discu-

15.7.1 Notation for Query Trees and Query Graphs

A **query tree** is a tree data structure that corresponds to a relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and represents the relational algebra operations as internal nodes. An execution of the query tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.

Figure 15.4(a) shows a query tree (the same as shown in Figure 6.9) for query Q2 of Chapters 5 to 8: For every project located in 'Stafford', retrieve the project number, the controlling department number, and the department manager's last name, address, and birthdate. This query is specified on the relational schema of Figure 5.5 and corresponds to the following relational algebra expression:

$$\pi_{\text{Pnumber, Dnum, Lname, Address, Bdate}}(((\sigma_{\text{Plocation='Stafford'}}(\text{PROJECT})) \bowtie_{\text{Dnum=Dnumber}}(\text{DEPARTMENT})) \bowtie_{\text{Mgr\_ssn=Ssn}}(\text{EMPLOYEE}))$$

This corresponds to the following SQL query:

```
Q2:  SELECT   P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
     FROM     PROJECT AS P, DEPARTMENT AS D, EMPLOYEE AS E
     WHERE    P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND
              P.Plocation= 'Stafford';
```

In Figure 15.4(a) the three relations PROJECT, DEPARTMENT, and EMPLOYEE are represented by leaf nodes P, D, and E, while the relational algebra operations of the expression are represented by internal tree nodes. When this query tree is executed, the node marked (1) in Figure 15.4(a) must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin executing operation (2). Similarly, node (2) must begin executing and producing results before node (3) can start execution, and so on.

As we can see, the query tree represents a specific order of operations for executing a query. A more neutral representation of a query is the **query graph** notation. Figure 15.4(c) (the same as shown in Figure 6.13) shows the query graph for query Q2. Relations in the query are represented by **relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges**, as shown in Figure 15.4(c). Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

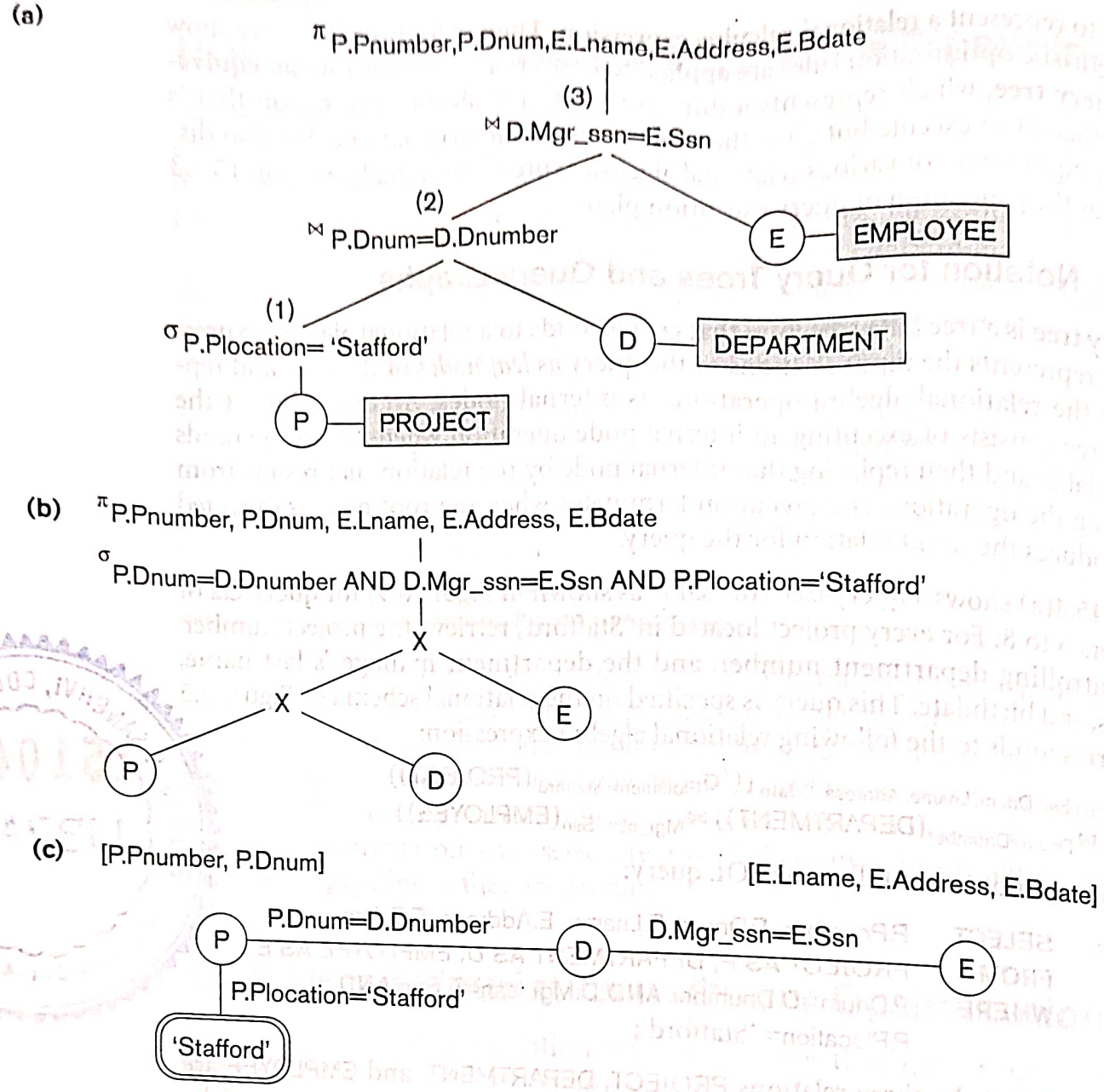


Figure 15.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

The query graph representation does not indicate an order on which operations to perform first. There is only a single graph corresponding to each query.[15] Although some optimization techniques were based on query graphs, it is now generally accepted that query trees are preferable because, in practice, the query optimizer needs to show the order of operations for query execution, which is not possible in query graphs.

15. Hence, a query graph corresponds to a *relational calculus* expression as shown in Section 6.6.5.

15.7.2 Heuristic Optimization of Query Trees

In general, many different relational algebra expressions—and hence many different query trees—can be equivalent; that is, they can correspond to the same query.[16]

The query parser will typically generate a standard **initial query tree** to correspond to an SQL query, without doing any optimization. For example, for a SELECT-PROJECT-JOIN query, such as Q2, the initial tree is shown in Figure 15.4(b). The CARTESIAN PRODUCT of the relations specified in the FROM clause is first applied; then the selection and join conditions of the WHERE clause are applied, followed by the projection on the SELECT clause attributes. Such a canonical query tree represents a relational algebra expression that is *very inefficient if executed directly,* because of the CARTESIAN PRODUCT (×) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each. However, the query tree in Figure 15.4(b) is in a simple standard form that can be easily created. It is now the job of the heuristic query optimizer to transform this initial query tree into a **final query tree** that is efficient to execute.

The optimizer must include rules for equivalence among relational algebra expressions that can be applied to the initial tree. The heuristic query optimization rules then utilize these equivalence expressions to transform the initial tree into the final, optimized query tree. First we discuss informally how a query tree is transformed by using heuristics. Then we discuss general transformation rules and show how they may be used in an algebraic heuristic optimizer.

Example of Transforming a Query. Consider the following query Q on the database of Figure 5.5: *Find the last names of employees born after 1957 who work on a project named 'Aquarius'.* This query can be specified in SQL as follows:

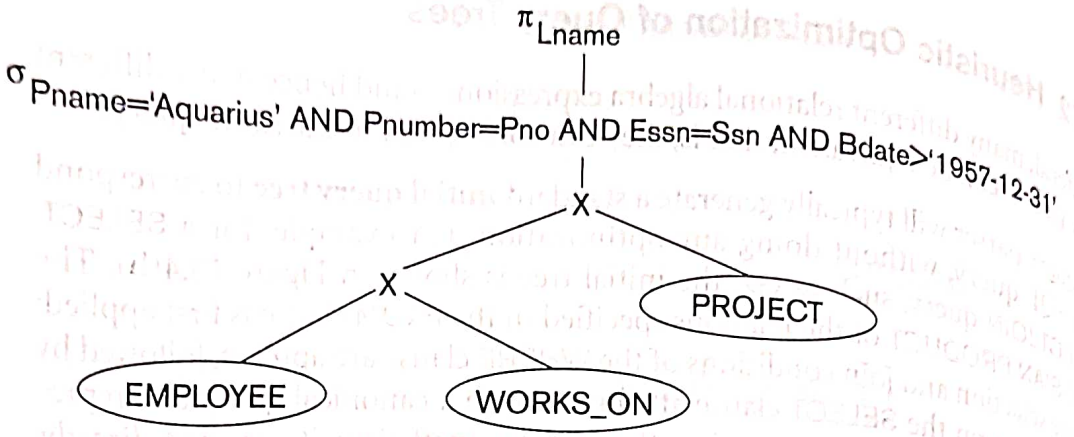
```
Q:  SELECT   Lname
    FROM     EMPLOYEE, WORKS_ON, PROJECT
    WHERE    Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn
             AND Bdate > '1957-12-31';
```

The initial query tree for Q is shown in Figure 15.5(a). Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS_ON, and PROJECT files. However, this query needs only one record from the PROJECT relation—for the 'Aquarius' project—and only the EMPLOYEE records for those whose date of birth is after '1957-12-31'. Figure 15.5(b) shows an improved query tree that first applies the SELECT operations to reduce the number of tuples that appear in the CARTESIAN PRODUCT.

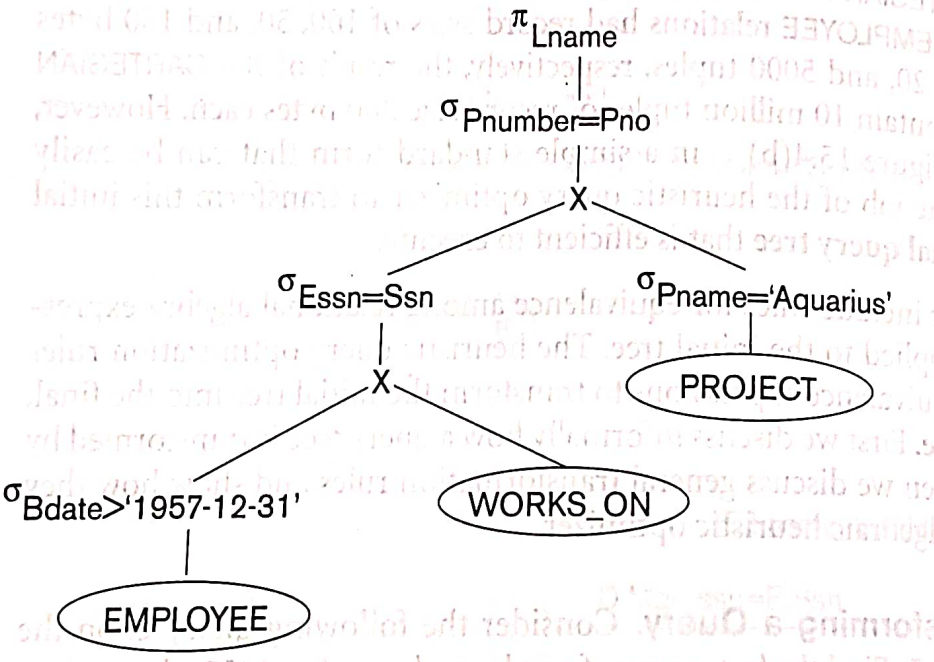
A further improvement is achieved by switching the positions of the EMPLOYEE and PROJECT relations in the tree, as shown in Figure 15.5(c). This uses the information

16. The same query may also be stated in various ways in a high-level query language such as SQL (see Chapter 8).

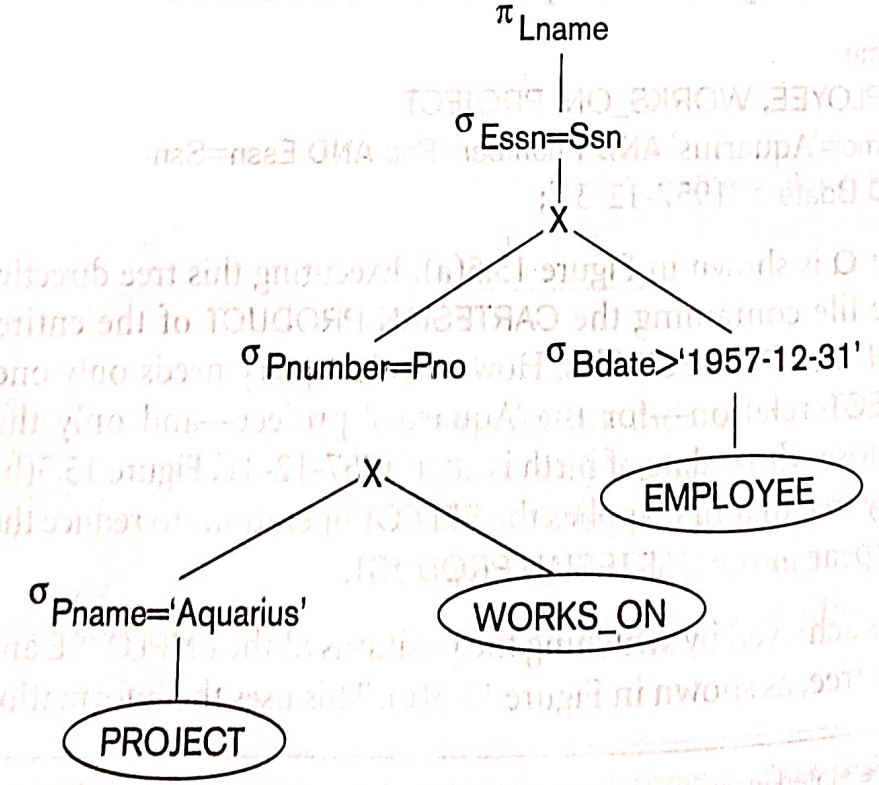
(a)


πLname
σPname='Aquarius' AND Pnumber=Pno AND Essn=Ssn AND Bdate>'1957-12-31'
X
X
PROJECT
EMPLOYEE
WORKS_ON

(b)


πLname
σPnumber=Pno
X
σEssn=Ssn
σPname='Aquarius'
X
PROJECT
σBdate>'1957-12-31'
WORKS_ON
EMPLOYEE

(c)


πLname
σEssn=Ssn
X
σPnumber=Pno
σBdate>'1957-12-31'
X
EMPLOYEE
σPname='Aquarius'
WORKS_ON
PROJECT

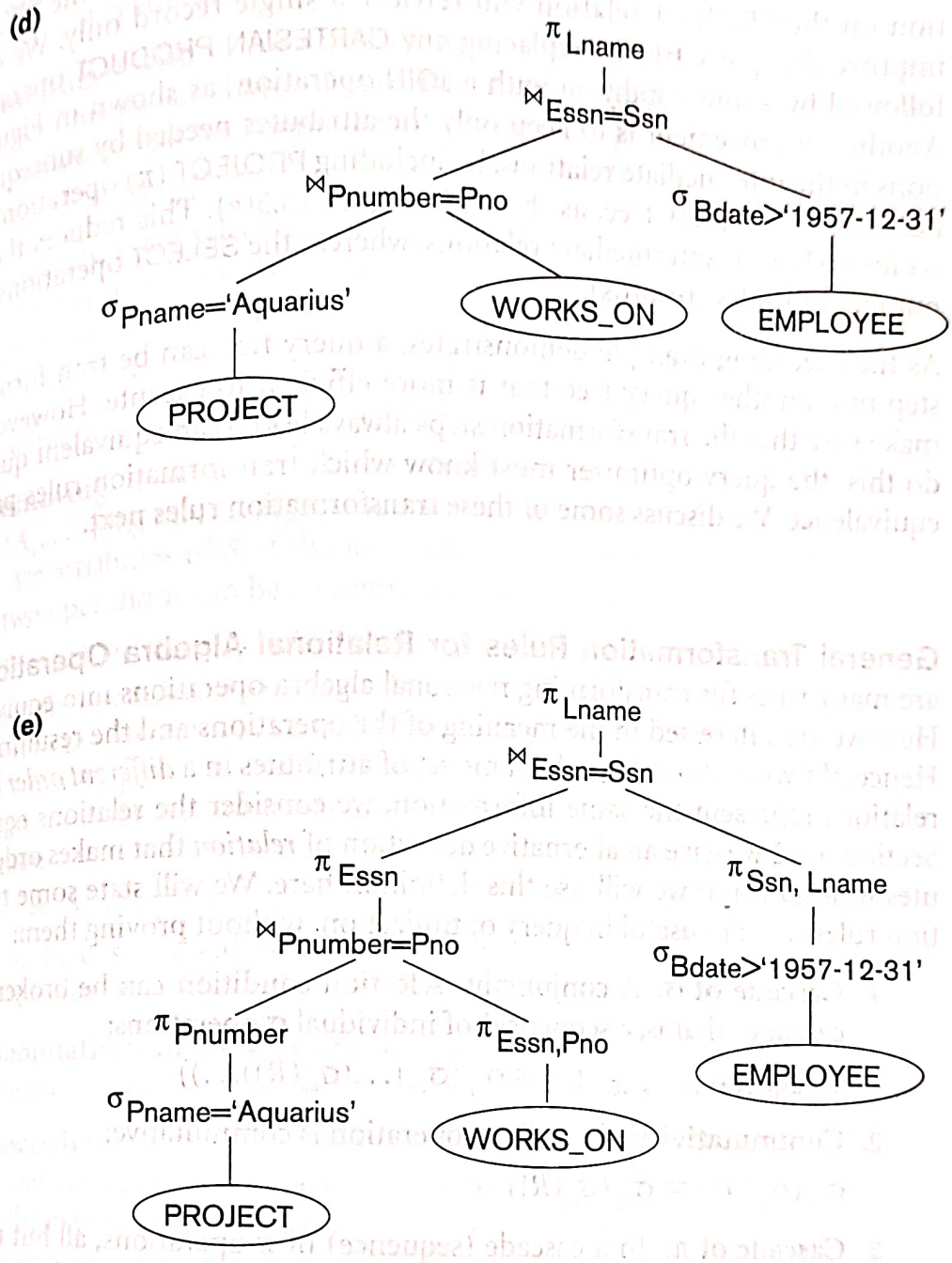


Figure 15.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.

that Pnumber is a key attribute of the project relation, and hence the SELECT operation on the PROJECT relation will retrieve a single record only. We can further improve the query tree by replacing any CARTESIAN PRODUCT operation that is followed by a join condition with a JOIN operation, as shown in Figure 15.5(d). Another improvement is to keep only the attributes needed by subsequent operations in the intermediate relations, by including PROJECT (π) operations as early as possible in the query tree, as shown in Figure 15.5(e). This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

As the preceding example demonstrates, a query tree can be transformed step by step into another query tree that is more efficient to execute. However, we must make sure that the transformation steps always lead to an equivalent query tree. To do this, the query optimizer must know which transformation rules preserve this equivalence. We discuss some of these transformation rules next.

General Transformation Rules for Relational Algebra Operations. There are many rules for transforming relational algebra operations into equivalent ones. Here we are interested in the meaning of the operations and the resulting relations. Hence, if two relations have the same set of attributes in a *different order* but the two relations represent the same information, we consider the relations equivalent. In Section 5.1.2 we gave an alternative definition of *relation* that makes order of attributes unimportant; we will use this definition here. We will state some transformation rules that are useful in query optimization, without proving them:

1. **Cascade of σ.** A conjunctive selection condition can be broken up into a cascade (that is, a sequence) of individual σ operations:

   $$\sigma_{c_1 \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$$

2. **Commutativity of σ.** The σ operation is commutative:

   $$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

3. **Cascade of π.** In a cascade (sequence) of π operations, all but the last one can be ignored:

   $$\pi_{\text{List}_1}(\pi_{\text{List}_2}(\ldots(\pi_{\text{List}_n}(R))\ldots)) \equiv \pi_{\text{List}_1}(R)$$

4. **Commuting σ with π.** If the selection condition c involves only those attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:

   $$\pi_{A_1, A_2, \ldots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, A_2, \ldots, A_n}(R))$$

5. **Commutativity of $\bowtie$ (and $\times$).** The $\bowtie$ operation is commutative, as is the $\times$ operation:

   $$R \bowtie_c S \equiv S \bowtie_c R$$

   $$R \times S \equiv S \times R$$

   Notice that, although the order of attributes may not be the same in the relations resulting from the two joins (or two Cartesian products), the *meaning*

is the same because order of attributes is not important in the alternative definition of relation.

6. **Commuting σ with ⋈ (or ×).** If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:

$$\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S$$

Alternatively, if the selection condition c can be written as (c_1 AND c_2), where condition c_1 involves only the attributes of R and condition c_2 involves only the attributes of S, the operations commute as follows:

$$\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S))$$

The same rules apply if the ⋈ is replaced by a × operation.

7. **Commuting π with ⋈ (or ×).** Suppose that the projection list is $L = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$, where $A_1, \ldots, A_n$ are attributes of R and $B_1, \ldots, B_m$ are attributes of S. If the join condition c involves only attributes in L, the two operations can be commuted as follows:

$$\pi_L(R \bowtie_c S) \equiv (\pi_{A_1, \ldots, A_n}(R)) \bowtie_c (\pi_{B_1, \ldots, B_m}(S))$$

If the join condition c contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed. For example, if attributes $A_{n+1}, \ldots, A_{n+k}$ of R and $B_{m+1}, \ldots, B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

$$\pi_L(R \bowtie_c S) \equiv \pi_L((\pi_{A_1, \ldots, A_n, A_{n+1}, \ldots, A_{n+k}}(R)) \bowtie_c (\pi_{B_1, \ldots, B_m, B_{m+1}, \ldots, B_{m+p}}(S)))$$

For ×, there is no condition c, so the first transformation rule always applies by replacing $\bowtie_c$ with ×.

8. **Commutativity of set operations.** The set operations ∪ and ∩ are commutative but – is not.

9. **Associativity of ⋈, ×, ∪, and ∩.** These four operations are individually associative; that is, if θ stands for any one of these four operations (throughout the expression), we have:

$$(R \,\theta\, S) \,\theta\, T \equiv R \,\theta\, (S \,\theta\, T)$$

10. **Commuting σ with set operations.** The σ operation commutes with ∪, ∩, and –. If θ stands for any one of these three operations (throughout the expression), we have:

$$\sigma_c(R \,\theta\, S) \equiv (\sigma_c(R)) \,\theta\, (\sigma_c(S))$$

11. **The π operation commutes with ∪.**

$$\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$$

12. **Converting a (σ, ×) sequence into ⋈.** If the condition c of a σ that follows a × corresponds to a join condition, convert the (σ, ×) sequence into a ⋈ as follows:

$$(\sigma_c(R \times S)) \equiv (R \bowtie_c S)$$

There are other possible transformations. For example, a selection or join condition c can be converted into an equivalent condition by using the following rules (DeMorgan's laws):

NOT (c_1 AND c_2) ≡ (NOT c_1) OR (NOT c_2)
NOT (c_1 OR c_2) ≡ (NOT c_1) AND (NOT c_2)

Additional transformations discussed in Chapters 5 and 6 are not repeated here. We discuss next how transformations can be used in heuristic optimization.

Outline of a Heuristic Algebraic Optimization Algorithm. We can now outline the steps of an algorithm that utilizes some of the above rules to transform an initial query tree into an optimized tree that is more efficient to execute (in most cases). The algorithm will lead to transformations similar to those discussed in our example of Figure 15.5. The steps of the algorithm are as follows:

1. Using Rule 1, break up any SELECT operations with conjunctive conditions into a cascade of SELECT operations. This permits a greater degree of freedom in moving SELECT operations down different branches of the tree.
2. Using Rules 2, 4, 6, and 10 concerning the commutativity of SELECT with other operations, move each SELECT operation as far down the query tree as is permitted by the attributes involved in the select condition.
3. Using Rules 5 and 9 concerning commutativity and associativity of binary operations, rearrange the leaf nodes of the tree using the following criteria. First, position the leaf node relations with the most restrictive SELECT operations so they are executed first in the query tree representation. The definition of *most restrictive* SELECT can mean either the ones that produce a relation with the fewest tuples or with the smallest absolute size.[17] Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.[18]
4. Using Rule 12, combine a CARTESIAN PRODUCT operation with a subsequent SELECT operation in the tree into a JOIN operation, if the condition represents a join condition.
5. Using Rules 3, 4, 7, and 11 concerning the cascading of PROJECT and the commuting of PROJECT with other operations, break down and move lists of projection attributes down the tree as far as possible by creating new PROJECT operations as needed. Only those attributes needed in the query

17. Either definition can be used, since these rules are heuristic.

18. Note that a CARTESIAN PRODUCT is acceptable in some cases—for example, if each relation has only a single tuple because each had a previous select condition on a key field.

result and in subsequent operations in the query tree should be kept after each PROJECT operation.

6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.

In our example, Figure 15.5(b) shows the tree of Figure 15.5(a) after applying steps 1 and 2 of the algorithm; Figure 15.5(c) shows the tree after step 3; Figure 15.5(d) after step 4; and Figure 15.5(e) after step 5. In step 6 we may group together the operations in the subtree whose root is the operation π_{Essn} into a single algorithm. We may also group the remaining operations into another subtree, where the tuples resulting from the first algorithm replace the subtree whose root is the operation π_{Essn}, because the first grouping means that this subtree is executed first.

Summary of Heuristics for Algebraic Optimization. We now summarize the basic heuristics for algebraic optimization. The main heuristic is to apply first the operations that reduce the size of intermediate results. This includes performing as early as possible SELECT operations to reduce the number of tuples and PROJECT operations to reduce the number of attributes. This is done by moving SELECT and PROJECT operations as far down the tree as possible. Additionally, the SELECT and JOIN operations that are most restrictive—that is, result in relations with the fewest tuples or with the smallest absolute size—should be executed before other similar operations. This is done by reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately.

15.7.3 Converting Query Trees into Query Execution Plans

An execution plan for a relational algebra expression represented as a query tree includes information about the access methods available for each relation as well as the algorithms to be used in computing the relational operators represented in the tree. As a simple example, consider query Q1 from Chapter 5, whose corresponding relational algebra expression is

$$\pi_{Fname,\ Lname,\ Address}(\sigma_{Dname='Research'}(DEPARTMENT) \bowtie_{Dnumber=Dno} EMPLOYEE)$$

The query tree is shown in Figure 15.6. To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation (assuming one exists), a table scan as access method for EMPLOYEE, a nested-loop join algorithm for the join, and a scan of the JOIN result for the PROJECT operator. Additionally, the approach taken for executing the query may specify a materialized or a pipelined evaluation.

With **materialized evaluation**, the result of an operation is stored as a temporary relation (that is, the result is *physically materialized*). For instance, the JOIN operation can be computed and the entire result stored as a temporary relation, which is then read as input by the algorithm that computes the PROJECT operation, which would produce the query result table. On the other hand, with **pipelined evaluation**, as the resulting tuples of an operation are produced, they are forwarded directly to the next operation in the query sequence. For example, as the selected

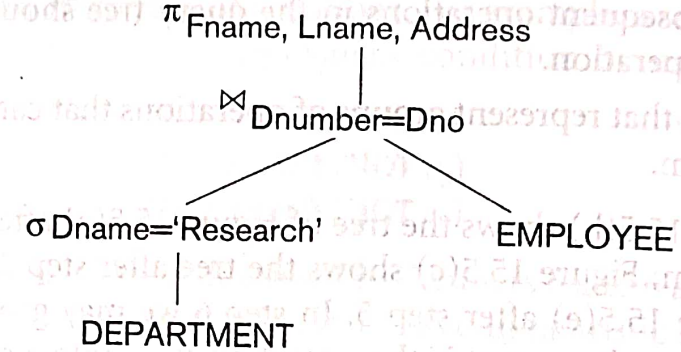


Figure 15.6
A query tree for query Q1.

tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm would then consume the tuples from the buffer, and those tuples that result from the JOIN operation are pipelined to the projection operation algorithm. The advantage of pipelining is the cost savings in not having to write the intermediate results to disk and not having to read them back for the next operation.