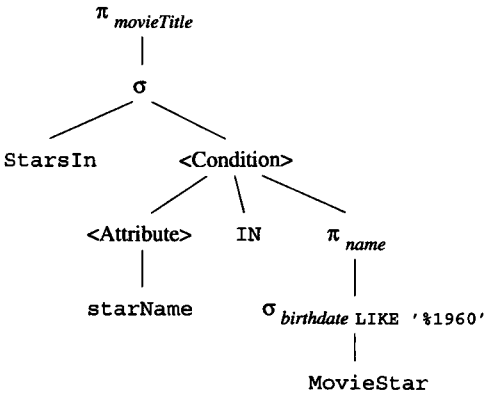

Figure 16.17: An expression using a two-argument σ, midway between a parse tree and relational algebra

2. The outer query has also been replaced, using the rule for select-from-where expressions from Section 16.3.1. However, we have expressed the necessary selection as a two-argument selection, rather than by the conventional σ operator of relational algebra. As a result, the upper node of the parse tree labeled <Condition> has not been replaced, but remains as an argument of the selection, with its parentheses and <Query> replaced by relational algebra, per point (1).

This tree needs further transformation, which we discuss next. □

We need rules that allow us to replace a two-argument selection by a one-argument selection and other operators of relational algebra. Each form of condition may require its own rule. In common situations, it is possible to remove the two-argument selection and reach an expression that is pure relational algebra. However, in extreme cases, the two-argument selection can be left in place and considered part of the logical query plan.

We shall give, as an example, the rule that lets us deal with the condition in Fig. 16.17 involving the `IN` operator. Note that the subquery in this condition is uncorrelated; that is, the subquery's relation can be computed once and for all, independent of the tuple being tested. The rule for eliminating such a condition is stated informally as follows:

- Suppose we have a two-argument selection in which the first argument represents some relation R and the second argument is a <Condition> of the form t `IN` S, where expression S is an uncorrelated subquery, and t is a tuple composed of (some) attributes of R. We transform the tree as follows:

  a) Replace the <Condition> by the tree that is the expression for S. If S may have duplicates, then it is necessary to include a δ operation

at the root of the expression for S, so the expression being formed does not produce more copies of tuples than the original query does.

b) Replace the two-argument selection by a one-argument selection σ_C, where C is the condition that equates each component of the tuple t to the corresponding attribute of the relation S.

c) Give σ_C an argument that is the product of R and S.

Figure 16.18 illustrates this transformation.

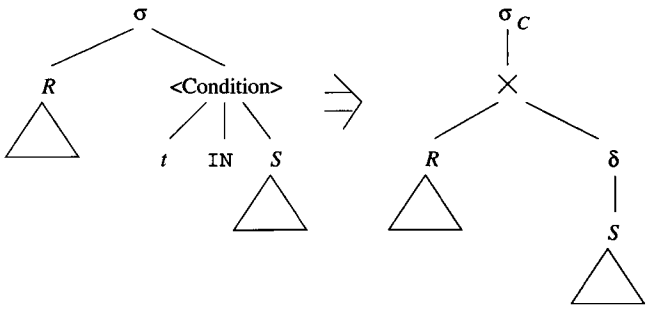

Figure 16.18: This rule handles a two-argument selection with a condition involving IN

Example 16.17: Consider the tree of Fig. 16.17, to which we shall apply the rule for IN conditions described above. In this figure, relation R is `StarsIn`, and relation S is the result of the relational-algebra expression consisting of the subtree rooted at π_{name}. The tuple t has one component, the attribute `starName`.

The two-argument selection is replaced by $\sigma_{starName=name}$; its condition C equates the one component of tuple t to the attribute of the result of query S. The child of the σ node is a $\times$ node, and the arguments of the $\times$ node are the node labeled `StarsIn` and the root of the expression for S. Notice that, because `name` is the key for `MovieStar`, there is no need to introduce a duplicate-eliminating δ in the expression for S. The new expression is shown in Fig. 16.19. It is completely in relational algebra, and is equivalent to the expression of Fig. 16.16, although its structure is quite different. □

The strategy for translating subqueries to relational algebra is more complex when the subquery is correlated. Since correlated subqueries involve unknown values defined outside themselves, they cannot be translated in isolation. Rather, we need to translate the subquery so that it produces a relation in which certain extra attributes appear — the attributes that must later be compared with the externally defined attributes. The conditions that relate attributes from the subquery to attributes outside are then applied to this relation, and

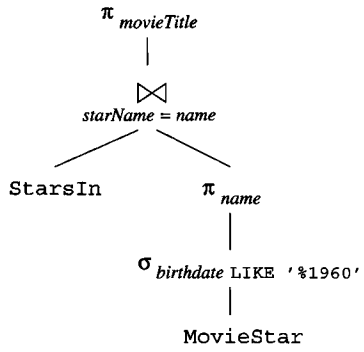

Figure 16.19: Applying the rule for `IN` conditions

the extra attributes that are no longer necessary can then be projected out. During this process, we must avoid introducing duplicate tuples, if the query does not eliminate duplicates at the end. The following example illustrates this technique.

```
SELECT DISTINCT m1.movieTitle, m1.movieYear
FROM StarsIn m1
WHERE m1.movieYear - 40 <= (
    SELECT AVG(birthdate)
    FROM StarsIn m2, MovieStar s
    WHERE m2.starName = s.name AND
        m1.movieTitle = m2.movieTitle AND
        m1.movieYear = m2.movieYear
);
```

Figure 16.20: Finding movies with high average star age

Example 16.18: Figure 16.20 is a SQL rendition of the query: "find the movies where the average age of the stars was at most 40 when the movie was made." To simplify, we treat `birthdate` as a birth year, so we can take its average and get a value that can be compared with the `movieYear` attribute of `StarsIn`. We have also written the query so that each of the three references to relations has its own tuple variable, in order to help remind us where the various attributes come from.

Fig. 16.21 shows the result of parsing the query and performing a partial translation to relational algebra. During this initial translation, we split the `WHERE`-clause of the subquery in two, and used part of it to convert the product of relations to an equijoin. We have retained the aliases `m1`, `m2`, and `s` in the nodes of this tree, in order to make clearer the origin of each attribute.

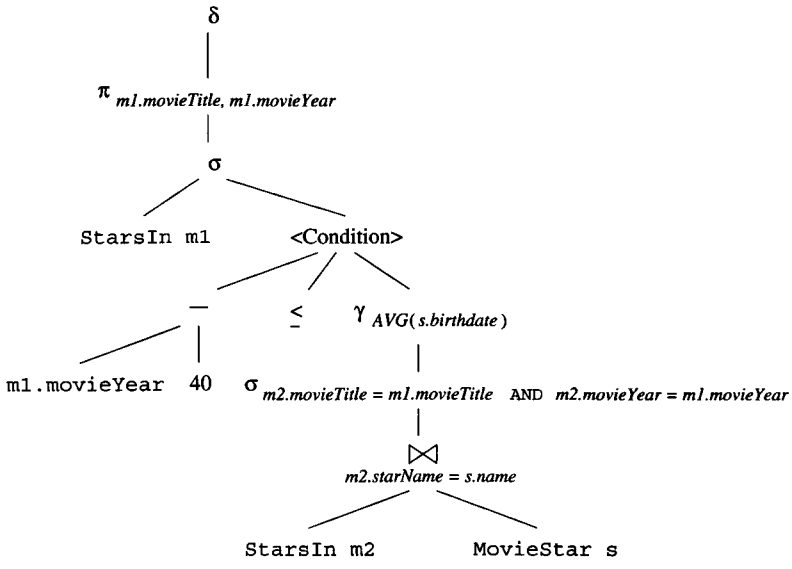

Figure 16.21: Partially transformed parse tree for Fig. 16.20

Alternatively, we could have used projections to rename attributes and thus avoid conflicting attribute names, but the result would be harder to follow.

In order to remove the <Condition> node and eliminate the two-argument σ, we need to create an expression that describes the relation in the right branch of the <Condition>. However, because the subquery is correlated, there is no way to obtain the attributes `m1.movieTitle` or `m1.movieYear` from the relations mentioned in the subquery, which are `StarsIn` (with alias `m2`) and `MovieStar`. Thus, we need to defer the selection

$$\sigma_{m2.movieTitle=m1.movieTitle \text{ AND } m2.movieYear=m1.movieYear}$$

until after the relation from the subquery is combined with the copy of `StarsIn` from the outer query (the copy aliased `m1`). To transform the logical query plan in this way, we need to modify the γ to group by the attributes `m2.movieTitle` and `m2.movieYear`, so these attributes will be available when needed by the selection. The net effect is that we compute for the subquery a relation consisting of movies, each represented by its title and year, and the average star birth year for that movie.

The modified group-by operator appears in Fig. 16.22; in addition to the two grouping attributes, we need to rename the average `abd` (average birthdate) so we can refer to it later. Figure 16.22 also shows the complete translation to relational algebra. Above the γ, the `StarsIn` from the outer query is joined with the result of the subquery. The selection from the subquery is then applied to the product of `StarsIn` and the result of the subquery; we show this selection as

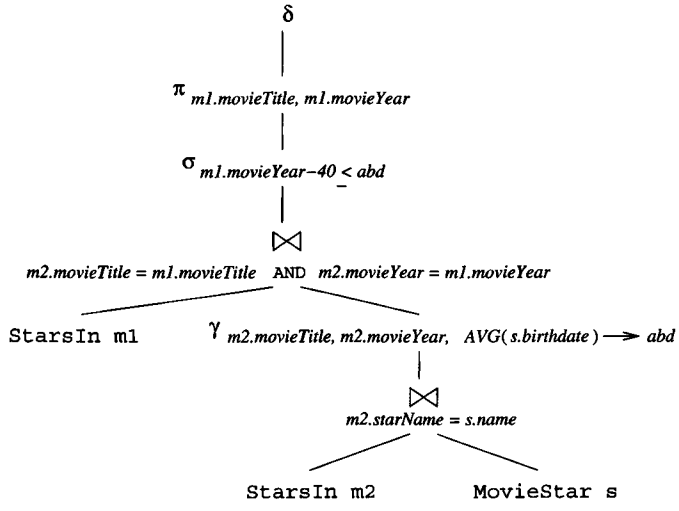

Figure 16.22: Translation of Fig. 16.21 to a logical query plan

a theta-join, which it would become after normal application of algebraic laws. Above the theta-join is another selection, this one corresponding to the selection of the outer query, in which we compare the movie's year to the average birth year of its stars. The algebraic expression finishes at the top like the expression of Fig. 16.21, with the projection onto the desired attributes and the elimination of duplicates.

As we shall see in Section 16.3.3, there is much more that a query optimizer can do to improve the query plan. This particular example satisfies three conditions that let us improve the plan considerably. The conditions are:

1. Duplicates are eliminated at the end,
2. Star names from `StarsIn m1` are projected out, and
3. The join between `StarsIn m1` and the rest of the expression equates the title and year attributes from `StarsIn m1` and `StarsIn m2`.

Because these conditions hold, we can replace all uses of `m1.movieTitle` and `m1.movieYear` by `m2.movieTitle` and `m2.movieYear`, respectively. Thus, the upper join in Fig. 16.22 is unnecessary, as is the argument `StarsIn m1`. This logical query plan is shown in Fig. 16.23. □

16.3.3 Improving the Logical Query Plan

When we convert our query to relational algebra we obtain one possible logical query plan. The next step is to rewrite the plan using the algebraic laws outlined

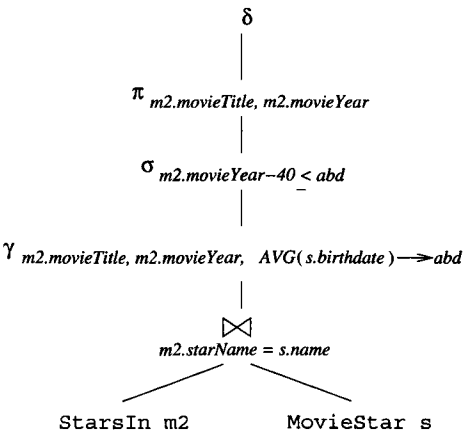

Figure 16.23: Simplification of Fig. 16.22

in Section 16.2. Alternatively, we could generate more than one logical plan, representing different orders or combinations of operators. But in this book we shall assume that the query rewriter chooses a single logical query plan that it believes is "best," meaning that it is likely to result ultimately in the cheapest physical plan.

We do, however, leave open the matter of what is known as "join ordering," so a logical query plan that involves joining relations can be thought of as a family of plans, corresponding to the different ways a join could be ordered and grouped. We discuss choosing a join order in Section 16.6. Similarly, a query plan involving three or more relations that are arguments to the other associative and commutative operators, such as union, should be assumed to allow reordering and regrouping as we convert the logical plan to a physical plan. We begin discussing the issues regarding ordering and physical plan selection in Section 16.4.

There are a number of algebraic laws from Section 16.2 that tend to improve logical query plans. The following are most commonly used in optimizers:

- Selections can be pushed down the expression tree as far as they can go. If a selection condition is the AND of several conditions, then we can split the condition and push each piece down the tree separately. This strategy is probably the most effective improvement technique, but we should recall the discussion in Section 16.2.3, where we saw that in some circumstances it was necessary to push the selection up the tree first.

- Similarly, projections can be pushed down the tree, or new projections can be added. As with selections, the pushing of projections should be done with care, as discussed in Section 16.2.4.

- Duplicate eliminations can sometimes be removed, or moved to a more

convenient position in the tree, as discussed in Section 16.2.6.

- Certain selections can be combined with a product below to turn the pair of operations into an equijoin, which is generally much more efficient to evaluate than are the two operations separately. We discussed these laws in Section 16.2.5.

Example 16.19: Let us consider the query of Fig. 16.16. First, we may split the two parts of the selection into $\sigma_{starName=name}$ and $\sigma_{birthdate\ \texttt{LIKE}\ \texttt{'\%1960'}}$. The latter can be pushed down the tree, since the only attribute involved, `birthdate`, is from the relation `MovieStar`. The first condition involves attributes from both sides of the product, but they are equated, so the product and selection is really an equijoin. The effect of these transformations is shown in Fig. 16.24. □

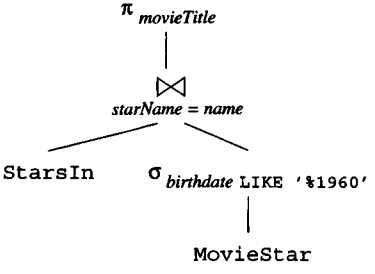

Figure 16.24: The effect of query rewriting

16.3.4 Grouping Associative/Commutative Operators

An operator that is associative and commutative operators may be thought of as having any number of operands. Thinking of an operator such as join as having any number of operands lets us reorder those operands so that when the multiway join is executed as a sequence of binary joins, they take less time than if we had executed the joins in the order implied by the parse tree. We discuss ordering multiway joins in Section 16.6.

Thus, we shall perform a last step before producing the final logical query plan: for each portion of the subtree that consists of nodes with the same associative and commutative operator, we group the nodes with these operators into a single node with many children. Recall that the usual associative/commutative operators are natural join, union, and intersection. Natural joins and theta-joins can also be combined with each other under certain circumstances:

1. We must replace the natural joins with theta-joins that equate the attributes of the same name.

2. We must add a projection to eliminate duplicate copies of attributes involved in a natural join that has become a theta-join.

3. The theta-join conditions must be associative. Recall there are cases, as discussed in Section 16.2.1, where theta-joins are not associative.

In addition, products can be considered as a special case of natural join and combined with joins if they are adjacent in the tree. Figure 16.25 illustrates this transformation in a situation where the logical query plan has a cluster of two union operators and a cluster of three natural join operators. Note that the letters R through W stand for any expressions, not necessarily for stored relations.

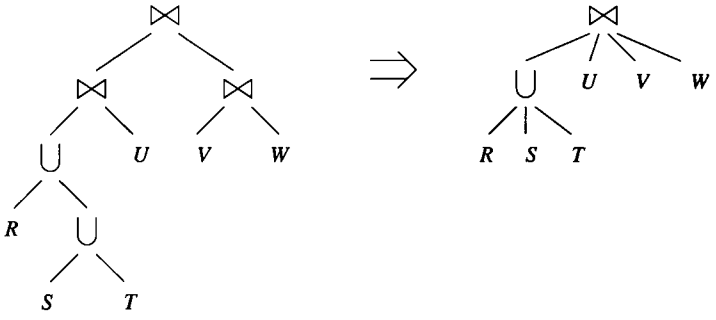

Figure 16.25: Final step in producing the logical query plan: group the associative and commutative operators

16.3.5 Exercises for Section 16.3

Exercise 16.3.1: Replace the natural joins in the following expressions by equivalent theta-joins and projections. Tell whether the resulting theta-joins form a commutative and associative group.

a) $\big(R(a,b) \bowtie S(b,c)\big) \bowtie_{S.c>T.c} T(c,d)$.

b) $\big(R(a,b) \bowtie S(b,c)\big) \bowtie \big(T(c,d) \bowtie U(d,e)\big)$.

c) $\big(R(a,b) \bowtie S(b,c)\big) \bowtie \big(T(c,d) \bowtie U(a,d)\big)$.

Exercise 16.3.2: Convert to relational algebra your parse trees from Exercise 16.1.3(a) and (b). For (b), show both the form with a two-argument selection and its eventual conversion to a one-argument (conventional σ_C) selection.

! **Exercise 16.3.3:** Give a rule for converting each of the following forms of <Condition> to relational algebra. All conditions may be assumed to be applied (by a two-argument selection) to a relation R. You may assume that the

subquery is not correlated with R. Be careful that you do not introduce or eliminate duplicates in opposition to the formal definition of SQL.

a) A condition of the form `EXISTS(<Query>)`.

b) A condition of the form a = `ANY <Query>`, where a is an attribute of R.

c) A condition of the form a = `ALL <Query>`, where a is an attribute of R.

!! **Exercise 16.3.4:** Repeat Exercise 16.3.3, but allow the subquery to be correlated with R. For simplicity, you may assume that the subquery has the simple form of select-from-where expression described in this section, with *no* further subqueries.

!! **Exercise 16.3.5:** From how many different expression trees could the grouped tree on the right of Fig. 16.25 have come? Remember that the order of children after grouping is not necessarily reflective of the ordering in the original expression tree.

16.4 Estimating the Cost of Operations

Having parsed a query and transformed it into a logical query plan, we must next turn the logical plan into a physical plan. We normally do so by considering many different physical plans that are derived from the logical plan, and evaluating or estimating the cost of each. After this evaluation, often called *cost-based enumeration*, we pick the physical query plan with the least estimated cost; that plan is the one passed to the query-execution engine. When enumerating possible physical plans derivable from a given logical plan, we select for each physical plan:

1. An order and grouping for associative-and-commutative operations like joins, unions, and intersections.

2. An algorithm for each operator in the logical plan, for instance, deciding whether a nested-loop join or a hash-join should be used.

3. Additional operators — scanning, sorting, and so on — that are needed for the physical plan but that were not present explicitly in the logical plan.

4. The way in which arguments are passed from one operator to the next, for instance, by storing the intermediate result on disk or by using iterators and passing an argument one tuple or one main-memory buffer at a time.

To make each of these choices, we need to understand what the costs of the various physical plans are. We cannot know these costs exactly without executing the plan. But almost always, the cost of executing a query plan is

Review of Notation

Recall from Section 15.1.3 the following size parameters:

- $B(R)$ is the number of blocks needed to hold relation R.
- $T(R)$ is the number of tuples of relation R.
- $V(R, a)$ is the *value count* for attribute a of relation R, that is, the number of distinct values relation R has in attribute a. Also, $V(R, [a_1, a_2, \ldots, a_n])$ is the number of distinct values R has when all of attributes $a_1, a_2, \ldots, a_n$ are considered together, that is, the number of tuples in $\delta(\pi_{a_1, a_2, \ldots, a_n}(R))$.

significantly greater than all the work done by the query compiler in selecting a plan. Thus, we do not want to execute more than one plan for one query, and we are forced to estimate the cost of any plan without executing it.

Therefore, our first problem is how to estimate costs of plans accurately. Such estimates are based on parameters of the data (see the box on "Review of Notation") that must be either computed exactly from the data or estimated by a process of "statistics gathering" that we discuss in Section 16.5.1. Given values for these parameters, we may make a number of reasonable estimates of relation sizes that can be used to predict the cost of a complete physical plan.

16.4.1 Estimating Sizes of Intermediate Relations

The physical plan is selected to minimize the estimated cost of evaluating the query. No matter what method is used for executing query plans, and no matter how costs of query plans are estimated, the sizes of intermediate relations of the plan have a profound influence on costs. Ideally, we want rules for estimating the number of tuples in an intermediate relation so that the rules:

1. Give accurate estimates.
2. Are easy to compute.
3. Are logically consistent; that is, the size estimate for an intermediate relation should not depend on how that relation is computed. For instance, the size estimate for a join of several relations should not depend on the order in which we join the relations.

There is no universally agreed-upon way to meet these three conditions. We shall give some simple rules that serve in most situations. Fortunately, the goal of size estimation is not to predict the exact size; it is to help select a physical

query plan. Even an inaccurate size-estimation method will serve that purpose well if it errs consistently, that is, if the size estimator assigns the least cost to the best physical query plan, even if the actual cost of that plan turns out to be different from what was predicted.

16.4.2 Estimating the Size of a Projection

The extended projection of Section 5.2.5 is a bag projection and does not eliminate duplicates. We shall treat a clasical, duplicate-eliminating projection as a bag-projection followed by a δ. The extended projection of bags is different from the other operators, in that the size of the result is computable exactly. Normally, tuples shrink during a projection, as some components are eliminated. However, the extended projection allows the creation of new components that are combinations of attributes, and so there are situations where a π operator actually increases the size of the relation.

Example 16.20: Suppose $R(a, b, c)$ is a relation, where a and b are integers of four bytes each, and c is a string of 100 bytes. Let tuple headers require 12 bytes. Then each tuple of R requires 120 bytes. Let blocks be 1024 bytes long, with block headers of 24 bytes. We can thus fit 8 tuples in one block. Suppose $T(R) = 10{,}000$; i.e., there are 10,000 tuples in R. Then $B(R) = 1250$.

Consider $S = \pi_{a+b\to x,c}(R)$; that is, we replace a and b by their sum. Tuples of S require 116 bytes: 12 for header, 4 for the sum, and 100 for the string. Although tuples of S are slightly smaller than tuples of R, we can still fit only 8 tuples in a block. Thus, $T(S) = 10{,}000$ and $B(S) = 1250$.

Now consider $U = \pi_{a,b}(R)$, where we eliminate the string component. Tuples of U are only 20 bytes long. $T(U)$ is still 10,000. However, we can now pack 50 tuples of U into one block, so $B(U) = 200$. This projection thus shrinks the relation by a factor slightly more than 6. □

16.4.3 Estimating the Size of a Selection

When we perform a selection, we generally reduce the number of tuples, although the sizes of tuples remain the same. In the simplest kind of selection, where an attribute is equated to a constant, there is an easy way to estimate the size of the result, provided we know, or can estimate, the number of different values the attribute has. Let $S = \sigma_{A=c}(R)$, where A is an attribute of R and c is a constant. Then we recommend as an estimate:

- $T(S) = T(R)/V(R, A)$

This rule surely holds if the value of A is chosen randomly from among all the possible values.

The size estimate is more problematic when the selection involves an inequality comparison, for instance, $S = \sigma_{a<10}(R)$. One might think that on the average, half the tuples would satisfy the comparison and half not, so $T(R)/2$

The Zipfian Distribution

In estimating the size of a selection $\sigma_{A=c}$ it is not necessary to assume that values of A appear equally often. In fact, many attributes have values whose occurrences follow a *Zipfian distribution*, where the frequencies of the ith most common values are in proportion to $1/\sqrt{i}$. For example, if the most common value appears 1000 times, then the second most common value would be expected to appear about $1000/\sqrt{2}$ times, or 707 times, and the third most common value would appear about $1000/\sqrt{3}$ times, or 577 times. Originally postulated as a way to describe the relative frequencies of words in English sentences, this distribution has been found to appear in many sorts of data. For example, in the US, state populations follow an approximate Zipfian distribution. The three most populous states, California, Texas, and New York, have populations in ratio approximately 1:0.62:0.56, compared with the Zipfian ideal of 1:0.71:0.58. Thus, if `state` were an attribute of a relation describing US people, say a list of magazine subscribers, we would expect the values of `state` to distribute in the Zipfian, rather than uniform manner.

As long as the constant in the selection condition is chosen randomly, it doesn't matter whether the values of the attribute involved have a uniform, Zipfian, or other distribution; the *average* size of the matching set will still be $T(R)/V(R,a)$. However, if the constants are also chosen with a Zipfian distribution, then we would expect the average size of the selected set to be somewhat larger than $T(R)/V(R,a)$.

would estimate the size of S. However, there is an intuition that queries involving an inequality tend to retrieve a small fraction of the possible tuples.[3] Thus, we propose a rule that acknowledges this tendency, and assumes the typical inequality will return about one third of the tuples, rather than half the tuples. If $S = \sigma_{a<c}(R)$, then our estimate for $T(S)$ is:

- $T(S) = T(R)/3$

The case of a "not equals" comparison is rare. However, should we encounter a selection like $S = \sigma_{a\neq 10}(R)$, we recommend assuming that essentially all tuples will satisfy the condition. That is, take $T(S) = T(R)$ as an estimate. Alternatively, we may use $T(S) = T(R)\big(V(R,a)-1\big)/V(R,a)$, which is slightly less, as an estimate, acknowledging that about fraction $1/V(R,a)$ tuples of R will fail to meet the condition because their a-value *does* equal the constant.

When the selection condition C is the `AND` of several equalities and inequalities, we can treat the selection $\sigma_C(R)$ as a cascade of simple selections, each of

[3]For instance, if you had data about faculty salaries, would you be more likely to query for those faculty who made *less* than $200,000 or *more* than $200,000?

which checks for one of the conditions. Note that the order in which we place these selections doesn't matter. The effect will be that the size estimate for the result is the size of the original relation multiplied by the *selectivity* factor for each condition. That factor is 1/3 for any inequality, 1 for $\neq$, and $1/V(R,A)$ for any attribute A that is compared to a constant in the condition C.

Example 16.21: Let $R(a,b,c)$ be a relation, and $S = \sigma_{a=10 \text{ AND } b<20}(R)$. Also, let $T(R) = 10{,}000$, and $V(R,a) = 50$. Then our best estimate of $T(S)$ is $T(R)/(50 \times 3)$, or 67. That is, 1/50th of the tuples of R will survive the $a = 10$ filter, and 1/3 of those will survive the $b < 20$ filter.

An interesting special case where our analysis breaks down is when the condition is contradictory. For instance, consider $S = \sigma_{a=10 \text{ AND } a>20}(R)$. According to our rule, $T(S) = T(R)/3V(R,a)$, or 67 tuples. However, it should be clear that no tuple can have both $a = 10$ and $a > 20$, so the correct answer is $T(S) = 0$. When rewriting the logical query plan, the query optimizer can look for instances of many special-case rules. In the above instance, the optimizer can apply a rule that finds the selection condition logically equivalent to FALSE and replaces the expression for S by the empty set. □

When a selection involves an OR of conditions, say $S = \sigma_{C_1 \text{ OR } C_2}(R)$, then we have less certainty about the size of the result. One simple assumption is that no tuple will satisfy both conditions, so the size of the result is the sum of the number of tuples that satisfy each. That measure is generally an overestimate, and in fact can sometimes lead us to the absurd conclusion that there are more tuples in S than in the original relation R.

A less simple, but possibly more accurate estimate of the size of

$$S = \sigma_{C_1 \text{ OR } C_2}(R)$$

is to assume that C_1 and C_2 are independent. Then, if R has n tuples, m_1 of which satisfy C_1 and m_2 of which satisfy C_2, we would estimate the number of tuples in S as $n\big(1-(1-m_1/n)(1-m_2/n)\big)$. In explanation, $1 - m_1/n$ is the fraction of tuples that do not satisfy C_1, and $1 - m_2/n$ is the fraction that do not satisfy C_2. The product of these numbers is the fraction of R's tuples that are *not* in S, and 1 minus this product is the fraction that are in S.

Example 16.22: Suppose $R(a,b)$ has $T(R) = 10{,}000$ tuples, and

$$S = \sigma_{a=10 \text{ OR } b<20}(R)$$

Let $V(R,a) = 50$. Then the number of tuples that satisfy $a = 10$ we estimate at 200, i.e., $T(R)/V(R,a)$. The number of tuples that satisfy $b < 20$ we estimate at $T(R)/3$, or 3333.

The simplest estimate for the size of S is the sum of these numbers, or 3533. The more complex estimate based on independence of the conditions $a = 10$ and $b < 20$ gives $10000\big(1-(1-200/10000)(1-3333/10000)\big)$, or 3466. In this case, there is little difference between the two estimates, and it is very unlikely

that choosing one over the other would change our estimate of the best physical query plan. □

The final operator that could appear in a selection condition is NOT. The estimated number of tuples of R that satisfy condition NOT C is $T(R)$ minus the estimated number that satisfy C.

16.4.4 Estimating the Size of a Join

We shall consider here only the natural join. Other joins can be handled according to the following outline:

1. The number of tuples in the result of an equijoin can be computed exactly as for a natural join, after accounting for the change in variable names. Example 16.24 will illustrate this point.

2. Other theta-joins can be estimated as if they were a selection following a product. Note that the number of tuples in a product is the product of the number of tuples in the relations involved.

We shall begin our study with the assumption that the natural join of two relations involves only the equality of two attributes. That is, we study the join $R(X,Y) \bowtie S(Y,Z)$, but initially we assume that Y is a single attribute although X and Z can represent any set of attributes.

The problem is that we don't know how the Y-values in R and S relate. For instance:

1. The two relations could have disjoint sets of Y-values, in which case the join is empty and $T(R \bowtie S) = 0$.

2. Y might be the key of S and the corresponding foreign key of R, so each tuple of R joins with exactly one tuple of S, and $T(R \bowtie S) = T(R)$.

3. Almost all the tuples of R and S could have the same Y-value, in which case $T(R \bowtie S)$ is about $T(R)T(S)$.

To focus on the most common situations, we shall make two simplifying assumptions:

- *Containment of Value Sets.* If Y is an attribute appearing in several relations, then each relation chooses its values from the front of a fixed list of values $y_1, y_2, y_3, \ldots$ and has all the values in that prefix. As a consequence, if R and S are two relations with an attribute Y, and $V(R,Y) \leq V(S,Y)$, then every Y-value of R will be a Y-value of S.

- *Preservation of Value Sets.* If we join a relation R with another relation, then an attribute A that is not a join attribute (i.e., not present in both relations) does not lose values from its set of possible values. More precisely, if A is an attribute of R but not of S, then $V(R \bowtie S, A) = V(R, A)$.

Assumption (1), containment of value sets, clearly might be violated, but it *is* satisfied when Y is a key in S and the corresponding foreign key in R. It also is approximately true in many other cases, since we would intuitively expect that if S has many Y-values, then a given Y-value that appears in R has a good chance of appearing in S.

Assumption (2), preservation of value sets, also might be violated, but it is true when the join attribute(s) of $R \bowtie S$ are a key for S and the corresponding foreign key of R. In fact, (2) can only be violated when there are "dangling tuples" in R, that is, tuples of R that join with no tuple of S; and even if there *are* dangling tuples in R, the assumption might still hold.

Under these assumptions, we can estimate the size of $R(X,Y) \bowtie S(Y,Z)$ as follows. Suppose r is a tuple in R, and S is a tuple in S. What is the probability that r and s agree on attribute Y? Suppose that $V(R,Y) \geq V(S,Y)$. Then the Y-value of s is surely one of the Y values that appear in R, by the containment-of-value-sets assumption. Hence, the chance that r has the same Y-value as s is $1/V(R,Y)$. Similarly, if $V(R,Y) < V(S,Y)$, then the value of Y in r will appear in S, and the probability is $1/V(S,Y)$ that r and s will share the same Y-value. In general, we see that the probability of agreement on the Y value is $1/\max\big(V(R,Y),V(S,Y)\big)$. Thus:

- $T(R \bowtie S) = T(R)T(S)/\max\big(V(R,Y),V(S,Y)\big)$

That is, the estimated number of tuples in $T(R \bowtie S)$ is the number of pairs of tuples — one from R and one from S, times the probability that such a pair shares a common Y value.

Example 16.23: Let us consider the following three relations and their important statistics:

$R(a,b)$	$S(b,c)$	$U(c,d)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R,b) = 20$	$V(S,b) = 50$	
	$V(S,c) = 100$	$V(U,c) = 500$

Suppose we want to compute the natural join $R \bowtie S \bowtie U$. One way is to group R and S first, as $(R \bowtie S) \bowtie U$. Our estimate for $T(R \bowtie S)$ is $T(R)T(S)/\max\big(V(R,b),V(S,b)\big)$, which is $1000 \times 2000/50$, or 40,000.

We then need to join $R \bowtie S$ with U. Our estimate for the size of the result is $T(R \bowtie S)T(U)/\max\big(V(R \bowtie S,c),V(U,c)\big)$. By our assumption that value sets are preserved, $V(R \bowtie S,c)$ is the same as $V(S,c)$, or 100; that is no values of attribute c disappeared when we performed the join. In that case, we get as our estimate for the number of tuples in $R \bowtie S \bowtie U$ the value $40{,}000 \times 5000/\max(100,500)$, or 400,000.

We could also start by joining S and U. If we do, then we get the estimate $T(S \bowtie U) = T(S)T(U)/\max\big(V(S,c),V(U,c)\big) = 2000 \times 5000/500 = 20{,}000$.

By our assumption that value sets are preserved, $V(S \bowtie U, b) = V(S, b) = 50$, so the estimated size of the result is

$$T(R)T(S \bowtie U)/\max\big(V(R,b), V(S \bowtie U, b)\big)$$

which is $1000 \times 20{,}000/50$, or 400,000. □

16.4.5 Natural Joins With Multiple Join Attributes

When the set of attributes Y in the join $R(X,Y) \bowtie S(Y,Z)$ consists of more than one attribute, the same argument as we used for a single attribute Y applies to each attribute in Y. That is:

- The estimate of the size of $R \bowtie S$ is computed by multiplying $T(R)$ by $T(S)$ and dividing by the larger of $V(R,y)$ and $V(S,y)$ for each attribute y that is common to R and S.

Example 16.24: The following example uses the rule above. It also illustrates that the analysis we have been doing for natural joins applies to any equijoin. Consider the join

$$R(a,b,c) \bowtie_{R.b=S.d \text{ AND } R.c=S.e} S(d,e,f)$$

Suppose we have the following size parameters:

$R(a,b,c)$	$S(d,e,f)$
$T(R) = 1000$	$T(S) = 2000$
$V(R,b) = 20$	$V(S,d) = 50$
$V(R,c) = 100$	$V(S,e) = 50$

We can think of this join as a natural join if we regard $R.b$ and $S.d$ as the same attribute and also regard $R.c$ and $S.e$ as the same attribute. Then the rule given above tells us the estimate for the size of $R \bowtie S$ is the product 1000×2000 divided by the larger of 20 and 50 and also divided by the larger of 100 and 50. Thus, the size estimate for the join is $1000 \times 2000/(50 \times 100) = 400$ tuples. □

Example 16.25: Let us reconsider Example 16.23, but consider the third possible order for the joins, where we first take $R(a,b) \bowtie U(c,d)$. This join is actually a product, and the number of tuples in the result is $T(R)T(U) = 1000 \times 5000 = 5{,}000{,}000$. Note that the number of different b's in the product is $V(R,b) = 20$, and the number of different c's is $V(U,c) = 500$.

When we join this product with $S(b,c)$, we multiply the numbers of tuples and divide by both $\max\big(V(R,b), V(S,b)\big)$ and $\max\big(V(U,c), V(S,c)\big)$. This quantity is $2000 \times 5{,}000{,}000/(50 \times 500) = 400{,}000$. Note that this third way of joining gives the same estimate for the size of the result that we found in Example 16.23. □

16.4.6 Joins of Many Relations

Finally, let us consider the general case of a natural join:

$$S = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$$

Suppose that attribute A appears in k of the R_i's, and the numbers of its sets of values in these k relations — that is, the various values of $V(R_i, A)$ for $i = 1, 2, \ldots, k$ — are $v_1 \leq v_2 \leq \cdots \leq v_k$, in order from smallest to largest. Suppose we pick a tuple from each relation. What is the probability that all tuples selected agree on attribute A?

In answer, consider the tuple t_1 chosen from the relation that has the smallest number of A-values, v_1. By the containment-of-value-sets assumption, each of these v_1 values is among the A-values found in the other relations that have attribute A. Consider the relation that has v_i values in attribute A. Its selected tuple t_i has probability $1/v_i$ of agreeing with t_1 on A. Since this claim is true for all $i = 2, 3, \ldots, k$, the probability that all k tuples agree on A is the product $1/v_2 v_3 \cdots v_k$. This analysis gives us the rule for estimating the size of any join.

- Start with the product of the number of tuples in each relation. Then, for each attribute A appearing at least twice, divide by all but the least of the $V(R, A)$'s.

Likewise, we can estimate the number of values that will remain for attribute A after the join. By the preservation-of-value-sets assumption, it is the least of these $V(R, A)$'s.

Example 16.26: Consider the join $R(a, b, c) \bowtie S(b, c, d) \bowtie U(b, e)$, and suppose the important statistics are as given in Fig. 16.26. To estimate the size of this join, we begin by multiplying the relation sizes; $1000 \times 2000 \times 5000$. Next, we look at the attributes that appear more than once; these are b, which appears three times, and c, which appears twice. We divide by the two largest of $V(R, b)$, $V(S, b)$, and $V(U, b)$; these are 50 and 200. Finally, we divide by the larger of $V(R, c)$ and $V(S, c)$, which is 200. The resulting estimate is

$$1000 \times 2000 \times 5000/(50 \times 200 \times 200) = 5000$$

We can also estimate the number of values for each of the attributes in the join. Each estimate is the least value count for the attribute among all the relations in which it appears. These numbers are, for a, b, c, d, e respectively: 100, 20, 100, 400, and 500. □

Based on the two assumptions we have made — containment and preservation of value sets — we have a surprising and convenient property of the estimating rule given above.

- No matter how we group and order the terms in a natural join of n relations, the estimation rules, applied to each join individually, yield the

$R(a,b,c)$	$S(b,c,d)$	$U(b,e)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R,a) = 100$		
$V(R,b) = 20$	$V(S,b) = 50$	$V(U,b) = 200$
$V(R,c) = 200$	$V(S,c) = 100$	
	$V(S,d) = 400$	
		$V(U,e) = 500$

Figure 16.26: Parameters for Example 16.26

same estimate for the size of the result. Moreover, this estimate is the same that we get if we apply the rule for the join of all n relations as a whole.

Examples 16.23 and 16.25 form an illustration of this rule for the three groupings of a three-relation join, including the grouping where one of the "joins" is actually a product.

16.4.7 Estimating Sizes for Other Operations

We have seen two operations — selection and join — with reasonable estimating techniques. In addition, projections do not change the number of tuples in a relation, and products multiply the numbers of tuples in the argument relations. However, for the remaining operations, the size of the result is not easy to determine. We shall review the other relational-algebra operators and give some suggestions as to how this estimation could be done.

Union

If the bag union is taken, then the size is exactly the sum of the sizes of the arguments. A set union can be as large as the sum of the sizes or as small as the larger of the two arguments. We suggest that something in the middle be chosen, e.g., the larger plus half the smaller.

Intersection

The result can have as few as 0 tuples or as many as the smaller of the two arguments, regardless of whether set- or bag-intersection is taken. One approach is to take the average of the extremes, which is half the smaller.

Difference

When we compute $R - S$, the result can have between $T(R)$ and $T(R) - T(S)$ tuples. We suggest the average as an estimate: $T(R) - T(S)/2$.

Duplicate Elimination

If $R(a_1, a_2, \ldots, a_n)$ is a relation, then $V(R, [a_1, a_2, \ldots, a_n])$ is the size of $\delta(R)$. However, often we shall not have this statistic available, so it must be approximated. In the extremes, the size of $\delta(R)$ could be the same as the size of R (no duplicates) or as small as 1 (all tuples in R are the same).[4] Another upper limit on the number of tuples in $\delta(R)$ is the maximum number of distinct tuples that could exist: the product of $V(R, a_i)$ for $i = 1, 2, \ldots, n$. That number could be smaller than other estimates of $T(\delta(R))$. There are several rules that could be used to estimate $T(\delta(R))$. One reasonable one is to take the smaller of $T(R)/2$ and the product of all the $V(R, a_i)$'s.

Grouping and Aggregation

Suppose we have an expression $\gamma_L(R)$, the size of whose result we need to estimate. If the statistic $V(R, [g_1, g_2, \ldots, g_k])$, where the g_i's are the grouping attributes in L, is available, then that is our answer. However, that statistic may well not be obtainable, so we need another way to estimate the size of $\gamma_L(R)$. The number of tuples in $\gamma_L(R)$ is the same as the number of groups. There could be as few as one group in the result or as many groups as there are tuples in R. As with δ, we can also upper-bound the number of groups by a product of $V(R, A)$'s, but here attribute A ranges over only the grouping attributes of L. We again suggest an estimate that is the smaller of $T(R)/2$ and this product.

16.4.8 Exercises for Section 16.4

Exercise 16.4.1: Below are the vital statistics for four relations, W, X, Y, and Z:

$W(a,b)$	$X(b,c)$	$Y(c,d)$	$Z(d,e)$
$T(W) = 100$	$T(X) = 200$	$T(Y) = 300$	$T(Z) = 400$
$V(W,a) = 20$	$V(X,b) = 50$	$V(Y,c) = 50$	$V(Z,d) = 40$
$V(W,b) = 60$	$V(X,c) = 100$	$V(Y,d) = 50$	$V(Z,e) = 100$

Estimate the sizes of relations that are the results of the following expressions:

(a) $W \bowtie X \bowtie Y \bowtie Z$ (b) $\sigma_{a=10}(W)$ (c) $\sigma_{c=20}(Y)$
(d) $\sigma_{c=20}(Y) \bowtie Z$ (e) $W \times Y$ (f) $\sigma_{d>10}(Z)$
(g) $\sigma_{a=1 \text{ AND } b=2}(W)$ (h) $\sigma_{a=1 \text{ AND } b>2}(W)$ (i) $X \bowtie_{X.c<Y.c} Y$

Exercise 16.4.2: Here are the statistics for four relations E, F, G, and H:

[4] Strictly speaking, if R is empty there are no tuples in either R or $\delta(R)$, so the lower bound is 0. However, we are rarely interested in this special case.

$E(a,b,c)$	$F(a,b,d)$	$G(a,c,d)$	$H(b,c,d)$
$T(E) = 1000$	$T(F) = 2000$	$T(G) = 3000$	$T(H) = 4000$
$V(E,a) = 1000$	$V(F,a) = 50$	$V(G,a) = 50$	$V(H,b) = 40$
$V(E,b) = 50$	$V(F,b) = 100$	$V(G,c) = 300$	$V(H,c) = 100$
$V(E,c) = 20$	$V(F,d) = 200$	$V(G,d) = 500$	$V(H,d) = 400$

How many tuples does the join of these tuples have, using the techniques for estimation from this section?

! **Exercise 16.4.3:** How would you estimate the size of a semijoin?

!! **Exercise 16.4.4:** Suppose we compute $R(a,b) \bowtie S(a,c)$, where R and S each have 1000 tuples. The a attribute of each relation has 100 different values, and they are the *same* 100 values. If the distribution of values was uniform; i.e., each a-value appeared in exactly 10 tuples of each relation, then there would be 10,000 tuples in the join. Suppose instead that the 100 a-values have the same Zipfian distribution in each relation. Precisely, let the values be $a_1, a_2, \ldots, a_{100}$. Then the number of tuples of both R and S that have a-value a_i is proportional to $1/\sqrt{i}$. Under these circumstances, how many tuples does the join have? You should ignore the fact that the number of tuples with a given a-value may not be an integer.

16.5 Introduction to Cost-Based Plan Selection

Whether selecting a logical query plan or constructing a physical query plan from a logical plan, the query optimizer needs to estimate the cost of evaluating certain expressions. We study the issues involved in cost-based plan selection here, and in Section 16.6 we consider in detail one of the most important and difficult problems in cost-based plan selection: the selection of a join order for several relations.

As before, we shall assume that the "cost" of evaluating an expression is approximated well by the number of disk I/O's performed. The number of disk I/O's, in turn, is influenced by:

1. The particular logical operators chosen to implement the query, a matter decided when we choose the logical query plan.
2. The sizes of intermediate results, whose estimation we discussed in Section 16.4.
3. The physical operators used to implement logical operators, e.g., the choice of a one-pass or two-pass join, or the choice to sort or not sort a given relation; this matter is discussed in Section 16.7.
4. The ordering of similar operations, especially joins as discussed in Section 16.6.

5. The method of passing arguments from one physical operator to the next, which is also discussed in Section 16.7.

Many issues need to be resolved in order to perform effective cost-based plan selection. In this section, we first consider how the size parameters, which were so essential for estimating relation sizes in Section 16.4, can be obtained from the database efficiently. We then revisit the algebraic laws we introduced to find the preferred logical query plan. Cost-based analysis justifies the use of many of the common heuristics for transforming logical query plans, such as pushing selections down the tree. Finally, we consider the various approaches to enumerating all the physical query plans that can be derived from the selected logical plan. Especially important are methods for reducing the number of plans that need to be evaluated, while making it likely that the least-cost plan is still considered.

16.5.1 Obtaining Estimates for Size Parameters

The formulas of Section 16.4 were predicated on knowing certain important parameters, especially $T(R)$, the number of tuples in a relation R, and $V(R, a)$, the number of different values in the column of relation R for attribute a. A modern DBMS generally allows the user or administrator explicitly to request the gathering of statistics, such as $T(R)$ and $V(R, a)$. These statistics are then used in query optimization, unchanged until the next command to gather statistics.

By scanning an entire relation R, it is straightforward to count the number of tuples $T(R)$ and also to discover the number of different values $V(R, a)$ for each attribute a. The number of blocks in which R can fit, $B(R)$, can be estimated either by counting the actual number of blocks used (if R is clustered), or by dividing $T(R)$ by the number of R's tuples that can fit in one block.

In addition, a DBMS may compute a *histogram* of the values for a given attribute. If $V(R, A)$ is not too large, then the histogram may consist of the number (or fraction) of the tuples having each of the values of attribute A. If there are many values of this attribute, then only the most frequent values may be recorded individually, while other values are counted in groups. The most common types of histograms are:

1. *Equal-width.* A width w is chosen, along with a constant v_0. Counts are provided of the number of tuples with values v in the ranges $v_0 \leq v < v_0 + w$, $v_0 + w \leq v < v_0 + 2w$, and so on. The value v_0 may be the lowest possible value or a lower bound on values seen so far. In the latter case, should a new, lower value be seen, we can lower the value of v_0 by w and add a new count to the histogram.

2. *Equal-height.* These are the common "percentiles." We pick some fraction p, and list the lowest value, the value that is fraction p from the lowest, the fraction $2p$ from the lowest, and so on, up to the highest value.

3. *Most-frequent-values.* We may list the most common values and their numbers of occurrences. This information may be provided along with a count of occurrences for all the other values as a group, or we may record frequent values in addition to an equal-width or equal-height histogram for the other values.

One advantage of keeping a histogram is that the sizes of joins can be estimated more accurately than by the simplified methods of Section 16.4. In particular, if a value of the join attribute appears explicitly in the histograms of both relations being joined, then we know exactly how many tuples of the result will have this value. For those values of the join attribute that do not appear explicitly in the histogram of one or both relations, we estimate their effect on the join as in Section 16.4. However, if we use an equal-width histogram, with the same bands for the join attributes of both relations, then we can estimate the size of the joins of corresponding bands, and sum those estimates. The result will be a good estimate, because only tuples in corresponding bands can join. The following examples will suggest how to carry out histogram-based estimation; we shall not use histograms in estimates subsequently.

Example 16.27: Consider histograms that mention the three most frequent values and their counts, and group the remaining values. Suppose we want to compute the join $R(a,b) \bowtie S(b,c)$. Let the histogram for $R.b$ be:

1: 200, 0: 150, 5: 100, others: 550

That is, of the 1000 tuples in R, 200 of them have b-value 1, 150 have b-value 0, and 100 have b-value 5. In addition, 550 tuples have b-values other than 0, 1, or 5, and none of these other values appears more than 100 times.

Let the histogram for $S.b$ be:

0: 100, 1: 80, 2: 70, others: 250

Suppose also that $V(R,b) = 14$ and $V(S,b) = 13$. That is, the 550 tuples of R with unknown b-values are divided among eleven values, for an average of 50 tuples each, and the 250 tuples of S with unknown b-values are divided among ten values, each with an average of 25 tuples each.

Values 0 and 1 appear explicitly in both histograms, so we can calculate that the 150 tuples of R with $b = 0$ join with the 100 tuples of S having the same b-value, to yield 15,000 tuples in the result. Likewise, the 200 tuples of R with $b = 1$ join with the 80 tuples of S having $b = 1$ to yield 16,000 more tuples in the result.

The estimate of the effect of the remaining tuples is more complex. We shall continue to make the assumption that every value appearing in the relation with the smaller set of values (S in this case) will also appear in the set of values of the other relation. Thus, among the eleven remaining b-values of S, we know one of those values is 2, and we shall assume another of the values is 5, since

that is one of the most frequent values in R. We estimate that 2 appears 50 times in R, and 5 appears 25 times in S. These estimates are each obtained by assuming that the value is one of the "other" values for its relation's histogram. The number of additional tuples from b-value 2 is thus $70 \times 50 = 3500$, and the number of additional tuples from b-value 5 is $100 \times 25 = 2500$.

Finally, there are nine other b-values that appear in both relations, and we estimate that each of them appears in 50 tuples of R and 25 tuples of S. Each of the nine values thus contributes $50 \times 25 = 1250$ tuples to the result. The estimate of the output size is thus:

$$15000 + 16000 + 3500 + 2500 + 9 \times 1250$$

or 48,250 tuples. Note that the simpler estimate from Section 16.4 would be $1000 \times 500/14$, or 35,714, based on the assumptions of equal numbers of occurrences of each value in each relation. □

Example 16.28: In this example, we shall assume an equal-width histogram, and we shall demonstrate how knowing that values of two relations are almost disjoint can impact the estimate of a join size. Our relations are:

```
Jan(day, temp)
July(day, temp)
```

and the query is:

```
SELECT Jan.day, July.day
FROM Jan, July
WHERE Jan.temp = July.temp;
```

That is, find pairs of days in January and July that had the same temperature. The query plan is to equijoin `Jan` and `July` on the temperature, and project onto the two `day` attributes.

Suppose the histogram of temperatures for the relations `Jan` and `July` are as given in the table of Fig. 16.27.[5] In general, if both join attributes have equal-width histograms with the same set of bands, then we can estimate the size of the join by considering each pair of corresponding bands and summing.

If two corresponding bands have T_1 and T_2 tuples, respectively, and the number of values in a band is V, then the estimate for the number of tuples in the join of those bands is T_1T_2/V, following the principles laid out in Section 16.4.4. For the histograms of Fig. 16.27, many of these products are 0, because one or the other of T_1 and T_2 is 0. The only bands for which neither is 0 are 40–49 and 50–59. Since $V = 10$ is the width of a band, the 40–49 band contributes $10 \times 5/10 = 5$ tuples, and the 50–59 band contributes $5 \times 20/10 = 10$ tuples.

[5] Our friends south of the equator should reverse the columns for January and July, and convert to centigrade as well.

Range	Jan	July
0–9	40	0
10–19	60	0
20–29	80	0
30–39	50	0
40–49	10	5
50–59	5	20
60–69	0	50
70–79	0	100
80–89	0	60
90–99	0	10

Figure 16.27: Histograms of temperature

Thus our estimate for the size of this join is $5 + 10 = 15$ tuples. If we had no histogram, and knew only that each relation had 245 tuples distributed among 100 values from 0 to 99, then our estimate of the join size would be $245 \times 245/100 = 600$ tuples. □

16.5.2 Computation of Statistics

Statistics normally are computed only periodically, for several reasons. First, statistics tend not to change radically in a short time. Second, even somewhat inaccurate statistics are useful as long as they are applied consistently to all the plans. Third, the alternative of keeping statistics up-to-date can make the statistics themselves into a "hot-spot" in the database; because statistics are read frequently, we prefer not to update them frequently too.

The recomputation of statistics might be triggered automatically after some period of time, or after some number of updates. However, a database administrator, noticing that poor-performing query plans are being selected by the query optimizer on a regular basis, might request the recomputation of statistics in an attempt to rectify the problem.

Computing statistics for an entire relation R can be very expensive, particularly if we compute $V(R, a)$ for each attribute a in the relation (or even worse, compute histograms for each a). One common approach is to compute approximate statistics by sampling only a fraction of the data. For example, let us suppose we want to sample a small fraction of the tuples to obtain an estimate for $V(R, a)$. A statistically reliable calculation can be complex, depending on a number of assumptions, such as whether values for a are distributed uniformly, according to a Zipfian distribution, or according to some other distribution. However, the intuition is as follows. If we look at a small sample of R, say 1% of its tuples, and we find that most of the a-values we see are different, then it is likely that $V(R, a)$ is close to $T(R)$. If we find that the sample has very few different values of a, then it is likely that we have seen most of the a-values

that exist in the current relation.

16.5.3 Heuristics for Reducing the Cost of Logical Query Plans

One important use of cost estimates for queries or subqueries is in the application of heuristic transformations of the query. We already have observed in Section 16.3.3 how certain heuristics, such as pushing selections down the tree, can be expected almost certainly to improve the cost of a logical query plan, regardless of relation sizes. However, there are other points in the query optimization process where estimating the cost both before and after a transformation will allow us to apply a transformation where it appears to reduce cost and avoid the transformation otherwise. In particular, when the preferred logical query plan is being generated, we may consider a number of optional transformations and the costs before and after.

Because we are estimating the cost of a *logical* query plan, and so we have not yet made decisions about the physical operators that will be used to implement the operators of relational algebra, our cost estimate cannot be based on disk I/O's. Rather, we estimate the sizes of all intermediate results using the techniques of Section 16.4, and their sum is our heuristic estimate for the cost of the entire logical plan. One example will serve to illustrate the issues and process.

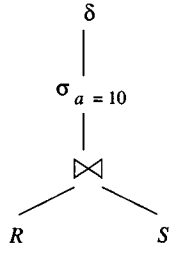

Figure 16.28: Logical query plan for Example 16.29

Example 16.29: Consider the initial logical query plan of Fig. 16.28, and let the statistics for the relations R and S be as follows:

$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$

To generate a final logical query plan from Fig. 16.28, we shall insist that the selection be pushed down as far as possible. However, we are not sure whether

it makes sense to push the δ below the join or not. Thus, we generate from Fig. 16.28 the two query plans shown in Fig. 16.29; they differ in whether we have chosen to eliminate duplicates before or after the join. Notice that in plan (a) the δ is pushed down both branches of the tree. If R and/or S is known to have no duplicates, then the δ along its branch could be eliminated.

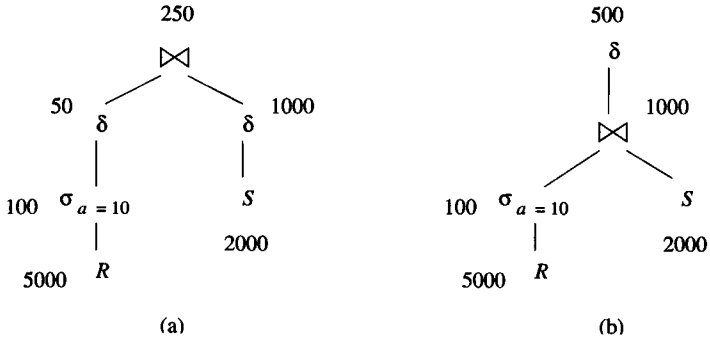

Figure 16.29: Two candidates for the best logical query plan

We know how to estimate the size of the result of the selections, from Section 16.4.3; we divide $T(R)$ by $V(R,a) = 50$. We also know how to estimate the size of the joins; we multiply the sizes of the arguments and divide by $\max\big(V(R,b), V(S,b)\big)$, which is 200. What we don't know is how to estimate the size of the relations with duplicates eliminated.

First, consider the size estimate for $\delta\big(\sigma_{a=10}(R)\big)$. Since $\sigma_{a=10}(R)$ has only one value for a and up to 100 values for b, and there are an estimated 100 tuples in this relation, the rule from Section 16.4.7 tells us that the product of the value counts for each of the attributes is not a limiting factor. Thus, we estimate the size of the result of δ as half the tuples in $\sigma_{a=10}(R)$, and Fig. 16.29(a) shows an estimate of 50 tuples for $\delta\big(\sigma_{a=10}(R)\big)$.

Now, consider the estimate of the result of the δ in Fig. 16.29(b). The join has one value for a, an estimated $\min\big(V(R,b), V(S,b)\big) = 100$ values for b, and an estimated $V(S,c) = 100$ values for c. Thus again the product of the value counts does not limit how big the result of the δ can be. We estimate this result as 500 tuples, or half the number of tuples in the join.

To compare the two plans of Fig. 16.29, we add the estimated sizes for all the nodes except the root and the leaves. We exclude the root and leaves, because these sizes are not dependent on the plan chosen. For plan (a) this cost, the sum of the estimated sizes of the interior nodes, is $100 + 50 + 1000 = 1150$, while for plan (b) the sum is $100 + 1000 = 1100$. Thus, by a small margin we conclude that deferring the duplicate elimination to the end is a better plan. We would come to the opposite conclusion if, say, R or S had fewer b-values. Then the join size would be greater, making the cost of plan (b) greater. □

Estimates for Result Sizes Need Not Be the Same

Notice that in Fig. 16.29 the estimates at the roots of the two trees are different: 250 in one case and 500 in the other. Because estimation is an inexact science, these sorts of anomalies will occur. In fact, it is the exception when we can offer a guarantee of consistency, as we did in Section 16.4.6.

Intuitively, the estimate for plan (b) is higher because if there are duplicates in both R and S, these duplicates will be multiplied in the join; e.g., for tuples that appear 3 times in R and twice in S, their join will appear six times in $R \bowtie S$. Our simple formula for estimating the size of the result of a δ does not take into account the possibility that the effect of duplicates has been amplified by previous operations.

16.5.4 Approaches to Enumerating Physical Plans

Now, let us consider the use of cost estimates in the conversion of a logical query plan to a physical query plan. The baseline approach, called *exhaustive*, is to consider all combinations of choices for each of the issues outlined at the beginning of Section 16.4 (order of joins, physical implementation of operators, and so on). Each possible physical plan is assigned an estimated cost, and the one with the smallest cost is selected.

However, there are a number of other approaches to selection of a physical plan. In this section, we shall outline various approaches that have been used, while Section 16.6 focuses on selecting a join order. Before proceeding, let us comment that there are two broad approaches to exploring the space of possible physical plans:

- *Top-down*: Here, we work down the tree of the logical query plan from the root. For each possible implementation of the operation at the root, we consider each possible way to evaluate its argument(s), and compute the cost of each combination, taking the best.[6]
- *Bottom-up*: For each subexpression of the logical-query-plan tree, we compute the costs of all possible ways to compute that subexpression. The possibilities and costs for a subexpression E are computed by considering the options for the subexpressions of E, and combining them in all possible ways with implementations for the root operator of E.

There is actually not much difference between the two approaches in their broadest interpretations, since either way, all possible combinations of ways to

[6]Remember from Section 16.3.4 that a single node of the logical-query-plan tree may represent many uses of a single commutative and associative operator, such as join. Thus, the consideration of all possible plans for a single node may itself involve enumeration of very many choices.

implement each operator in the query tree are considered. We shall concentrate on bottom-up methods in what follows.

You may, in fact, have noticed that there is an apparent simplification of the bottom-up method, where we consider only the *best* plan for each subexpression when we compute the plans for a larger subexpression. This approach, called *dynamic programming* in the list of methods below, is not guaranteed to yield the best overall plan, although often it does. The approach called *Selinger-style* (or *System-R-style*) optimization, also listed below, exploits additional properties that some of the plans for a subexpression may have, in order to produce optimal overall plans from plans that are not optimal for certain subexpressions.

Heuristic Selection

One option is to use the same approach to selecting a physical plan that is generally used for selecting a logical plan: make a sequence of choices based on heuristics. In Section 16.6.6, we shall discuss a "greedy" heuristic for join ordering, where we start by joining the pair of relations whose result has the smallest estimated size, then repeat the process for the result of that join and the other relations in the set to be joined. There are many other heuristics that may be applied; here are some of the most commonly used ones:

1. If the logical plan calls for a selection $\sigma_{A=c}(R)$, and stored relation R has an index on attribute A, then perform an index-scan (as in Section 15.1.1) to obtain only the tuples of R with A-value equal to c.

2. More generally, if the selection involves one condition like $A = c$ above, and other conditions as well, we can implement the selection by an index-scan followed by a further selection on the tuples, which we shall represent by the physical operator *filter*. This matter is discussed further in Section 16.7.1.

3. If an argument of a join has an index on the join attribute(s), then use an index-join with that relation in the inner loop.

4. If one argument of a join is sorted on the join attribute(s), then prefer a sort-join to a hash-join, although not necessarily to an index-join if one is possible.

5. When computing the union or intersection of three or more relations, group the smallest relations first.

Branch-and-Bound Plan Enumeration

This approach, often used in practice, begins by using heuristics to find a good physical plan for the entire logical query plan. Let the cost of this plan be C. Then as we consider other plans for subqueries, we can eliminate any plan for a subquery that has a cost greater than C, since that plan for the subquery

could not possibly participate in a plan for the complete query that is better than what we already know. Likewise, if we construct a plan for the complete query that has cost less than C, we replace C by the cost of this better plan in subsequent exploration of the space of physical query plans.

An important advantage of this approach is that we can choose when to cut off the search and take the best plan found so far. For instance, if the cost C is small, then even if there are much better plans to be found, the time spent finding them may exceed C, so it does not make sense to continue the search. However, if C is large, then investing time in the hope of finding a faster plan is wise.

Hill Climbing

This approach, in which we really search for a "valley" in the space of physical plans and their costs, starts with a heuristically selected physical plan. We can then make small changes to the plan, e.g., replacing one method for executing an operator by another, or reordering joins by using the associative and/or commutative laws, to find "nearby" plans that have lower cost. When we find a plan such that no small modification yields a plan of lower cost, we make that plan our chosen physical query plan.

Dynamic Programming

In this variation of the general bottom-up strategy, we keep for each subexpression only the plan of least cost. As we work up the tree, we consider possible implementations of each node, assuming the best plan for each subexpression is also used. We examine this approach extensively in Section 16.6.

Selinger-Style Optimization

This approach improves upon the dynamic-programming approach by keeping for each subexpression not only the plan of least cost, but certain other plans that have higher cost, yet produce a result that is sorted in an order that may be useful higher up in the expression tree. Examples of such *interesting* orders are when the result of the subexpression is sorted on one of:

1. The attribute(s) specified in a sort (τ) operator at the root.
2. The grouping attribute(s) of a later group-by (γ) operator.
3. The join attribute(s) of a later join.

If we take the cost of a plan to be the sum of the sizes of the intermediate relations, then there appears to be no advantage to having an argument sorted. However, if we use the more accurate measure, disk I/O's, as the cost, then the advantage of having an argument sorted becomes clear if we can use one of the sort-based algorithms of Section 15.4, and save the work of the first pass for the argument that is sorted already.

16.5.5 Exercises for Section 16.5

Exercise 16.5.1: Estimate the size of the join $R(a,b) \bowtie S(b,c)$ using histograms for $R.b$ and $S.b$. Assume $V(R,b) = V(S,b) = 20$, and the histograms for both attributes give the frequency of the four most common values, as tabulated below:

	0	1	2	3	4	others
$R.b$	5	6	4	5		32
$S.b$	10	8	5		7	48

How does this estimate compare with the simpler estimate, assuming that all 20 values are equally likely to occur, with $T(R) = 52$ and $T(S) = 78$?

Exercise 16.5.2: Estimate the size of the join $R(a,b) \bowtie S(b,c)$ if we have the following histogram information:

	$b < 0$	$b = 0$	$b > 0$
R	500	100	400
S	300	200	500

! **Exercise 16.5.3:** In Example 16.29 we suggested that reducing the number of values that either attribute named b had could make plan (a) better than plan (b) of Fig. 16.29. For what values of:

a) $V(R,b)$

b) $V(S,b)$

will plan (a) have a lower estimated cost than plan (b)?

! **Exercise 16.5.4:** Consider four relations R, S, T, and V. Respectively, they have 200, 300, 400, and 500 tuples, chosen randomly and independently from the same pool of 1000 tuples (e.g., the probabilities of a given tuple being in R is 1/5, in S is 3/10, and in both is 3/50).

a) What is the expected size of $R \cup S \cup T \cup V$?

b) What is the expected size of $R \cap S \cap T \cap V$?

c) What order of unions gives the least cost (estimated sum of the sizes of the intermediate relations)?

d) What order of intersections gives the least cost (estimated sum of the sizes of the intermediate relations)?

! **Exercise 16.5.5:** Repeat Exercise 16.5.4 if all four relations have 500 of the 1000 tuples, at random.

!! **Exercise 16.5.6:** Suppose we wish to compute the expression

$$\tau_b\big(R(a,b) \bowtie S(b,c) \bowtie T(c,d)\big)$$

That is, we join the three relations and produce the result sorted on attribute b. Let us make the simplifying assumptions:

i. We shall not "join" R and T first, because that is a product.

ii. Any other join can be performed with a two-pass sort-join or hash-join, but in no other way.

iii. Any relation, or the result of any expression, can be sorted by a two-phase, multiway merge-sort, but in no other way.

iv. The result of the first join will be passed as an argument to the last join one block at a time and not stored temporarily on disk.

v. Each relation occupies 1000 blocks, and the result of either join of two relations occupies 5000 blocks.

Answer the following based on these assumptions:

a) What are all the subexpressions and orders that a Selinger-style optimization would consider?

b) Which query plan uses the fewest disk I/O's?[7]

!! **Exercise 16.5.7:** Give an example of a logical query plan of the form $E \bowtie F$, for some expressions E and F (which you may choose), where using the best plans to evaluate E and F does not allow any choice of algorithm for the final join that minimizes the total cost of evaluating the entire expression. Make whatever assumptions you wish about the number of available main-memory buffers and the sizes of relations mentioned in E and F.

16.6 Choosing an Order for Joins

In this section we focus on a critical problem in cost-based optimization: selecting an order for the (natural) join of three or more relations. Similar ideas can be applied to other binary operations like union or intersection, but these operations are less important in practice, because they typically take less time to execute than joins, and they more rarely appear in clusters of three or more.

[7]Notice that, because we have made some very specific assumptions about the join methods to be used, we can estimate disk I/O's, instead of relying on the simpler, but less accurate, counts of tuples as our cost measure.

16.6.1 Significance of Left and Right Join Arguments

When ordering a join, we should remember that many of the join methods discussed in Chapter 15 are asymmetric. That is, the roles played by the two argument relations are different, and the cost of the join depends on which relation plays which role. Perhaps most important, the one-pass join of Section 15.2.3 reads one relation — preferably the smaller — into main memory, creating a structure such as a hash table to facilitate matching of tuples from the other relation. It then reads the other relation, one block at a time, to join its tuples with the tuples stored in memory.

For instance, suppose that when we select a physical plan we decide to use a one-pass join. Then we shall assume the left argument of the join is the smaller relation and store it in a main-memory data structure. This relation is called the *build relation*. The right argument of the join, called the *probe relation*, is read a block at a time and its tuples are matched in main memory with those of the build relation. Other join algorithms that distinguish between their arguments include:

1. Nested-loop join, where we assume the left argument is the relation of the outer loop.

2. Index-join, where we assume the right argument has the index.

16.6.2 Join Trees

When we have the join of two relations, we need to order the arguments. We shall conventionally select the one whose estimated size is the smaller as the left argument. It is quite common for there to be a significant and discernible difference in the sizes of arguments, because a query involving joins often also involves a selection on at least one attribute, and that selection reduces the estimated size of one of the relations greatly.

Example 16.30: Recall the query

```
SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name AND
    birthdate LIKE '%1960';
```

from Fig. 16.4, which leads to the preferred logical query plan of Fig. 16.24, in which we take the join of relation `StarsIn` and the result of a selection on relation `MovieStar`. We have not given estimates for the sizes of relations `StarsIn` or `MovieStar`, but we can assume that selecting for stars born in a single year will produce about 1/50th of the tuples in `MovieStar`. Since there are generally several stars per movie, we expect `StarsIn` to be larger than `MovieStar` to begin with, so the second argument of the join, $\sigma_{birthdate \texttt{ LIKE '\%1960'}}(\texttt{MovieStar})$, is much smaller than the first argument `StarsIn`. We conclude that the order of

arguments in Fig. 16.24 should be reversed, so that the selection on `MovieStar` is the left argument. □

There are only two choices for a join tree when there are two relations — take either of the two relations to be the left argument. When the join involves more than two relations, the number of possible join trees grows rapidly. For example, Fig. 16.30 shows three of the five shapes of trees in which four relations R, S, T, and U, are joined. However, each of these trees has the four relations in alphabetical order from the left. Since order of arguments matters, and there are $n!$ ways to order n things, each tree represents $4! = 24$ different trees when the possible labelings of the leaves are considered.

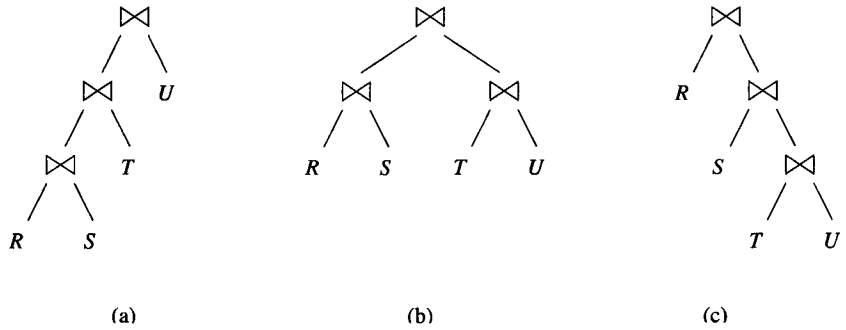

Figure 16.30: Ways to join four relations

16.6.3 Left-Deep Join Trees

Figure 16.30(a) is an example of what is called a *left-deep* tree. In general, a binary tree is left-deep if all right children are leaves. Similarly, a tree like Fig. 16.30(c), all of whose left children are leaves, is called a *right-deep* tree. A tree such as Fig. 16.30(b), that is neither left-deep nor right-deep, is called *bushy*. We shall argue below that there is a two-fold advantage to considering only left-deep trees as possible join orders.

1. The number of possible left-deep trees with a given number of leaves is large, but not nearly as large as the number of all trees. Thus, searches for query plans can be used for larger queries if we limit the search to left-deep trees.

2. Left-deep trees for joins interact well with common join algorithms — nested-loop joins and one-pass joins in particular. Query plans based on left-deep trees plus these join implementations will tend to be more efficient than the same algorithms used with non-left-deep trees.

The "leaves" in a left- or right-deep join tree can actually be interior nodes, with operators other than a join. Thus, for instance, Fig. 16.24 is technically a

left-deep join tree with one join operator. The fact that a selection is applied to the right operand of the join does not take the tree out of the left-deep join class.

The number of left-deep trees does not grow nearly as fast as the number of all trees for the multiway join of a given number of relations. For n relations, there is only one left-deep tree shape, to which we may assign the relations in $n!$ ways. There are the same number of right-deep trees for n relations. However, the total number of tree shapes $T(n)$ for n relations is given by the recurrence:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= \sum_{i=1}^{n-1} T(i)T(n-i) \end{aligned}$$

The explanation for the second equation is that we may pick any number i between 1 and $n-1$ to be the number of leaves in the left subtree of the root, and those leaves may be arranged in any of the $T(i)$ ways that trees with i leaves can be arranged. Similarly, the remaining $n-i$ leaves in the right subtree can be arranged in any of $T(n-i)$ ways.

The first few values of $T(n)$ are:

n	1	2	3	4	5	6
$T(n)$	1	1	2	5	14	42

To get the total number of trees once relations are assigned to the leaves, we multiply $T(n)$ by $n!$. Thus, for instance, the number of leaf-labeled trees of 6 leaves is $42 \times 6!$ or 30,240, of which 6!, or 720, are left-deep trees and another 720 are right-deep trees.

Now, let us consider the second advantage mentioned for left-deep join trees: their tendency to produce efficient plans. We shall give two examples:

1. If one-pass joins are used, and the build relation is on the left, then the amount of memory needed at any one time tends to be smaller than if we used a right-deep tree or a bushy tree for the same relations.

2. If we use nested-loop joins, with the relation of the outer loop on the left, then we avoid constructing any intermediate relation more than once.

Example 16.31: Consider the left-deep tree in Fig. 16.30(a), and suppose that we use a simple one-pass join for each of the three $\bowtie$ operators. As always, the left argument is the build relation; i.e., it will be held in main memory. To compute $R \bowtie S$, we need to keep R in main memory, and as we compute $R \bowtie S$ we need to keep the result in main memory as well. Thus, we need $B(R) + B(R \bowtie S)$ main-memory buffers. If we pick R to be the smallest of the relations, and a selection has made R be rather small, then there is likely to be no problem making this number of buffers available.

Having computed $R \bowtie S$, we must join this relation with T. However, the buffers used for R are no longer needed and can be reused to hold (some of) the result of $(R \bowtie S) \bowtie T$. Similarly, when we join this relation with U, the

Role of the Buffer Manager

The reader may notice a difference between our approach in the series of examples such as Example 15.4 and 15.6, where we assumed that there was a fixed limit on the number of main-memory buffers available for a join, and the more flexible assumption taken here, where we assume that as many buffers as necessary are available, but we try not to use "too many." Recall from Section 15.7 that the buffer manager has significant flexibility to allocate buffers to operations. However, if too many buffers are allocated at once, there will be thrashing, thus degrading the assumed performance of the algorithm being used.

relation $R \bowtie S$ is no longer needed, and its buffers can be reused for the result of the final join. In general, a left-deep join tree that is computed by one-pass joins requires main-memory space for at most two of the temporary relations any time.

Now, let us consider a similar implementation of the right-deep tree of Fig. 16.30(c). The first thing we need to do is load R into main-memory buffers, since left arguments are always the build relation. Then, we need to construct $S \bowtie (T \bowtie U)$ and use that as the probe relation for the join at the root. To compute $S \bowtie (T \bowtie U)$ we need to bring S into buffers and then compute $T \bowtie U$ as the probe relation for S. But $T \bowtie U$ requires that we first bring T into buffers. Now we have all three of R, S, and T in memory at the same time. In general, if we try to compute a right-deep join tree with n leaves, we shall have to bring $n-1$ relations into memory simultaneously.

Of course it is possible that the total size $B(R) + B(S) + B(T)$ is less than the amount of space we need at either of the two intermediate stages of the computation of the left-deep tree, which are $B(R) + B(R \bowtie S)$ and $B(R \bowtie S) + B\big((R \bowtie S) \bowtie T\big)$, respectively. However, as we pointed out in Example 16.30, queries with several joins often will have a small relation with which we can start as the leftmost argument in a left-deep tree. If R is small, we might expect $R \bowtie S$ to be significantly smaller than S and $(R \bowtie S) \bowtie T$ to be smaller than T, further justifying the use of a left-deep tree. □

Example 16.32: Now, let us suppose we are going to implement the four-way join of Fig. 16.30 by nested-loop joins, and that we use an iterator (as in Section 15.1.6) for each of the three joins involved. Also, assume for simplicity that each of the relations R, S, T, and U are stored relations, rather than expressions. If we use the left-deep tree of Fig. 16.30(a), then the iterator at the root gets a main-memory-sized chunk of its left argument $(R \bowtie S) \bowtie T$. It then joins the chunk with all of U, but as long as U is a stored relation, it is only necessary to scan U, not to construct it. When the next chunk of the left argument is obtained and put in memory, U will be read again, but nested-loop

join requires that repetition, which cannot be avoided if both arguments are large.

Similarly, to get a chunk of $(R \bowtie S) \bowtie T$, we get a chunk of $R \bowtie S$ into memory and scan T. Several scans of T may eventually be necessary, but cannot be avoided. Finally, to get a chunk of $R \bowtie S$ requires reading a chunk of R and comparing it with S, perhaps several times. However, in all this action, only stored relations are read multiple times, and this repeated reading is an artifact of the way nested-loop join works when the main memory is insufficient to hold an entire relation.

Now, compare the behavior of iterators on the left-deep tree with the behavior of iterators on the right-deep tree of Fig. 16.30(c). The iterator at the root starts by reading a chunk of R. It must then construct the entire relation $S \bowtie (T \bowtie U)$ and compare it with that chunk of R. When we read the next chunk of R into memory, $S \bowtie (T \bowtie U)$ must be constructed again. Each subsequent chunk of R likewise requires constructing this same relation.

Of course, we could construct $S \bowtie (T \bowtie U)$ once and store it, either in memory or on disk. If we store it on disk, we are using extra disk I/O's compared with the left-deep tree's plan, and if we store it in memory, then we run into the same problem with overuse of memory that we discussed in Example 16.31. □

16.6.4 Dynamic Programming to Select a Join Order and Grouping

To pick an order for the join of many relations we have three choices:

1. Consider them all.
2. Consider a subset.
3. Use a heuristic to pick one.

We shall here consider a sensible approach to enumeration called *dynamic programming*. It can be used either to consider all orders, or to consider certain subsets only, such as orders restricted to left-deep trees. In Section 16.6.6 we consider a heuristic for selecting a single ordering. Dynamic programming is a common algorithmic paradigm.[8] The idea behind dynamic programming is that we fill in a table of costs, remembering only the minimum information we need to proceed to a conclusion.

Suppose we want to join $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$. In a dynamic programming algorithm, we construct a table with an entry for each subset of one or more of the n relations. In that table we put:

1. The estimated size of the join of these relations. For this quantity we may use the formula of Section 16.4.6.

[8]See Aho, Hopcroft and Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, for a general treatment of dynamic programming.

2. The least cost of computing the join of these relations. We shall use in our examples the sum of the sizes of the intermediate relations (not including the R_i's themselves or the join of the full set of relations associated with this table entry).

3. The expression that yields the least cost. This expression joins the set of relations in question, with some grouping. We can optionally restrict ourselves to left-deep expressions, in which case the expression is just an ordering of the relations.

The construction of this table is an induction on the subset size. There are two variations, depending on whether we wish to consider all possible tree shapes or only left-deep trees. We explain the difference when we discuss the inductive step of table construction.

BASIS: The entry for a single relation R consists of the size of R, a cost of 0, and an expression that is just R itself. The entry for a pair of relations $\{R_i, R_j\}$ is also easy to compute. The cost is 0, since there are no intermediate relations involved, and the size estimate is given by the rule of Section 16.4.6; it is the product of the sizes of R_i and R_j divided by the larger value-set size for each attribute shared by R_i and R_j, if any. The expression is either $R_i \bowtie R_j$ or $R_j \bowtie R_i$. Following the idea introduced in Section 16.6.1, we pick the smaller of R_i and R_j as the left argument.

INDUCTION: Now, we can build the table, computing entries for all subsets of size 3, 4, and so on, until we get an entry for the one subset of size n. That entry tells us the best way to compute the join of all the relations; it also gives us the estimated cost of that method, which is needed as we compute later entries. We need to see how to compute the entry for a set of k relations $\mathcal{R}$.

If we wish to consider only left-deep trees, then for each of the k relations R in $\mathcal{R}$ we consider the possibility that we compute the join for $\mathcal{R}$ by first computing the join of $\mathcal{R} - \{R\}$ and then joining it with R. The cost of the join for $\mathcal{R}$ is the cost of $\mathcal{R} - \{R\}$ plus the size of the result for $\mathcal{R} - \{R\}$. We pick whichever R yields the least cost. The expression for $\mathcal{R}$ has the best join expression for $\mathcal{R} - \{R\}$ as the left argument of a final join, and R as the right argument. The size for $\mathcal{R}$ is whatever the formula from Section 16.4.6 gives.

If we wish to consider all trees, then computing the entry for a set of relations $\mathcal{R}$ is somewhat more complex. We need to consider all ways to partition $\mathcal{R}$ into disjoint sets $\mathcal{R}_1$ and $\mathcal{R}_2$. For each such subset, we consider the sum of:

1. The best costs of $\mathcal{R}_1$ and $\mathcal{R}_2$.

2. The sizes of the results for $\mathcal{R}_1$ and $\mathcal{R}_2$.

For whichever partition gives the best cost, we use this sum as the cost for $\mathcal{R}$, and the expression for $\mathcal{R}$ is the join of the best join orders for $\mathcal{R}_1$ and $\mathcal{R}_2$.

Example 16.33: Consider the join of four relations R, S, T, and U. For simplicity, we shall assume they each have 1000 tuples. Their attributes and the estimated sizes of values sets for the attributes in each relation are summarized in Fig. 16.31.

$R(a,b)$	$S(b,c)$	$T(c,d)$	$U(d,a)$
$V(R,a) = 100$			$V(U,a) = 50$
$V(R,b) = 200$	$V(S,b) = 100$		
	$V(S,c) = 500$	$V(T,c) = 20$	
		$V(T,d) = 50$	$V(U,d) = 1000$

Figure 16.31: Parameters for Example 16.33

For the singleton sets, the sizes, costs, and best plans are as in the table of Fig. 16.32. That is, for each single relation, the size is as given, 1000 for each, the cost is 0 since there are no intermediate relations needed, and the best (and only) expression is the relation itself.

	$\{R\}$	$\{S\}$	$\{T\}$	$\{U\}$
Size	1000	1000	1000	1000
Cost	0	0	0	0
Best plan	R	S	T	U

Figure 16.32: The table for singleton sets

Now, consider the pairs of relations. The cost for each is 0, since there are still no intermediate relations in a join of two. There are two possible plans, since either of the two relations can be the left argument, but since the sizes happen to be the same for each relation we have no basis on which to choose between the plans. We shall take the first, in alphabetical order, to be the left argument in each case. The sizes of the resulting relations are computed by the usual formula. The results are summarized in Fig. 16.33.

	$\{R,S\}$	$\{R,T\}$	$\{R,U\}$	$\{S,T\}$	$\{S,U\}$	$\{T,U\}$
Size	5000	1,000,000	10,000	2000	1,000,000	1000
Cost	0	0	0	0	0	0
Best plan	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$

Figure 16.33: The table for pairs of relations

Now, consider the table for joins of three out of the four relations. The only way to compute a join of three relations is to pick two to join first. The size estimate for the result is computed by the standard formula, and we omit the

details of this calculation; remember that we'll get the same size regardless of which way we compute the join.

The cost estimate for each triple of relations is the size of the one intermediate relation — the join of the first two chosen. Since we want this cost to be as small as possible, we consider each pair of two out of the three relations and take the pair with the smallest size.

For the expression, we group the two chosen relations first, but these could be either the left or right argument. Let us suppose that we are only interested in left-deep trees, so we always use the join of the first two relations as the left argument. Since in all cases the estimated size for the join of two of our relations is at least 1000 (the size of each individual relation), were we to allow non-left-deep trees we would always select the single relation as the left argument in this example. The summary table for the triples is shown in Fig. 16.34.

	$\{R,S,T\}$	$\{R,S,U\}$	$\{R,T,U\}$	$\{S,T,U\}$
Size	10,000	50,000	10,000	2,000
Cost	2,000	5,000	1,000	1,000
Best plan	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

Figure 16.34: The table for triples of relations

Let us consider $\{R,S,T\}$ as an example of the calculation. We must consider each of the three pairs in turn. If we start with $R \bowtie S$, then the cost is the size of this relation, which is 5000 (see Fig. 16.33). Starting with $R \bowtie T$ gives us a cost of 1,000,000 for the intermediate relation, and starting with $S \bowtie T$ has a cost of 2000. Since the latter is the smallest cost of the three options, we choose that plan. The choice is reflected not only in the cost entry of the $\{R,S,T\}$ column, but in the best-plan row, where the plan that groups S and T first appears.

Now, we must consider the situation for the join of all four relations. There are two general ways we can compute the join of all four:

1. Pick three to join in the best possible way, and then join in the fourth.

2. Divide the four relations into two pairs of two, join the pairs and then join the results.

Of course, if we consider only left-deep trees then the second type of plan is excluded, because it yields bushy trees. The table of Fig. 16.35 summarizes the seven possible ways to group the joins, based on the preferred groupings from Figs. 16.33 and 16.34.

For instance, consider the first expression in Fig. 16.35. It represents joining R, S, and T first, and then joining that result with U. From Fig. 16.34, we know that the best way to join R, S, and T is to join S and T first. We have used the left-deep form of this expression, and joined U on the right to continue

Grouping	Cost
$((S \bowtie T) \bowtie R) \bowtie U$	12,000
$((R \bowtie S) \bowtie U) \bowtie T$	55,000
$((T \bowtie U) \bowtie R) \bowtie S$	11,000
$((T \bowtie U) \bowtie S) \bowtie R$	3,000
$(T \bowtie U) \bowtie (R \bowtie S)$	6,000
$(R \bowtie T) \bowtie (S \bowtie U)$	2,000,000
$(S \bowtie T) \bowtie (R \bowtie U)$	12,000

Figure 16.35: Join groupings and their costs

the left-deep form. If we consider only left-deep trees, then this expression and relation order is the only option. If we allowed bushy trees, we would join U on the left, since it is smaller than the join of the other three. The cost of this join is 12,000, which is the sum of the cost and size of $(S \bowtie T) \bowtie R$, which are 2000 and 10,000, respectively.

The last three expressions in Fig. 16.35 represent additional options if we include bushy trees. These are formed by joining relations first in two pairs. For example, the last line represents the strategy of joining $R \bowtie U$ and $S \bowtie T$, and then joining the result. The cost of this expression is the sum of the sizes and costs of the two pairs. The costs are 0, as must be the case for any pair, and the sizes are 10,000 and 2000, respectively. Since we generally select the smaller relation to be the left argument, we show the expression as $(S \bowtie T) \bowtie (R \bowtie U)$.

In this example, we see that the least of all costs is associated with the fourth expression: $((T \bowtie U) \bowtie S) \bowtie R$. This expression is the one we select for computing the join; its cost is 3000. Since it is a left-deep tree, it is the selected logical query plan regardless of whether our dynamic-programming strategy considers all plans or just left-deep plans. □

16.6.5 Dynamic Programming With More Detailed Cost Functions

Using relation sizes as the cost estimate simplifies the calculations in a dynamic-programming algorithm. However, a disadvantage of this simplification is that it does not involve the actual costs of the joins in the calculation. As an extreme example, if one possible join $R(a,b) \bowtie S(b,c)$ involves a relation R with one tuple and another relation S that has an index on the join attribute b, then the join takes almost no time. On the other hand, if S has no index, then we must scan it, taking $B(S)$ disk I/O's, even when R is a singleton. A cost measure that only involved the sizes of R, S, and $R \bowtie S$ cannot distinguish these two cases, so the cost of using $R \bowtie S$ in the grouping will be either overestimated or underestimated.

However, modifying the dynamic programming algorithm to take join algorithms into account is not hard. First, the cost measure we use becomes disk

I/O's. When computing the cost of $\mathcal{R}_1 \bowtie \mathcal{R}_2$, we sum the cost of $\mathcal{R}_1$, the cost of $\mathcal{R}_2$, and the least cost of joining these two relations using the best available algorithm. Since the latter cost usually depends on the sizes of $\mathcal{R}_1$ and $\mathcal{R}_2$, we must also compute estimates for these sizes as we did in Example 16.33.

An even more powerful version of dynamic programming is based on the Selinger-style optimization mentioned in Section 16.5.4. Now, for each set of relations that might be joined, we keep not only one cost, but several costs. Recall that Selinger-style optimization considers not only the least cost of producing the result of the join, but also the least cost of producing that relation sorted in any of a number of "interesting" orders. These interesting sorts include any that might be used to advantage in a later sort-join or that could be used to produce the output of the entire query in the sorted order desired by the user. When sorted relations must be produced, the use of sort-join, either one-pass or multipass, must be considered as an option, while without considering the value of sorting a result, hash-joins are always at least as good as the corresponding sort-join.

16.6.6 A Greedy Algorithm for Selecting a Join Order

As Example 16.33 suggests, even the carefully limited search of dynamic programming leads to a number of calculations that is exponential in the number of relations joined. It is reasonable to use an exhaustive method like dynamic programming or branch-and-bound search to find optimal join orders of five or six relations. However, when the number of joins grows beyond that, or if we choose not to invest the time necessary for an exhaustive search, then we can use a join-order heuristic in our query optimizer.

The most common choice of heuristic is a *greedy* algorithm, where we make one decision at a time about the order of joins and never backtrack or reconsider decisions once made. We shall consider a greedy algorithm that only selects a left-deep tree. The "greediness" is based on the idea that we want to keep the intermediate relations as small as possible at each level of the tree.

BASIS: Start with the pair of relations whose estimated join size is smallest. The join of these relations becomes the *current tree*.

INDUCTION: Find, among all those relations not yet included in the current tree, the relation that, when joined with the current tree, yields the relation of smallest estimated size. The new current tree has the old current tree as its left argument and the selected relation as its right argument.

Example 16.34: Let us apply the greedy algorithm to the relations of Example 16.33. The basis step is to find the pair of relations that have the smallest join. Consulting Fig. 16.33, we see that this honor goes to the join $T \bowtie U$, with a cost of 1000. Thus, $T \bowtie U$ is the "current tree."

We now consider whether to join R or S into the tree next. Thus we compare the sizes of $(T \bowtie U) \bowtie R$ and $(T \bowtie U) \bowtie S$. Figure 16.34 tells us that the

Join Selectivity

A useful way to view heuristics such as the greedy algorithm for selecting a left-deep join tree is that each relation R, when joined with the current tree, has a *selectivity*, which is the ratio of the size of the join result to size of the current tree's result. Since we usually do not have the exact sizes of either relation, we estimate these sizes as we have done previously. A greedy approach to join ordering is to pick that relation with the smallest selectivity.

For example, if a join attribute is a key for R, then the selectivity is at most 1, which is usually a favorable situation. Notice that, judging from the statistics of Fig. 16.31, attribute d is a key for U, and there are no keys for other relations, which suggests why joining T with U is the best way to start the join.

latter, with a size of 2000 is better than the former, with a size of 10,000. Thus, we pick as the new current tree $(T \bowtie U) \bowtie S$.

Now there is no choice; we must join R at the last step, leaving us with a total cost of 3000, the sum of the sizes of the two intermediate relations. Note that the tree resulting from the greedy algorithm is the same as that selected by the dynamic-programming algorithm in Example 16.33. However, there are examples where the greedy algorithm fails to find the best solution, while the dynamic-programming algorithm guarantees to find the best; see Exercise 16.6.4. □

16.6.7 Exercises for Section 16.6

Exercise 16.6.1: For the relations of Exercise 16.4.1, give the dynamic-programming table entries that evaluates all possible join orders allowing: a) All trees b) Left-deep trees only. What is the best choice in each case?

Exercise 16.6.2: Repeat Exercise 16.6.1 with the following modifications:

i. The schema for Z is changed to $Z(d, a)$.

ii. $V(Z, a) = 100$.

Exercise 16.6.3: Repeat Exercise 16.6.1 with the relations of Exercise 16.4.2.

Exercise 16.6.4: Consider the join of relations $R(a, b)$, $S(b, c)$, $T(c, d)$, and $U(a, d)$, where R and U each have 1000 tuples, while S and T each have 100 tuples. Further, there are 100 values of all attributes of all relations, except for attribute c, where $V(S, c) = V(T, c) = 10$.

a) What is the order selected by the greedy algorithm? What is its cost?

b) What is the optimum join ordering and its cost?

Exercise 16.6.5: How many trees are there for the join of (a) seven (b) eight relations? How many of these are neither left-deep nor right-deep?

! **Exercise 16.6.6:** Suppose we wish to join the relations R, S, T, and U in one of the tree structures of Fig. 16.30, and we want to keep all intermediate relations in memory until they are no longer needed. Following our usual assumption, the result of the join of all four will be consumed by some other process as it is generated, so no memory is needed for that relation. In terms of the number of blocks required for the stored relations and the intermediate relations [e.g., $B(R)$ or $B(R \bowtie S)$], give a lower bound on M, the number of blocks of memory needed, for each of the trees in Fig. 16.30? What assumptions let us conclude that one tree is certain to use less memory than another?

! **Exercise 16.6.7:** If we use dynamic programming to select an order for the join of k relations, how many entries of the table do we have to fill?

16.7 Completing the Physical-Query-Plan

We have parsed the query, converted it to an initial logical query plan, and improved that logical query plan with transformations described in Section 16.3. Part of the process of selecting the physical query plan is enumeration and cost-estimation for all of our options, which we discussed in Section 16.5. Section 16.6 focused on the question of enumeration, cost estimation, and ordering for joins of several relations. By extension, we can use similar techniques to order groups of unions, intersections, or any associative/commutative operation.

There are still several steps needed to turn the logical plan into a complete physical query plan. The principal issues that we must yet cover are:

1. Selection of algorithms to implement the operations of the query plan, when algorithm-selection was not done as part of some earlier step such as selection of a join order by dynamic programming.

2. Decisions regarding when intermediate results will be *materialized* (created whole and stored on disk), and when they will be *pipelined* (created only in main memory, and not necessarily kept in their entirety at any one time).

3. Notation for physical-query-plan operators, which must include details regarding access methods for stored relations and algorithms for implementation of relational-algebra operators.

We shall not discuss the subject of selection of algorithms for operators in its entirety. Rather, we sample the issues by discussing two of the most important operators: selection in Section 16.7.1 and joins in Section 16.7.2.

Then, we consider the choice between pipelining and materialization in Sections 16.7.3 through 16.7.5. A notation for physical query plans is presented in Section 16.7.6.

16.7.1 Choosing a Selection Method

One of the important steps in choosing a physical query plan is to pick algorithms for each selection operator. In Section 15.2.1 we mentioned the obvious implementation of a $\sigma_C(R)$ operator, where we access the entire relation R and see which tuples satisfy condition C. Then in Section 15.6.2 we considered the possibility that C was of the form "attribute equals constant," and we had an index on that attribute. If so, then we can find the tuples that satisfy condition C without looking at all of R. Now, let us consider the generalization of this problem, where we have a selection condition that is the AND of several conditions. Assume at least one condition is of the form $A\theta c$, where A is an attribute with an index, c is a constant, and θ is a comparison operator such as $=$ or $<$.

Each physical plan uses some number of attributes that each:

a) Have an index, and

b) Are compared to a constant in one of the terms of the selection.

We then use these indexes to identify the sets of tuples that satisfy each of the conditions. Sections 14.1.7 and 14.4.3 discussed how we could use pointers obtained from these indexes to find only the tuples that satisfied all the conditions before we read these tuples from disk.

For simplicity, we shall not consider the use of several indexes in this way. Rather, we limit our discussion to physical plans that:

1. Retrieve all tuples that satisfy a comparison for which an index exists, using the index-scan physical operator discussed in Section 15.1.1.

2. Consider each tuple selected in (1) to decide whether it satisfies the rest of the selection condition. The physical operator that performs this step is callled `Filter`.

In addition to physical plans of this form, we must also consider the plan that uses no index but reads the entire relation (using the table-scan physical operator) and passes each tuple to the `Filter` operator to check for satisfaction of the selection condition.

We decide among the possible physical plans for a selection by estimating the cost of reading data with each plan. To compare costs of alternative plans we cannot continue using the simplified cost estimate of intermediate-relation size. The reason is that we are now considering implementations of a single step of the logical query plan, and intermediate relations are independent of implementation.

Thus, we shall refocus our attention and resume counting disk I/O's, as we did when we discussed algorithms and their costs in Chapter 15. To simplify as before, we shall count only the cost of accessing the data blocks, not the index blocks. Recall that the number of index blocks needed is generally much smaller than the number of data blocks needed, so this approximation to disk I/O cost is usually accurate enough.

The following is an outline of how costs for the various plans are estimated. We assume that the operation is $\sigma_C(R)$, where condition C is the AND of one or more terms.

1. The cost of the table-scan algorithm coupled with a filter step is:
   (a) $B(R)$ if R is clustered, and
   (b) $T(R)$ if R is not clustered.
2. The cost of a plan that picks an equality term such as $a = 10$ for which an index on attribute a exists, uses index-scan to find the matching tuples, and then filters the retrieved tuples to see if they satisfy the full condition C is:
   (a) $B(R)/V(R,a)$ if the index is clustering, and
   (b) $T(R)/V(R,a)$ if the index is not clustering.
3. The cost of a plan that picks an inequality term such as $b < 20$ for which an index on attribute b exists, uses index-scan to retrieve the matching tuples, and then filters the retrieved tuples to see if they satisfy the full condition C is:
   (a) $B(R)/3$ if the index is clustering,[9] and
   (b) $T(R)/3$ if the index is not clustering.

Example 16.35: Consider selection $\sigma_{x=1 \text{ AND } y=2 \text{ AND } z<5}(R)$, where $R(x,y,z)$ has the following parameters: $T(R) = 5000$, $B(R) = 200$, $V(R,x) = 100$, and $V(R,y) = 500$. Further, suppose R is clustered, and there are indexes on all of x, y, and z, but only the index on z is clustering. The following are the options for implementing this selection:

1. Table-scan followed by filter. The cost is $B(R)$, or 200 disk I/O's, since R is clustered.
2. Use the index on x and the index-scan operator to find those tuples with $x = 1$, then use the filter operator to check that $y = 2$ and $z < 5$. Since there are about $T(R)/V(R,x) = 50$ tuples with $x = 1$, and the index is not clustering, we require about 50 disk I/O's.

[9] Recall that we assume the typical inequality retrieves only 1/3 the tuples, for reasons discussed in Section 16.4.3.

3. Use the index on y and index-scan to find those tuples with $y = 2$, then filter these tuples to see that $x = 1$ and $z < 5$. The cost for using this nonclustering index is about $T(R)/V(R, y)$, or 10 disk I/O's.

4. Use the clustering index on z and index-scan to find those tuples with $z < 5$, then filter these tuples to see that $x = 1$ and $y = 2$. The number of disk I/O's is about $B(R)/3 = 67$.

We see that the least cost plan is the third, with an estimated cost of 10 disk I/O's. Thus, the preferred physical plan for this selection retrieves all tuples with $y = 2$ and then filters for the other two conditions. □

16.7.2 Choosing a Join Method

We saw in Chapter 15 the costs associated with the various join algorithms. On the assumption that we know (or can estimate) how many buffers are available to perform the join, we can apply the formulas in Section 15.4.9 for sort-joins, Section 15.5.7 for hash-joins, and Sections 15.6.3 and 15.6.4 for indexed joins.

However, if we are not sure of, or cannot know, the number of buffers that will be available during the execution of this query (because we do not know what else the DBMS is doing at the same time), or if we do not have estimates of important size parameters such as the $V(R, a)$'s, then there are still some principles we can apply to choosing a join method. Similar ideas apply to other binary operations such as unions, and to the full-relation, unary operators, γ and δ.

- One approach is to call for the one-pass join, hoping that the buffer manager can devote enough buffers to the join, or that the buffer manager can come close, so thrashing is not a major cost. An alternative (for joins only, not for other binary operators) is to choose a nested-loop join, hoping that if the left argument cannot be granted enough buffers to fit in memory at once, then that argument will not have to be divided into too many pieces, and the resulting join will still be reasonably efficient.

- A sort-join is a good choice when either:

  1. One or both arguments are already sorted on their join attribute(s), or
  2. There are two or more joins on the same attribute, such as

     $$\big(R(a, b) \bowtie S(a, c)\big) \bowtie T(a, d)$$

     where sorting R and S on a will cause the result of $R \bowtie S$ to be sorted on a and used directly in a second sort-join.

- If there is an index opportunity such as a join $R(a, b) \bowtie S(b, c)$, where R is expected to be small (perhaps the result of a selection on a key that must yield only one tuple), and there is an index on the join attribute $S.b$, then we should choose an index-join.

- If there is no opportunity to use already-sorted relations or indexes, and a multipass join is needed, then hashing is probably the best choice, because the number of passes it requires depends on the size of the smaller argument rather than on both arguments.

16.7.3 Pipelining Versus Materialization

The last major issue we shall discuss in connection with choice of a physical query plan is pipelining of results. The naive way to execute a query plan is to order the operations appropriately (so an operation is not performed until the argument(s) below it have been performed), and store the result of each operation on disk until it is needed by another operation. This strategy is called ***materialization***, since each intermediate relation is materialized on disk.

A more subtle, and generally more efficient, way to execute a query plan is to interleave the execution of several operations. The tuples produced by one operation are passed directly to the operation that uses it, without ever storing the intermediate tuples on disk. This approach is called ***pipelining***, and it typically is implemented by a network of iterators (see Section 15.1.6), whose methods call each other at appropriate times. Since it saves disk I/O's, there is an obvious advantage to pipelining, but there is a corresponding disadvantage. Since several operations must share main memory at any time, there is a chance that algorithms with higher disk-I/O requirements must be chosen, or thrashing will occur, thus giving back all the disk-I/O savings that were gained by pipelining, and possibly more.

16.7.4 Pipelining Unary Operations

Unary operations — selection and projection — are excellent candidates for pipelining. Since these operations are tuple-at-a-time, we never need to have more than one block for input, and one block for the output. This mode of operation was suggested by Fig. 15.5.

We may implement a pipelined unary operation by iterators, as discussed in Section 15.1.6. The consumer of the pipelined result calls `GetNext()` each time another tuple is needed. In the case of a projection, it is only necessary to call `GetNext()` once on the source of tuples, project that tuple appropriately, and return the result to the consumer. For a selection σ_C (technically, the physical operator `Filter(`C`)`), it may be necessary to call `GetNext()` several times at the source, until one tuple that satisfies condition C is found. Figure 16.36 illustrates this process.

16.7.5 Pipelining Binary Operations

The results of binary operations can also be pipelined. We use one buffer to pass the result to its consumer, one block at a time. However, the number of other buffers needed to compute the result and to consume the result varies,

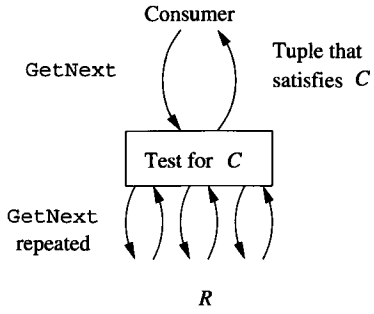

Figure 16.36: Execution of a pipelined selection using iterators

Materialization in Memory

One might imagine that there is an intermediate approach, between pipelining and materialization, where the entire result of one operation is stored in main-memory buffers (not on disk) before being passed to the consuming operation. We regard this possible mode of operation as pipelining, where the first thing that the consuming operation does is organize the entire relation, or a large portion of it, in memory. An example of this sort of behavior is a selection whose result becomes the left (build) argument to one of several join algorithms, including the simple one-pass join, multipass hash-join, or sort-join.

depending on the size of the result and the sizes of the arguments. We shall use an extended example to illustrate the tradeoffs and opportunities.

Example 16.36: Let us consider physical query plans for the expression

$$\big(R(w,x) \bowtie S(x,y)\big) \bowtie U(y,z)$$

We make the following assumptions:

1. R occupies 5000 blocks; S and U each occupy 10,000 blocks.
2. The intermediate result $R \bowtie S$ occupies k blocks for some k.
3. Both joins will be implemented as hash-joins, either one-pass or two-pass, depending on k.
4. There are 101 buffers available. This number, as usual, is set artificially low.

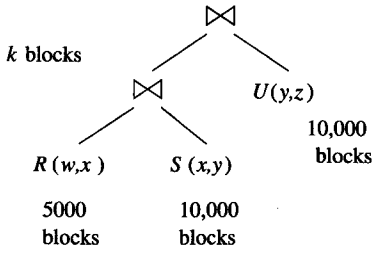

Figure 16.37: Logical query plan and parameters for Example 16.36

A sketch of the expression with key parameters is in Fig. 16.37.

First, consider the join $R \bowtie S$. Neither relation fits in main memory, so we need a two-pass hash-join. If the smaller relation R is partitioned into the maximum-possible 100 buckets on the first pass, then each bucket for R occupies 50 blocks.[10] If R's buckets have 50 blocks, then the second pass of the hash-join $R \bowtie S$ uses 51 buffers, leaving 50 buffers to use for the join of the result of $R \bowtie S$ with U.

Now, suppose that $k \leq 49$; that is, the result of $R \bowtie S$ occupies at most 49 blocks. Then we can pipeline the result of $R \bowtie S$ into 49 buffers, organize them for lookup as a hash table, and we have one buffer left to read each block of U in turn. We may thus execute the second join as a one-pass join. The total number of disk I/O's is:

a) 45,000 to perform the two-pass hash join of R and S.

b) 10,000 to read U in the one-pass hash-join of $(R \bowtie S) \bowtie U$.

The total is 55,000 disk I/O's.

Now, suppose $k > 49$, but $k \leq 5000$. We can still pipeline the result of $R \bowtie S$, but we need to use another strategy, in which this relation is joined with U in a 50-bucket, two-pass hash-join.

1. Before we start on $R \bowtie S$, we hash U into 50 buckets of 200 blocks each.
2. Next, we perform a two-pass hash join of R and S using 51 buckets as before, but as each tuple of $R \bowtie S$ is generated, we place it in one of the 50 remaining buffers that is used to help form the 50 buckets for the join of $R \bowtie S$ with U. These buffers are written to disk when they get full, as is normal for a two-pass hash-join.
3. Finally, we join $R \bowtie S$ with U bucket by bucket. Since $k \leq 5000$, the buckets of $R \bowtie S$ will be of size at most 100 blocks, so this join is feasible. The fact that buckets of U are of size 200 blocks is not a problem, since

[10] We shall assume for convenience that all buckets wind up with exactly their fair share of tuples.

we are using buckets of $R \bowtie S$ as the build relation and buckets of U as the probe relation in the one-pass joins of buckets.

The number of disk I/O's for this pipelined join is:

a) 20,000 to read U and write its tuples into buckets.

b) 45,000 to perform the two-pass hash-join $R \bowtie S$.

c) k to write out the buckets of $R \bowtie S$.

d) k + 10,000 to read the buckets of $R \bowtie S$ and U in the final join.

The total cost is thus 75,000 + $2k$. Note that there is an apparent discontinuity as k grows from 49 to 50, since we had to change the final join from one-pass to two-pass. In practice, the cost would not change so precipitously, since we could use the one-pass join even if there were not enough buffers and a small amount of thrashing occurred.

Last, let us consider what happens when $k > 5000$. Now, we cannot perform a two-pass join in the 50 buffers available if the result of $R \bowtie S$ is pipelined. We could use a three-pass join, but that would require an extra 2 disk I/O's per block of either argument, or 20,000 + $2k$ more disk I/O's. We can do better if we instead decline to pipeline $R \bowtie S$. Now, an outline of the computation of the joins is:

1. Compute $R \bowtie S$ using a two-pass hash join and store the result on disk.

2. Join $R \bowtie S$ with U, also using a two-pass hash-join. Note that since $B(U)$ = 10,000, we can perform a two-pass hash-join using 100 buckets, regardless of how large k is. Technically, U should appear as the left argument of its join in Fig. 16.37 if we decide to make U the build relation for the hash join.

The number of disk I/O's for this plan is:

a) 45,000 for the two-pass join of R and S.

b) k to store $R \bowtie S$ on disk.

c) 30,000 + $3k$ for the two-pass hash-join of U with $R \bowtie S$.

The total cost is thus 75,000 + $4k$, which is less than the cost of going to a three-pass join at the final step. The three complete plans are summarized in the table of Fig. 16.38. □

Range of k	Pipeline or Materialize	Algorithm for final join	Total Disk I/O's
$k \leq 49$	Pipeline	one-pass	55,000
$50 \leq k \leq 5000$	Pipeline	50-bucket, two-pass	$75{,}000 + 2k$
$5000 < k$	Materialize	100-bucket, two-pass	$75{,}000 + 4k$

Figure 16.38: Costs of physical plans as a function of the size of $R \bowtie S$

16.7.6 Notation for Physical Query Plans

We have seen many examples of the operators that can be used to form a physical query plan. In general, each operator of the logical plan becomes one or more operators of the physical plan, and leaves (stored relations) of the logical plan become, in the physical plan, one of the scan operators applied to that relation. In addition, materialization would be indicated by a `Store` operator applied to the intermediate result that is to be materialized, followed by a suitable scan operator (usually `TableScan`, since there is no index on the intermediate relation unless one is constructed explicitly) when the materialized result is accessed by its consumer. However, for simplicity, in our physical-query-plan trees we shall indicate that a certain intermediate relation is materialized by a double line crossing the edge between that relation and its consumer. All other edges are assumed to represent pipelining between the supplier and consumer of tuples.

We shall now catalog the various operators that are typically found in physical query plans. Unlike the relational algebra, whose notation is fairly standard, each DBMS will use its own internal notation for physical query plans.

Operators for Leaves

Each relation R that is a leaf operand of the logical-query-plan tree will be replaced by a scan operator. The options are:

1. `TableScan(R)`: All blocks holding tuples of R are read in arbitrary order.
2. `SortScan(R,L)`: Tuples of R are read in order, sorted according to the attribute(s) on list L.
3. `IndexScan(R,C)`: Here, C is a condition of the form $A\theta c$, where A is an attribute of R, θ is a comparison such as $=$ or $<$, and c is a constant. Tuples of R are accessed through an index on attribute A. If the comparison θ is not $=$, then the index must be one, such as a B-tree, that supports range queries.
4. `IndexScan(R,A)`: Here A is an attribute of R. The entire relation R is retrieved via an index on $R.A$. This operator behaves like `TableScan`,

but may be more efficient if R is not clustered.

Physical Operators for Selection

A logical operator $\sigma_C(R)$ is often combined, or partially combined, with the access method for relation R, when R is a stored relation. Other selections, where the argument is not a stored relation or an appropriate index is not available, will be replaced by the corresponding physical operator we have called `Filter`. Recall the strategy for choosing a selection implementation, which we discussed in Section 16.7.1. The notation we shall use for the various selection implementations are:

1. We may simply replace $\sigma_C(R)$ by the operator `Filter(C)`. This choice makes sense if there is no index on R, or no index on an attribute that condition C mentions. If R, the argument of the selection, is actually an intermediate relation being pipelined to the selection, then no other operator besides `Filter` is needed. If R is a stored or materialized relation, then we must use an operator, `TableScan` or `SortScan(R,L)`, to access R. We prefer sort-scan if the result of $\sigma_C(R)$ will later be passed to an operator that requires its argument sorted.

2. If condition C can be expressed as $A\theta c$ `AND` D for some other condition D, and there is an index on $R.A$, then we may:

   (a) Use the operator `IndexScan(R,`$A\theta$`c)` to access R, and

   (b) Use `Filter(D)` in place of the selection $\sigma_C(R)$.

Physical Sort Operators

Sorting of a relation can occur at any point in the physical query plan. We have already introduced the `SortScan(R,L)` operator, which reads a stored relation R and produces it sorted according to the list of attributes L. When we apply a sort-based algorithm for operations such as join or grouping, there is an initial phase in which we sort the argument according to some list of attributes. It is common to use an explicit physical operator `Sort(L)` to perform this sort on an operand relation that is not stored. This operator can also be used at the top of the physical-query-plan tree if the result needs to be sorted because of an `ORDER BY` clause in the original query, thus playing the same role as the τ operator of Section 5.2.6.

Other Relational-Algebra Operations

All other operations are replaced by a suitable physical operator. These operators can be given designations that indicate:

1. The operation being performed, e.g., join or grouping.

2. Necessary parameters, e.g., the condition in a theta-join or the list of elements in a grouping.

3. A general strategy for the algorithm: sort-based, hash-based, or index-based, e.g.

4. A decision about the number of passes to be used: one-pass, two-pass, or multipass (recursive, using as many passes as necessary for the data at hand). Alternatively, this choice may be left until run-time.

5. An anticipated number of buffers the operation will require.

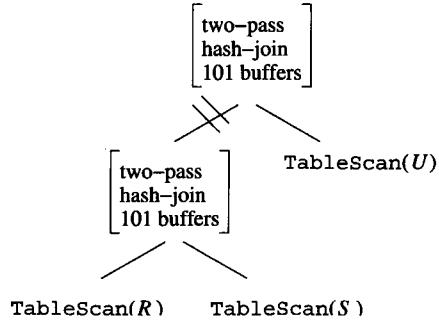

Figure 16.39: A physical plan from Example 16.36

Example 16.37: Figure 16.39 shows the physical plan developed in Example 16.36 for the case $k > 5000$. In this plan, we access each of the three relations by a table-scan. We use a two-pass hash-join for the first join, materialize it, and use a two-pass hash-join for the second join. By implication of the double-line symbol for materialization, the left argument of the top join is also obtained by a table-scan, and the result of the first join is stored using the `Store` operator.

In contrast, if $k \leq 49$, then the physical plan developed in Example 16.36 is that shown in Fig. 16.40. Notice that the second join uses a different number of passes, a different number of buffers, and a left argument that is pipelined, not materialized. □

Example 16.38: Consider the selection operation in Example 16.35, where we decided that the best of options was to use the index on y to find those tuples with $y = 2$, then check these tuples for the other conditions $x = 1$ and $z < 5$. Figure 16.41 shows the physical query plan. The leaf indicates that R will be accessed through its index on y, retrieving only those tuples with $y = 2$. The filter operator says that we complete the selection by further selecting those of the retrieved tuples that have both $x = 1$ and $z < 5$. □

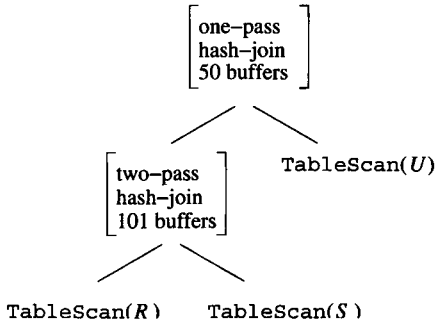

Figure 16.40: Another physical plan for the case where $R \bowtie S$ is expected to be very small

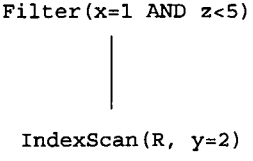

Figure 16.41: Annotating a selection to use the most appropriate index

16.7.7 Ordering of Physical Operations

Our final topic regarding physical query plans is the matter of order of operations. The physical query plan is generally represented as a tree, and trees imply something about order of operations, since data must flow up the tree. However, since bushy trees may have interior nodes that are neither ancestors nor descendants of one another, the order of evaluation of interior nodes may not always be clear. Moreover, since iterators can be used to implement operations in a pipelined manner, it is possible that the times of execution for various nodes overlap, and the notion of "ordering" nodes makes no sense.

If materialization is implemented in the obvious store-and-later-retrieve way, and pipelining is implemented by iterators, then we may establish a fixed sequence of events whereby each operation of a physical query plan is executed. The following rules summarize the ordering of events implicit in a physical-query-plan tree:

1. Break the tree into subtrees at each edge that represents materialization. The subtrees will be executed one-at-a-time.

2. Order the execution of the subtrees in a bottom-up, left-to-right manner. To be precise, perform a preorder traversal of the entire tree. Order the subtrees in the order in which the preorder traversal exits from the subtrees.

3. Execute all nodes of each subtree using a network of iterators. Thus, all the nodes in one subtree are executed simultaneously, with `GetNext` calls among their operators determining the exact order of events.

Following this strategy, the query optimizer can now generate executable code, perhaps a sequence of function calls, for the query.

16.7.8 Exercises for Section 16.7

Exercise 16.7.1: Consider a relation $R(a, b, c, d)$ that has a clustering index on a and nonclustering indexes on each of the other attributes. The relevant parameters are: $B(R) = 1000$, $T(R) = 5000$, $V(R, a) = 20$, $V(R, b) = 1000$, $V(R, c) = 5000$, and $V(R, d) = 500$. Give the best query plan (index-scan or table-scan followed by a filter step) and the disk-I/O cost for each of the following selections:

a) $\sigma_{a=1 \text{ AND } b=2 \text{ AND } d=3}(R)$.

b) $\sigma_{a=1 \text{ AND } b=2 \text{ AND } c\geq 3}(R)$.

c) $\sigma_{a=1 \text{ AND } b\leq 2 \text{ AND } c\geq 3}(R)$.

! **Exercise 16.7.2:** In terms of $B(R)$, $T(R)$, $V(R, x)$, and $V(R, y)$, express the following conditions about the cost of implementing a selection on R:

a) It is better to use index-scan with a nonclustering index on x and a term that equates x to a constant than a nonclustering index on y and a term that equates y to a constant.

b) It is better to use index-scan with a nonclustering index on x and a term that equates x to a constant than a clustering index on y and a term that equates y to a constant.

c) It is better to use index-scan with a nonclustering index on x and a term that equates x to a constant than a clustering index on y and a term of the form $y > C$ for some constant C.

Exercise 16.7.3: How would the conclusions about when to pipeline in Example 16.36 change if the size of relation R were not 5000 blocks, but: (a) 2000 blocks ! (b) 10,000 blocks ! (c) 100 blocks?

! **Exercise 16.7.4:** Suppose we want to compute $\big(R(a, b) \bowtie S(a, c)\big) \bowtie T(a, d)$ in the order indicated. We have $M = 101$ main-memory buffers, and $B(R) = B(S) = 2000$. Because the join attribute a is the same for both joins, we decide to implement the first join $R \bowtie S$ by a two-pass sort-join, and we shall use the appropriate number of passes for the second join, first dividing T into some number of sublists sorted on a, and merging them with the sorted and pipelined stream of tuples from the join $R \bowtie S$. For what values of $B(T)$ should we choose for the join of T with $R \bowtie S$:

a) A one-pass join; i.e., we read T into memory, and compare its tuples with the tuples of $R \bowtie S$ as they are generated.

b) A two-pass join; i.e., we create sorted sublists for T and keep one buffer in memory for each sorted sublist, while we generate tuples of $R \bowtie S$.

16.8 Summary of Chapter 16

- ✦ *Compilation of Queries*: Compilation turns a query into a physical query plan, which is a sequence of operations that can be implemented by the query-execution engine. The principal steps of query compilation are parsing, semantic checking, selection of the preferred logical query plan (algebraic expression), and generation from that of the best physical plan.

- ✦ *The Parser*: The first step in processing a SQL query is to parse it, as one would for code in any programming language. The result of parsing is a parse tree with nodes corresponding to SQL constructs.

- ✦ *View Expansion*: Queries that refer to virtual views must have these references in the parse tree replaced by the tree for the expression that defines the view. This expansion often introduces several opportunities to optimize the complete query.

- ✦ *Semantic Checking*: A preprocessor examines the parse tree, checks that the attributes, relation names, and types make sense, and resolves attribute references.

- ✦ *Conversion to a Logical Query Plan*: The query processor must convert the semantically checked parse tree to an algebraic expression. Much of the conversion to relational algebra is straightforward, but subqueries present a problem. One approach is to introduce a two-argument selection that puts the subquery in the condition of the selection, and then apply appropriate transformations for the common special cases.

- ✦ *Algebraic Transformations*: There are many ways that a logical query plan can be transformed to a better plan by using algebraic transformations. Section 16.2 enumerates the principal ones.

- ✦ *Choosing a Logical Query Plan*: The query processor must select that query plan that is most likely to lead to an efficient physical plan. In addition to applying algebraic transformations, it is useful to group associative and commutative operators, especially joins, so the physical query plan can choose the best order and grouping for these operations.

- ✦ *Estimating Sizes of Relations*: When selecting the best logical plan, or when ordering joins or other associative-commutative operations, we use the estimated size of intermediate relations as a surrogate for the true

running time. Knowing, or estimating, both the size (number of tuples) of relations and the number of distinct values for each attribute of each relation helps us get good estimates of the sizes of intermediate relations.

- *Histograms*: Some systems keep histograms of the values for a given attribute. This information can be used to obtain better estimates of intermediate-relation sizes than the simple methods stressed here.

- *Cost-Based Optimization*: When selecting the best physical plan, we need to estimate the cost of each possible plan. Various strategies are used to generate all or some of the possible physical plans that implement a given logical plan.

- *Plan-Enumeration Strategies*: The common approaches to searching the space of physical plans for the best include dynamic programming (tabularizing the best plan for each subexpression of the given logical plan), Selinger-style dynamic programming (which includes the sort-order of results as part of the table, giving best plans for each sort-order and for an unsorted result), greedy approaches (making a series of locally optimal decisions, given the choices for the physical plan that have been made so far), and branch-and-bound (enumerating only plans that are not immediately known to be worse than the best plan found so far).

- *Left-Deep Join Trees*: When picking a grouping and order for the join of several relations, it is common to restrict the search to left-deep trees, which are binary trees with a single spine down the left edge, with only leaves as right children. This form of join expression tends to yield efficient plans and also limits significantly the number of physical plans that need to be considered.

- *Physical Plans for Selection*: If possible, a selection should be broken into an index-scan of the relation to which the selection is applied (typically using a condition in which the indexed attribute is equated to a constant), followed by a filter operation. The filter examines the tuples retrieved by the index-scan and passes through only those that meet the portions of the selection condition other than that on which the index scan is based.

- *Pipelining Versus Materialization*: Ideally, the result of each physical operator is consumed by another operator, with the result being passed between the two in main memory ("pipelining"), perhaps using an iterator to control the flow of data from one to the other. However, sometimes there is an advantage to storing ("materializing") the result of one operator to save space in main memory for other operators. Thus, the physical-query-plan generator should consider both pipelining and materialization of intermediates.

16.9 References for Chapter 16

The surveys mentioned in the bibliographic notes to Chapter 15 also contain material relevant to query compilation. In addition, we recommend the survey [1], which contains material on the query optimizers of commercial systems.

Three of the earliest studies of query optimization are [4], [5], and [3]. Paper [6], another early study, incorporates the idea of pushing selections down the tree with the greedy algorithm for join-order choice. [2] is the source for "Selinger-style optimization" as well as describing the System R optimizer, which was one of the most ambitious attempts at query optimization of its day.

1. G. Graefe (ed.), *Data Engineering* **16**:4 (1993), special issue on query processing in commercial database management systems, IEEE.

2. P. Griffiths-Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database system," *Proc. ACM SIGMOD Intl. Conf. on Management of Data* (1979), pp. 23–34.

3. P. A. V. Hall, "Optimization of a single relational expression in a relational database system," *IBM J. Research and Development* **20**:3 (1976), pp. 244–257.

4. F. P. Palermo, "A database search problem," in: J. T. Tou (ed.) *Information Systems COINS IV*, Plenum, New York, 1974.

5. J. M. Smith and P. Y. Chang, "Optimizing the performance of a relational algebra database interface," *Comm. ACM* **18**:10 (1975), pp. 568–579.

6. E. Wong and K. Youssefi, "Decomposition — a strategy for query processing," *ACM Trans. on Database Systems* **1**:3 (1976), pp. 223–241.