15

Algorithms for Query Processing and Optimization

In this chapter we discuss the techniques used by a DBMS to process, optimize, and execute high-level queries. A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated.[1] The **scanner** identifies the language tokens—such as SQL keywords, attribute names, and relation names—in the text of the query, whereas the **parser** checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language. The query must also be **validated,** by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried. An internal representation of the query is then created, usually as a tree data structure called a **query tree.** It is also possible to represent the query using a graph data structure called a **query graph.** The DBMS must then devise an **execution strategy** for retrieving the result of the query from the database files. A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as **query optimization.**

Figure 15.1 shows the different steps of processing a high-level query. The **query optimizer** module has the task of producing an execution plan, and the **code generator** generates the code to execute that plan. The **runtime database processor** has the task of running the query code,

1. We will not discuss the parsing and syntax-checking phase of query processing here; this material is discussed in compiler textbooks.

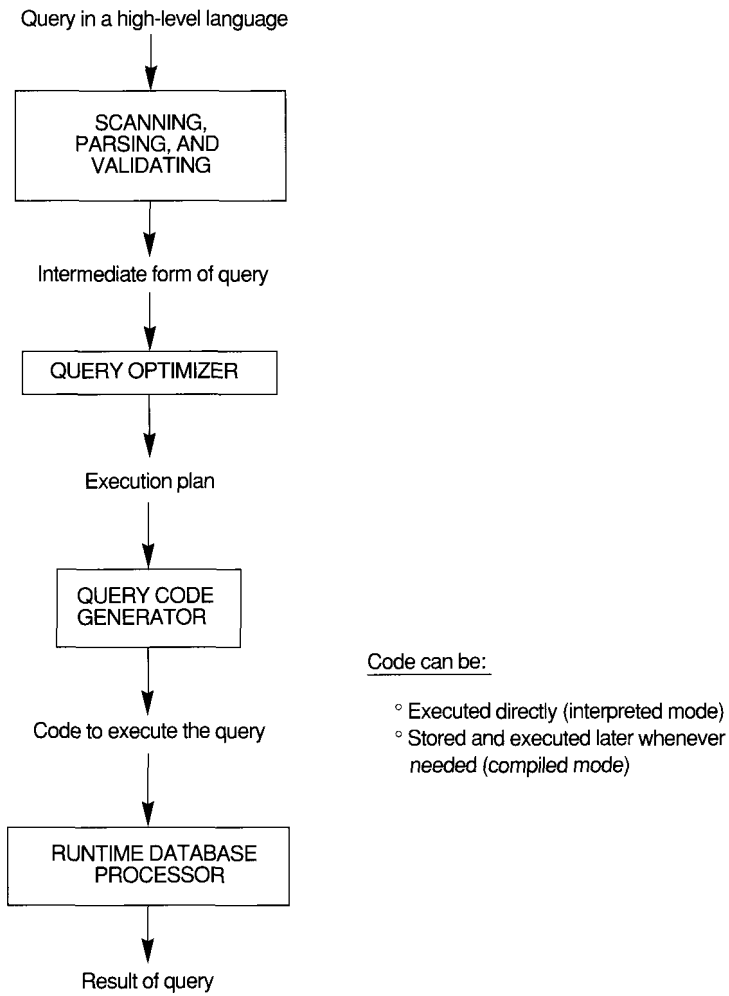

FIGURE 15.1 Typical steps when processing a high-level query.

whether in compiled or interpreted mode, to produce the query result. If a runtime error results, an error message is generated by the runtime database processor.

The term *optimization* is actually a misnomer because in some cases the chosen execution plan is not the optimal (best) strategy—it is just a *reasonably efficient strategy* for executing the query. Finding the optimal strategy is usually too time-consuming except for the simplest of queries and may require information on how the files are implemented and even on the contents of the files—information that may not be fully available in the DBMS catalog. Hence, *planning of an execution strategy* may be a more accurate description than *query optimization*.

For lower-level navigational database languages in legacy systems—such as the network DML or the hierarchical HDML (see Appendixes E and F)—the programmer must

choose the query execution strategy while writing a database program. If a DBMS provides only a navigational language, there is *limited need or opportunity* for extensive query optimization by the DBMS; instead, the programmer is given the capability to choose the "optimal" execution strategy. On the other hand, a high-level query language—such as SQL for relational DBMSs (RDBMSs) or OQL (see Chapter 21) for object DBMSs (ODBMSs)—is more declarative in nature because it specifies what the intended results of the query are, rather than identifying the details of *how* the result should be obtained. Query optimization is thus necessary for queries that are specified in a high-level query language.

We will concentrate on describing query optimization in the context of an RDBMS because many of the techniques we describe have been adapted for ODBMSs.[2] A relational DBMS must systematically evaluate alternative query execution strategies and choose a reasonably efficient or optimal strategy. Each DBMS typically has a number of general database access algorithms that implement relational operations such as SELECT or JOIN or combinations of these operations. Only execution strategies that can be implemented by the DBMS access algorithms and that apply to the particular query and particular physical database design can be considered by the query optimization module.

We start in Section 15.1 with a general discussion of how SQL queries are typically translated into relational algebra queries and then optimized. We then discuss algorithms for implementing relational operations in Sections 15.2 through 15.6. Following this, we give an overview of query optimization strategies. There are two main techniques for implementing query optimization. The first technique is based on **heuristic rules** for ordering the operations in a query execution strategy. A heuristic is a rule that works well in most cases but is not guaranteed to work well in every possible case. The rules typically reorder the operations in a query tree. The second technique involves **systematically estimating** the cost of different execution strategies and choosing the execution plan with the lowest cost estimate. The two techniques are usually combined in a query optimizer. We discuss heuristic optimization in Section 15.7 and cost estimation in Section 15.8. We then provide a brief overview of the factors considered during query optimization in the ORACLE commercial RDBMS in Section 15.9. Section 15.10 introduces the topic of semantic query optimization, in which known constraints are used to devise efficient query execution strategies.

15.1 Translating SQL Queries into Relational Algebra

In practice, SQL is the query language that is used in most commercial RDBMSs. An SQL query is first translated into an equivalent extended relational algebra expression—represented as a query tree data structure—that is then optimized. Typically, SQL queries are decomposed into **query blocks,** which form the basic units that can be translated into the

2. There are some query optimization problems and techniques that are pertinent only to ODBMSs. However, we do not discuss these here as we can give only an introduction to query optimization.

algebraic operators and optimized. A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as separate query blocks. Because SQL includes aggregate operators—such as MAX, MIN, SUM, and COUNT—these operators must also be included in the extended algebra, as we discussed in Section 6.4.

Consider the following SQL query on the EMPLOYEE relation in Figure 5.5:

```
SELECT  LNAME, FNAME
FROM    EMPLOYEE
WHERE   SALARY >  (SELECT    MAX (SALARY)
                   FROM      EMPLOYEE
                   WHERE     DNO=5);
```

This query includes a nested subquery and hence would be decomposed into two blocks. The inner block is

```
(SELECT MAX (SALARY)
FROM    EMPLOYEE
WHERE   DNO=5)
```

and the outer block is

```
SELECT LNAME, FNAME
FROM   EMPLOYEE
WHERE  SALARY > c
```

where c represents the result returned from the inner block. The inner block could be translated into the extended relational algebra expression

$$\mathfrak{F}_{\text{MAX SALARY}}(\sigma_{\text{DNO}=5}(\text{EMPLOYEE}))$$

and the outer block into the expression

$$\pi_{\text{LNAME, FNAME}}(\sigma_{\text{SALARY}>c}(\text{EMPLOYEE}))$$

The *query optimizer* would then choose an execution plan for each block. We should note that in the above example, the inner block needs to be evaluated only once to produce the maximum salary, which is then used—as the constant c—by the outer block. We called this an *uncorrelated nested query* in Chapter 8. It is much harder to optimize the more complex *correlated nested queries* (see Section 8.5), where a tuple variable from the outer block appears in the WHERE-clause of the inner block.

15.2 Algorithms for External Sorting

Sorting is one of the primary algorithms used in query processing. For example, whenever an SQL query specifies an ORDER BY-clause, the query result must be sorted. Sorting is also a key component in sort-merge algorithms used for JOIN and other operations (such as UNION and INTERSECTION), and in duplicate elimination algorithms for the PROJECT operation (when an SQL query specifies the DISTINCT option in the SELECT clause). We will discuss one of these algorithms in this section. Note that sorting may be avoided if an appropriate index exists to allow ordered access to the records.

External sorting refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.[3] The typical external sorting algorithm uses a **sort-merge strategy,** which starts by sorting small subfiles—called **runs**—of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn. The sort-merge algorithm, like other database algorithms, requires *buffer space* in main memory, where the actual sorting and merging of the runs is performed. The basic algorithm, outlined in Figure 15.2, consists of two phases: (1) the sorting phase and (2) the merging phase.

In the **sorting phase,** runs (portions or pieces) of the file that can fit in the available buffer space are read into main memory, sorted using an internal sorting algorithm, and written back to disk as temporary sorted subfiles (or runs). The size of a run and **number of initial runs (n_R)** is dictated by the **number of file blocks (b)** and the **available buffer**

```
set   i ← 1;
      j ← b;    {size of the file in blocks}
      k ← nB;   {size of buffer in blocks}
      m ← ⌈(j/k)⌉;
{Sort Phase}
while (i <= m)
      do {
         read next k blocks of the file into the buffer or if there are less than k blocks remaining,
             then read in the remaining blocks;
         sort the records in the buffer and write as a temporary subfile;
         i ← i + 1;
         }
{Merge Phase: merge subfiles until only 1 remains}
set   i ← 1;
      p ← ⌈log(k-1) m⌉;  {p is the number of passes for the merging phase}
      j ← m;
while (i <= p)
      do {
         n ← 1;
         q ← ⌈(j/ (k-1))⌉ ;  {number of subfiles to write in this pass}
         while (n <= q)
              do {
                read next k-1 subfiles or remaining subfiles (from previous pass) one block at a time;
                merge and write as new subfile one block at a time;
                n ← n + 1;
                  }
         j ← q;
         i ← i + 1;
                  }
```

FIGURE 15.2 Outline of the sort-merge algorithm for external sorting.

3. *Internal sorting algorithms* are suitable for sorting data structures that can fit entirely in memory.

space (n_B). For example, if n_B = 5 blocks and the size of the file b = 1024 blocks, then $n_R = \lceil (b/n_B) \rceil$, or 205 initial runs each of size 5 blocks (except the last run which will have 4 blocks). Hence, after the sort phase, 205 sorted runs are stored as temporary subfiles on disk.

In the **merging phase,** the sorted runs are merged during one or more **passes.** The **degree of merging (d_M)** is the number of runs that can be merged together in each pass. In each pass, one buffer block is needed to hold one block from each of the runs being merged, and one block is needed for containing one block of the merge result. Hence, d_M is the smaller of (n_B − 1) and n_R, and the number of passes is $\lceil (log_{dM}(n_R)) \rceil$. In our example, d_M = 4 (four-way merging), so the 205 initial sorted runs would be merged into 52 at the end of the first pass, which are then merged into 13, then 4, then 1 run, which means that *four passes* are needed. The minimum d_M of 2 gives the worst-case performance of the algorithm, which is

$$(2 * b) + (2 * (b * (\log_2 b)))$$

The first term represents the number of block accesses for the sort phase, since each file block is accessed twice—once for reading into memory and once for writing the records back to disk after sorting. The second term represents the number of block accesses for the merge phase, assuming the worst-case d_M of 2. In general, the log is taken to the base d_M and the expression for number of block accesses becomes

$$(2 * b) + (2 * (b * (\log_{dM} n_R)))$$

15.3 Algorithms for SELECT and JOIN Operations

15.3.1 Implementing the SELECT Operation

There are many options for executing a SELECT operation; some depend on the file having specific access paths and may apply only to certain types of selection conditions. We discuss some of the algorithms for implementing SELECT in this section. We will use the following operations, specified on the relational database of Figure 5.5, to illustrate our discussion:

(OP1): $\sigma_{SSN='123456789'}$(EMPLOYEE)
(OP2): $\sigma_{DNUMBER>5}$(DEPARTMENT)
(OP3): $\sigma_{DNO=5}$(EMPLOYEE)
(OP4): $\sigma_{DNO=5\ AND\ SALARY>30000\ AND\ SEX='F'}$(EMPLOYEE)
(OP5): $\sigma_{ESSN='123456789'\ AND\ PNO=10}$(WORKS_ON)

Search Methods for Simple Selection. A number of search algorithms are possible for selecting records from a file. These are also known as **file scans,** because they scan the records of a file to search for and retrieve records that satisfy a selection condition.[4]

4. A selection operation is sometimes called a **filter,** since it filters out the records in the file that do *not* satisfy the selection condition.

If the search algorithm involves the use of an index, the index search is called an **index scan.** The following search methods (S1 through S6) are examples of some of the search algorithms that can be used to implement a select operation:

- S1. *Linear search (brute force):* Retrieve *every record* in the file, and test whether its attribute values satisfy the selection condition.
- S2. *Binary search:* If the selection condition involves an equality comparison on a key attribute on which the file is ordered, binary search—which is more efficient than linear search—can be used. An example is OP1 if SSN is the ordering attribute for the EMPLOYEE file.[5]
- S3. *Using a primary index (or hash key):* If the selection condition involves an equality comparison on a **key attribute** with a primary index (or hash key)—for example, SSN = '123456789' in OP1—use the primary index (or hash key) to retrieve the record. Note that this condition retrieves a single record (at most).
- S4. *Using a primary index to retrieve multiple records:* If the comparison condition is >, >=, <, or <= on a key field with a primary index—for example, DNUMBER > 5 in OP2—use the index to find the record satisfying the corresponding equality condition (DNUMBER = 5), then retrieve all subsequent records in the (ordered) file. For the condition DNUMBER < 5, retrieve all the preceding records.
- S5. *Using a clustering index to retrieve multiple records:* If the selection condition involves an equality comparison on a **non-key attribute** with a clustering index—for example, DNO = 5 in OP3—use the index to retrieve all the records satisfying the condition.
- S6. *Using a secondary (B^+-tree) index on an equality comparison:* This search method can be used to retrieve a single record if the indexing field is a **key** (has unique values) or to retrieve multiple records if the indexing field is **not a key.** This can also be used for comparisons involving >, >=, <, or <=.

In Section 15.8, we discuss how to develop formulas that estimate the access cost of these search methods in terms of number of block accesses and access time. Method S1 applies to any file, but all the other methods depend on having the appropriate access path on the attribute used in the selection condition. Methods S4 and S6 can be used to retrieve records in a certain *range*—for example, 30000<=SALARY<=35000. Queries involving such conditions are called **range queries.**

Search Methods for Complex Selection. If a condition of a SELECT operation is a **conjunctive condition**—that is, if it is made up of several simple conditions connected with the AND logical connective such as OP4 above—the DBMS can use the following additional methods to implement the operation:

- S7. *Conjunctive selection using an individual index:* If an attribute involved in any **single simple condition** in the conjunctive condition has an access path that

5. Generally, binary search is not used in database search because ordered files are not used unless they also have a corresponding primary index.

permits the use of one of the Methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record *satisfies the remaining simple conditions* in the conjunctive condition.

- S8. *Conjunctive selection using a composite index:* If two or more attributes are involved in equality conditions in the conjunctive condition and a composite index (or hash structure) exists on the combined fields—for example, if an index has been created on the composite key (ESSN, PNO) of the WORKS_ON file for OP5—we can use the index directly.
- S9. *Conjunctive selection by intersection of record pointers:*[6] If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive condition, and if the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the **set of record pointers** that satisfy the individual condition. The **intersection** of these sets of record pointers gives the record pointers that satisfy the conjunctive condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.[7]

Whenever a single condition specifies the selection—such as OP1, OP2, or OP3—we can only check whether an access path exists on the attribute involved in that condition. If an access path exists, the method corresponding to that access path is used; otherwise, the brute force linear search approach of method S1 can be used. Query optimization for a SELECT operation is needed mostly for conjunctive select conditions whenever *more than one* of the attributes involved in the conditions have an access path. The optimizer should choose the access path that *retrieves the fewest records* in the most efficient way by estimating the different costs (see Section 15.8) and choosing the method with the least estimated cost.

When the optimizer is choosing between multiple simple conditions in a conjunctive select condition, it typically considers the selectivity of each condition. The **selectivity (s)** is defined as the ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation), and thus is a number between zero and 1—zero selectivity means no records satisfy the condition and 1 means all the records satisfy the condition. Although exact selectivities of all conditions may not be available, **estimates of selectivities** are often kept in the DBMS catalog and are used by the optimizer. For example, for an equality condition on a key attribute of relation $r(R)$, $s = 1/|r(R)|$, where $|r(R)|$ is the number of tuples in relation $r(R)$. For an equality condition on an attribute with i *distinct values*, s can be estimated by $(|r(R)|/i)/|r(R)|$ or

6. A record pointer uniquely identifies a record and provides the address of the record on disk; hence, it is also called the **record identifier** or **record id.**

7. The technique can have many variations—for example, if the indexes are *logical indexes* that store primary key values instead of record pointers.

$1/i$, assuming that the records are evenly distributed among the distinct values.[8] Under this assumption, $|r(R)|/i$ records will satisfy an equality condition on this attribute. In general, the number of records satisfying a selection condition with selectivity s is estimated to be $|r(R)| * s$. The smaller this estimate is, the higher the desirability of using that condition first to retrieve records.

Compared to a conjunctive selection condition, a **disjunctive condition** (where simple conditions are connected by the OR logical connective rather than by AND) is much harder to process and optimize. For example, consider OP4′:

(OP49): $\sigma_{\text{DNO=5 OR SALARY>30000 OR SEX='F'}}(\text{EMPLOYEE})$

With such a condition, little optimization can be done, because the records satisfying the disjunctive condition are the *union* of the records satisfying the individual conditions. Hence, if any *one* of the conditions does not have an access path, we are compelled to use the brute force linear search approach. Only if an access path exists on *every* condition can we optimize the selection by retrieving the records satisfying each condition—or their record ids—and then applying the union operation to eliminate duplicates.

A DBMS will have available many of the methods discussed above, and typically many additional methods. The query optimizer must choose the appropriate one for executing each SELECT operation in a query. This optimization uses formulas that estimate the costs for each available access method, as we shall discuss in Section 15.8. The optimizer chooses the access method with the lowest estimated cost.

15.3.2 Implementing the JOIN Operation

The JOIN operation is one of the most time-consuming operations in query processing. Many of the join operations encountered in queries are of the EQUIJOIN and NATURAL JOIN varieties, so we consider only these two here. For the remainder of this chapter, the term **join** refers to an EQUIJOIN (or NATURAL JOIN). There are many possible ways to implement a **two-way join,** which is a join on two files. Joins involving more than two files are called **multiway joins.** The number of possible ways to execute multiway joins grows very rapidly. In this section we discuss techniques for implementing only two-way joins. To illustrate our discussion, we refer to the relational schema of Figure 5.5 once more—specifically, to the EMPLOYEE, DEPARTMENT, and PROJECT relations. The algorithms we consider are for join operations of the form

$R \bowtie_{A=B} S$

where A and B are domain-compatible attributes of R and S, respectively. The methods we discuss can be extended to more general forms of join. We illustrate four of the most common techniques for performing such a join, using the following example operations:

(OP6): $\text{EMPLOYEE} \bowtie_{\text{DNO=DNUMBER}} \text{DEPARTMENT}$

(OP7): $\text{DEPARTMENT} \bowtie_{\text{MGRSSN=SSN}} \text{EMPLOYEE}$

8. In more sophisticated optimizers, histograms representing the distribution of the records among the different attribute values can be kept in the catalog.

Methods for Implementing Joins

- J1. *Nested-loop join (brute force):* For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition $t[A] = s[B]$.[9]
- J2. *Single-loop join (using an access structure to retrieve the matching records):* If an index (or hash key) exists for one of the two join attributes—say, B of S—retrieve each record t in R, one at a time (single loop), and then use the access structure to retrieve directly all matching records s from S that satisfy $s[B] = t[A]$.
- J3. *Sort-merge join:* If the records of R and S are *physically sorted* (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way possible. Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B. If the files are not sorted, they may be sorted first by using external sorting (see Section 15.2). In this method, pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file—unless both A and B are nonkey attributes, in which case the method needs to be modified slightly. A sketch of the sort-merge join algorithm is given in Figure 15.3a. We use $R(i)$ to refer to the i^{th} record in R. A variation of the sort-merge join can be used when secondary indexes exist on both join attributes. The indexes provide the ability to access (scan) the records in order of the join attributes, but the records themselves are physically scattered all over the file blocks, so this method may be quite inefficient, as every record access may involve accessing a different disk block.
- J4. *Hash-join:* The records of files R and S are both hashed to the same hash file, using the same hashing function on the join attributes A of R and B of S as hash keys. First, a single pass through the file with fewer records (say, R) hashes its records to the hash file buckets; this is called the **partitioning phase,** since the records of R are partitioned into the hash buckets. In the second phase, called the **probing phase,** a single pass through the other file (S) then hashes each of its records to *probe* the appropriate bucket, and that record is combined with all matching records from R in that bucket. This simplified description of hash-join assumes that the smaller of the two files *fits entirely into memory buckets* after the first phase. We will discuss variations of hash-join that do not require this assumption below.

In practice, techniques J1 to J4 are implemented by accessing *whole disk blocks* of a file, rather than individual records. Depending on the available buffer space in memory, the number of blocks read in from the file can be adjusted.

9. For disk files, it is obvious that the loops will be over disk blocks so this technique has also been called *nested-block join*.

```
(a) sort the tuples in R on attribute A; (*assume R has n tuples (records) *)
    sort the tuples in S on attribute B; (*assume S has m tuples (records) *)
    set i←1, j←1;
    while (i ≤ n) and (j ≤ m)
    do{   if R(i)[A] > S(j)[B]
              then     set j← j+1
          elseif R(i)[A] < S(j)[B]
              then     set i← i+1
              else {   (* R(i)[A] = S(j)[B], so we output a matched tuple*)
                       output the combined tuple <R(i), S(j)> to T;
                           (*output other tuples that match R(i), if any*)
                           set l← j+1;
                           while (l ≤ m) and (R(i)[A] = S(l)[B])
                           do    {        output the combined tuple <R(i), S(l)> to T;
                                          set l← l+1
                                 }
                           (*output other tuples that match S(j), if any*)
                           set k← i+1;
                           while (k ≤ n) and (R(k)[A] = S(j)[B])
                           do    {        output the combined tuple <R(k), S(j)> to T;
                                          set k← k+1
                                 }
                          set i← i+1, j←j+1
                   }
     }

(b) create a tuple t[<attribute list>] in T' for each tuple t in R;
        (*T' contains the projection result before duplicate elimination*)
    if <attribute list> includes a key of R
        then T←T'
        else {   sort the tuples in T';
                 set i←1, j← 2;
                 while i ≤ n
                     do   {   output the tuple T'[i] to T;
                              while T'[i] = T'[j] and j ≤ n do j← j+1; (*eliminate duplicates*)
                              i←j; j← i+1
                          }
             }
    (* T contains the projection result after duplicate elimination *)
```

FIGURE 15.3 Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where R has n tuples and S has m tuples. (a) Implementing the operation $T \leftarrow R \bowtie_{A=B} S$. (b) Implementing the operation $T \leftarrow \pi_{<\text{attribute list}>}(R)$.

Effects of Available Buffer Space and Join Selection Factor on Join Performance. The buffer space available has an important effect on the various join algorithms. First, let us consider the nested-loop approach (J1). Looking again at the operation OP6 above, assume that the number of buffers available in main memory for implementing the join is $n_B = 7$ blocks (buffers). For illustration, assume that the DEPARTMENT file consists of $r_D = 50$ records stored in $b_D = 10$ disk blocks and that the EMPLOYEE file

```
(c) sort the tuples in R and S using the same unique sort attributes;
    set i←1, j←1;
    while (i ≤ n) and (j ≤ m)
    do  {   if R(i) > S(j)
            then  {   output S(j) to T;
                      set j←j+1
                  }
            elseif R(i) < S(j)
            then  {   output R(i) to T;
                      set i←i+1
                  }
            else set j←j+1 (*R(i)=S(j), so we skip one of the duplicate tuples*)
        }
    if (i ≤ n) then add tuples R(i) to R(n) to T;
    if (j ≤ m) then add tuples S(j) to S(m) to T;

(d) sort the tuples in R and S using the same unique sort attributes;
    set i←1, j←1;
    while (i ≤ n) and (j ≤ m)
    do  {   if R(i) > S(j)
            then      set j←j+1
            elseif R(i) < S(j)
            then      set i←i+1
            else  {   output R(i) to T; (* R(i)=S(j), so we output the tuple *)
                      set i←i+1, j←j+1
                  }
        }

(e) sort the tuples in R and S using the same unique sort attributes;
    set i←1, j←1;
    while (i ≤ n) and (j ≤ m)
    do  {   if R(i) > S(j)
            then      set j←j+1
            elseif R(i) < S(j)
            then  {   output R(i)to T; (* R(i) has no matching S(j), so output R(i)*)
                      set i←i+1
                  }
            else      set i←i+1, j←j+1
        }
    if (i ≤ n) then add tuples R(i) to R(n) to T;
```

FIGURE 15.3(CONTINUED) Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where R has n tuples and S has m tuples. (c) Implementing the operation $T \leftarrow R \cup S$. (d) Implementing the operation $T \leftarrow R \cap S$. (e) Implementing the operation $T \leftarrow R - S$.

consists of $r_E = 6000$ records stored in $b_E = 2000$ disk blocks. It is advantageous to read as many blocks as possible at a time into memory from the file whose records are used for the outer loop (that is, $n_B - 2$ blocks). The algorithm can then read one block at a time for the inner-loop file and use its records to **probe** (that is, search) the outer loop blocks in memory for matching records. This reduces the total number of block accesses. An extra buffer block is needed to contain the resulting records after they are joined, and the con-

tents of this buffer block are appended to the **result file**—the disk file that contains the join result—whenever it is filled. This buffer block is then reused to hold additional result records.

In the nested-loop join, it makes a difference which file is chosen for the outer loop and which for the inner loop. If EMPLOYEE is used for the outer loop, each block of EMPLOYEE is read once, and the entire DEPARTMENT file (each of its blocks) is read once for *each time* we read in $(n_B - 2)$ blocks of the EMPLOYEE file. We get the following:

Total number of blocks accessed for outer file = b_E

Number of times $(n_B - 2)$ blocks of outer file are loaded = $\lceil b_E/(n_B - 2) \rceil$

Total number of blocks accessed for inner file = $b_D * \lceil b_E/(n_B - 2) \rceil$

Hence, we get the following total number of block accesses:

$$b_E + (\lceil b_E/(n_B - 2) \rceil * b_D) = 2000 + (\lceil (2000/5) \rceil * 10) = 6000 \text{ block accesses}$$

On the other hand, if we use the DEPARTMENT records in the outer loop, by symmetry we get the following total number of block accesses:

$$b_D + (\lceil b_D/(n_B - 2) \rceil * b_E) = 10 + (\lceil (10/5) \rceil * 2000) = 4010 \text{ block accesses}$$

The join algorithm uses a buffer to hold the joined records of the result file. Once the buffer is filled, it is written to disk and reused.[10] If the result file of the join operation has b_{RES} disk blocks, each block is written once, so an additional b_{RES} block accesses should be added to the preceding formulas in order to estimate the total cost of the join operation. The same holds for the formulas developed later for other join algorithms. As this example shows, it is advantageous to use the file *with fewer blocks* as the outer-loop file in the nested-loop join.

Another factor that affects the performance of a join, particularly the single-loop method J2, is the percentage of records in a file that will be joined with records in the other file. We call this the **join selection factor**[11] of a file with respect to an equijoin condition with another file. This factor depends on the particular equijoin condition between the two files. To illustrate this, consider the operation OP7, which joins each DEPARTMENT record with the EMPLOYEE record for the manager of that department. Here, each DEPARTMENT record (there are 50 such records in our example) is expected to be joined with a *single* EMPLOYEE record, but many EMPLOYEE records (the 5950 of them that do not manage a department) will not be joined.

Suppose that secondary indexes exist on both the attributes SSN of EMPLOYEE and MGRSSN of DEPARTMENT, with the number of index levels $x_{SSN} = 4$ and $x_{MGRSSN} = 2$, respectively. We have two options for implementing method J2. The first retrieves each EMPLOYEE record and then uses the index on MGRSSN of DEPARTMENT to find a matching DEPARTMENT record. In this case, no

10. If we reserve two buffers for the result file, double buffering can be used to speed the algorithm (see Section 13.3).

11. This is different from the *join selectivity*, which we shall discuss in Section 15.8.

matching record will be found for employees who do not manage a department. The number of block accesses for this case is approximately

$$b_E + (r_E * (x_{\text{MGRSSN}} + 1)) = 2000 + (6000 * 3) = 20{,}000 \text{ block accesses}$$

The second option retrieves each DEPARTMENT record and then uses the index on SSN of EMPLOYEE to find a matching manager EMPLOYEE record. In this case, every DEPARTMENT record will have one matching EMPLOYEE record. The number of block accesses for this case is approximately

$$b_D + (r_D * (x_{\text{SSN}} + 1)) = 10 + (50 * 5) = 260 \text{ block accesses}$$

The second option is more efficient because the join selection factor of DEPARTMENT with respect to the join condition SSN = MGRSSN is 1, whereas the join selection factor of EMPLOYEE with respect to the same join condition is (50/6000), or 0.008. For method J2, either the smaller file or the file that has a match for every record (that is, the file with the high join selection factor) should be used in the (outer) join loop. It is also possible to create an index specifically for performing the join operation if one does not already exist.

The sort-merge join J3 is quite efficient if both files are already sorted by their join attribute. Only a single pass is made through each file. Hence, the number of blocks accessed is equal to the sum of the numbers of blocks in both files. For this method, both OP6 and OP7 would need $b_E + b_D = 2000 + 10 = 2010$ block accesses. However, both files are required to be ordered by the join attributes; if one or both are not, they may be sorted specifically for performing the join operation. If we estimate the cost of sorting an external file by $(b \log_2 b)$ block accesses, and if both files need to be sorted, the total cost of a sort-merge join can be estimated by $(b_E + b_D + b_E \log_2 b_E + b_D \log_2 b_D)$.[12]

Partition Hash Join and Hybrid Hash Join. The hash-join method J4 is also quite efficient. In this case only a single pass is made through each file, whether or not the files are ordered. If the hash table for the smaller of the two files can be kept entirely in main memory after hashing (partitioning) on its join attribute, the implementation is straightforward. If, however, parts of the hash file must be stored on disk, the method becomes more complex, and a number of variations to improve the efficiency have been proposed. We discuss two techniques: partition hash join and a variation called hybrid hash join, which has been shown to be quite efficient.

In the **partition hash join** algorithm, each file is first partitioned into M partitions using a **partitioning hash function** on the join attributes. Then, each pair of partitions is joined. For example, suppose we are joining relations R and S on the join attributes $R.A$ and $S.B$:

$$R \bowtie_{A=B} S$$

In the **partitioning phase,** R is partitioned into the M partitions $R_1, R_2, \ldots, R_M$, and S into the M partitions $S_1, S_2, \ldots, S_M$. The property of each pair of corresponding partitions R_i, S_i is that records in R_i *only need to be joined* with records in S_i, and vice versa. This property is ensured by using the *same hash function* to partition both files on their

12. We can use the more accurate formulas from Section 15.2 if we know the number of available buffers for sorting.

join attributes—attribute A for R and attribute B for S. The minimum number of in-memory buffers needed for the partitioning phase is $M + 1$. Each of the files R and S are partitioned separately. For each of the partitions, a single in-memory buffer—whose size is one disk block—is allocated to store the records that hash to this partition. Whenever the in-memory buffer for a partition gets filled, its contents are appended to a **disk subfile** that stores this partition. The partitioning phase has *two iterations*. After the first iteration, the first file R is partitioned into the subfiles $R_1, R_2, \ldots, R_M$, where all the records that hashed to the same buffer are in the same partition. After the second iteration, the second file S is similarly partitioned.

In the second phase, called the **joining** or **probing phase,** M *iterations* are needed. During iteration i, the two partitions R_i and S_i are joined. The minimum number of buffers needed for iteration i is the number of blocks in the smaller of the two partitions, say R_i, plus two additional buffers. If we use a nested loop join during iteration i, the records from the smaller of the two partitions R_i are copied into memory buffers; then all blocks from the other partition S_i are read—one at a time—and each record is used to **probe** (that is, search) partition R_i for matching record(s). Any matching records are joined and written into the result file. To improve the efficiency of in-memory probing, it is common to use an *in-memory hash table* for storing the records in partition R_i by using a *different* hash function from the partitioning hash function.[13]

We can approximate the cost of this partition hash-join as $3 * (b_R + b_S) + b_{RES}$ for our example, since each record is read once and written back to disk once during the partitioning phase. During the joining (probing) phase, each record is read a second time to perform the join. The *main difficulty* of this algorithm is to ensure that the partitioning hash function is **uniform**—that is, the partition sizes are nearly equal in size. If the partitioning function is **skewed** (nonuniform), then some partitions may be too large to fit in the available memory space for the second joining phase.

Notice that if the available in-memory buffer space $n_B > (b_R + 2)$, where b_R is the number of blocks for the *smaller* of the two files being joined, say R, then there is no reason to do partitioning since in this case the join can be performed entirely in memory using some variation of the nested-loop join based on hashing and probing. For illustration, assume we are performing the join operation OP6, repeated below:

(OP6): EMPLOYEE $\bowtie_{\text{DNO=DNUMBER}}$ DEPARTMENT

In this example, the smaller file is the DEPARTMENT file; hence, if the number of available memory buffers $n_B > (b_D + 2)$, the whole DEPARTMENT file can be read into main memory and organized into a hash table on the join attribute. Each EMPLOYEE block is then read into a buffer, and each EMPLOYEE record in the buffer is hashed on its join attribute and is used to *probe* the corresponding in-memory bucket in the DEPARTMENT hash table. If a matching record is found, the records are joined, and the result record(s) are written to the result buffer and eventually to the result file on disk. The cost in terms of block accesses is hence $(b_D + b_E)$, plus b_{RES}—the cost of writing the result file.

13. If the hash function used for partitioning is used again, all records in a partition will hash to the same bucket again.

The **hybrid hash-join algorithm** is a variation of partition hash join, where the *joining* phase for *one of the partitions* is included in the *partitioning* phase. To illustrate this, let us assume that the size of a memory buffer is one disk block; that n_B such buffers are *available*; and that the hash function used is $h(K) = K \bmod M$ so that M partitions are being created, where $M < n_B$. For illustration, assume we are performing the join operation OP6. In the *first pass* of the partitioning phase, when the hybrid hash-join algorithm is partitioning the smaller of the two files (DEPARTMENT in OP6), the algorithm divides the buffer space among the M partitions such that all the blocks of the *first partition* of DEPARTMENT completely reside in main memory. For each of the other partitions, only a single in-memory buffer—whose size is one disk block—is allocated; the remainder of the partition is written to disk as in the regular partition hash join. Hence, at the end of the *first pass of the partitioning phase*, the first partition of DEPARTMENT resides wholly in main memory, whereas each of the other partitions of DEPARTMENT resides in a disk subfile.

For the second pass of the partitioning phase, the records of the second file being joined—the larger file, EMPLOYEE in OP6—are being partitioned. If a record hashes to the *first partition*, it is joined with the matching record in DEPARTMENT and the joined records are written to the result buffer (and eventually to disk). If an EMPLOYEE record hashes to a partition other than the first, it is partitioned normally. Hence, at the end of the second pass of the partitioning phase, all records that hash to the first partition have been joined. Now there are M − 1 pairs of partitions on disk. Therefore, during the second **joining** or **probing** phase, M − 1 *iterations* are needed instead of M. The goal is to join as many records during the partitioning phase so as to save the cost of storing those records back to disk and rereading them a second time during the joining phase.

15.4 ALGORITHMS FOR PROJECT AND SET OPERATIONS

A PROJECT operation $\pi_{<\text{attribute list}>}(R)$ is straightforward to implement if <attribute list> includes a key of relation R, because in this case the result of the operation will have the same number of tuples as R, but with only the values for the attributes in <attribute list> in each tuple. If <attribute list> does not include a key of R, *duplicate tuples must be eliminated*. This is usually done by sorting the result of the operation and then eliminating duplicate tuples, which appear consecutively after sorting. A sketch of the algorithm is given in Figure 15.3b. Hashing can also be used to eliminate duplicates: as each record is hashed and inserted into a bucket of the hash file in memory, it is checked against those already in the bucket; if it is a duplicate, it is not inserted. It is useful to recall here that in SQL queries, the default is not to eliminate duplicates from the query result; only if the keyword DISTINCT is included are duplicates eliminated from the query result.

Set operations—UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT—are sometimes expensive to implement. In particular, the CARTESIAN PRODUCT operation $R \times S$ is quite expensive, because its result includes a record for each combination of

records from R and S. In addition, the attributes of the result include all attributes of R and S. If R has n records and j attributes and S has m records and k attributes, the result relation will have $n * m$ records and $j + k$ attributes. Hence, it is important to avoid the CARTESIAN PRODUCT operation and to substitute other equivalent operations during query optimization (see Section 15.7).

The other three set operations—UNION, INTERSECTION, and SET DIFFERENCE[14]—apply only to union-compatible relations, which have the same number of attributes and the same attribute domains. The customary way to implement these operations is to use variations of the **sort-merge technique:** the two relations are sorted on the same attributes, and, after sorting, a single scan through each relation is sufficient to produce the result. For example, we can implement the UNION operation, $R \cup S$, by scanning and merging both sorted files concurrently, and whenever the same tuple exists in both relations, only one is kept in the merged result. For the INTERSECTION operation, $R \cap S$, we keep in the merged result only those tuples that appear in *both relations*. Figure 15.3c to (e) sketches the implementation of these operations by sorting and merging. Some of the details are not included in these algorithms.

Hashing can also be used to implement UNION, INTERSECTION, and SET DIFFERENCE. One table is partitioned and the other is used to probe the appropriate partition. For example, to implement $R \cup S$, first hash (partition) the records of R; then, hash (probe) the records of S, but do not insert duplicate records in the buckets. To implement $R \cap S$, first partition the records of R to the hash file. Then, while hashing each record of S, probe to check if an identical record from R is found in the bucket, and if so add the record to the result file. To implement $R - S$, first hash the records of R to the hash file buckets. While hashing (probing) each record of S, if an identical record is found in the bucket, remove that record from the bucket.

15.5 Implementing Aggregate Operations and Outer Joins

15.5.1 Implementing Aggregate Operations

The aggregate operators (MIN, MAX, COUNT, AVERAGE, SUM), when applied to an entire table, can be computed by a table scan or by using an appropriate index, if available. For example, consider the following SQL query:

```
SELECT MAX(SALARY)
FROM   EMPLOYEE;
```

If an (ascending) index on SALARY exists for the EMPLOYEE relation, then the optimizer can decide on using the index to search for the largest value by following the *rightmost* pointer in each index node from the root to the rightmost leaf. That node would include

14. SET DIFFERENCE is called EXCEPT in SQL.

the largest SALARY value as its *last* entry. In most cases, this would be more efficient than a full table scan of EMPLOYEE, since no actual records need to be retrieved. The MIN aggregate can be handled in a similar manner, except that the *leftmost* pointer is followed from the root to leftmost leaf. That node would include the smallest SALARY value as its *first* entry.

The index could also be used for the COUNT, AVERAGE, and SUM aggregates, but only if it is a **dense index**—that is, if there is an index entry for every record in the main file. In this case, the associated computation would be applied to the values in the index. For a **nondense index,** the actual number of records associated with each index entry must be used for a correct computation (except for COUNT DISTINCT, where the number of distinct values can be counted from the index itself).

When a GROUP BY clause is used in a query, the aggregate operator must be applied separately to each group of tuples. Hence, the table must first be partitioned into subsets of tuples, where each partition (group) has the same value for the grouping attributes. In this case, the computation is more complex. Consider the following query:

```
SELECT   DNO, AVG(SALARY)
FROM     EMPLOYEE
GROUP BY DNO;
```

The usual technique for such queries is to first use either **sorting** or **hashing** on the grouping attributes to partition the file into the appropriate groups. Then the algorithm computes the aggregate function for the tuples in each group, which have the same grouping attribute(s) value. In the example query, the set of tuples for each department number would be grouped together in a partition and the average salary computed for each group.

Notice that if a **clustering index** (see Chapter 13) exists on the grouping attribute(s), then the records are *already partitioned* (grouped) into the appropriate subsets. In this case, it is only necessary to apply the computation to each group.

15.5.2 Implementing Outer Join

In Section 6.4, the *outer join operation* was introduced, with its three variations: left outer join, right outer join, and full outer join. We also discussed in Chapter 8 how these operations can be specified in SQL. The following is an example of a left outer join operation in SQL:

```
SELECT LNAME, FNAME, DNAME
FROM (EMPLOYEE LEFT OUTER JOIN DEPARTMENT ON DNO=DNUMBER);
```

The result of this query is a table of employee names and their associated departments. It is similar to a regular (inner) join result, with the exception that if an EMPLOYEE tuple (a tuple in the *left* relation) *does not have an associated department,* the employee's name will still appear in the resulting table, but the department name would be *null* for such tuples in the query result.

Outer join can be computed by modifying one of the join algorithms, such as nested-loop join or single-loop join. For example, to compute a *left* outer join, we use the left relation as the outer loop or single-loop because every tuple in the left relation must

appear in the result. If there are matching tuples in the other relation, the joined tuples are produced and saved in the result. However, if no matching tuple is found, the tuple is still included in the result but is padded with null value(s). The sort-merge and hash-join algorithms can also be extended to compute outer joins.

Alternatively, outer join can be computed by executing a combination of relational algebra operators. For example, the left outer join operation shown above is equivalent to the following sequence of relational operations:

1. Compute the (inner) JOIN of the EMPLOYEE and DEPARTMENT tables.

   TEMP1 $\leftarrow \pi_{\text{LNAME, FNAME, DNAME}}$ (EMPLOYEE $\bowtie_{\text{DNO=DNUMBER}}$ DEPARTMENT)

2. Find the EMPLOYEE tuples that do not appear in the (inner) JOIN result.

   TEMP2 $\leftarrow \pi_{\text{LNAME, FNAME}}$ (EMPLOYEE) $-$ $\pi_{\text{LNAME, FNAME}}$ (TEMP1)

3. Pad each tuple in TEMP2 with a null DNAME field.

   TEMP2 $\leftarrow$ TEMP2 $\times$ 'NULL'

4. Apply the UNION operation to TEMP1, TEMP2 to produce the LEFT OUTER JOIN result.

   RESULT $\leftarrow$ TEMP1 $\cup$ TEMP2

The cost of the outer join as computed above would be the sum of the costs of the associated steps (inner join, projections, and union). However, note that step 3 can be done as the temporary relation is being constructed in step 2; that is, we can simply pad each resulting tuple with a null. In addition, in step 4, we know that the two operands of the union are disjoint (no common tuples), so there is no need for duplicate elimination.

15.6 Combining Operations Using Pipelining

A query specified in SQL will typically be translated into a relational algebra expression that is *a sequence of relational operations*. If we execute a single operation at a time, we must generate temporary files on disk to hold the results of these temporary operations, creating excessive overhead. Generating and storing large temporary files on disk is time-consuming and can be unnecessary in many cases, since these files will immediately be used as input to the next operation. To reduce the number of temporary files, it is common to generate query execution code that correspond to algorithms for combinations of operations in a query.

For example, rather than being implemented separately, a JOIN can be combined with two SELECT operations on the input files and a final PROJECT operation on the resulting file; all this is implemented by one algorithm with two input files and a single output file. Rather than creating four temporary files, we apply the algorithm directly and get just one result file. In Section 15.7.2 we discuss how heuristic relational algebra optimization can group operations together for execution. This is called **pipelining** or **stream-based processing.**

It is common to create the query execution code dynamically to implement multiple operations. The generated code for producing the query combines several algorithms that correspond to individual operations. As the result tuples from one operation are produced, they are provided as input for subsequent operations. For example, if a join operation follows two select operations on base relations, the tuples resulting from each select are provided as input for the join algorithm in a **stream** or **pipeline** as they are produced.

15.7 Using Heuristics in Query Optimization

In this section we discuss optimization techniques that apply heuristic rules to modify the internal representation of a query—which is usually in the form of a query tree or a query graph data structure—to improve its expected performance. The parser of a high-level query first generates an *initial internal representation*, which is then optimized according to heuristic rules. Following that, a query execution plan is generated to execute groups of operations based on the access paths available on the files involved in the query.

One of the main **heuristic rules** is to apply SELECT and PROJECT operations *before* applying the JOIN or other binary operations. This is because the size of the file resulting from a binary operation—such as JOIN—is usually a multiplicative function of the sizes of the input files. The SELECT and PROJECT operations reduce the size of a file and hence should be applied *before* a join or other binary operation.

We start in Section 15.7.1 by introducing the query tree and query graph notations. These can be used as the basis for the data structures that are used for internal representation of queries. A query tree is used to represent a relational algebra or extended relational algebra expression, whereas a query graph is used to represent a relational calculus expression. We then show in Section 15.7.2 how heuristic optimization rules are applied to convert a query tree into an **equivalent query tree,** which represents a different relational algebra expression that is more efficient to execute but gives the same result as the original one. We also discuss the equivalence of various relational algebra expressions. Finally, Section 15.7.3 discusses the generation of query execution plans.

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 15.4a shows a query tree for query Q2 of Chapters 5 to 8: For every project located in 'Stafford', retrieve the project number, the controlling department number,

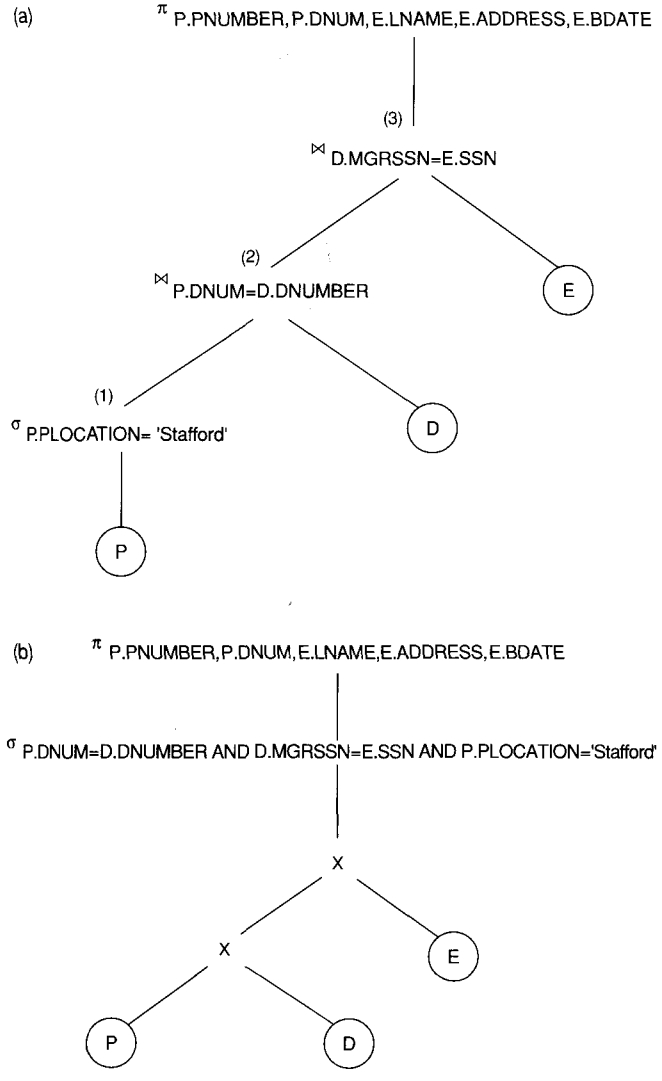

FIGURE 15.4 Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2.

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:

$$\pi_{\text{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}} ((\sigma_{\text{PLOCATION='STAFFORD'}}(\text{PROJECT})) \bowtie_{\text{DNUM=DNUMBER}} (\text{DEPARTMENT})) \bowtie_{\text{MGRSSN=SSN}} (\text{EMPLOYEE}))$$

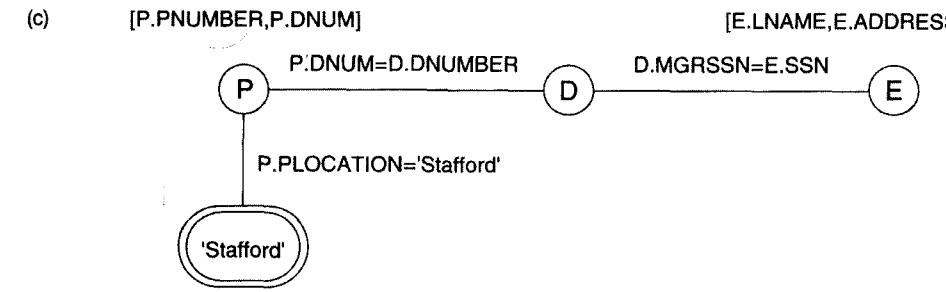

FIGURE 15.4(CONTINUED) (c) Query graph for Q2.

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.MGRSSN=E.SSN AND
           P.PLOCATION='STAFFORD';
```

In Figure 15.4a 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 15.4a 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 15.4c 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 15.4c. Finally, the attributes to be retrieved from each relation are displayed in square brackets above each relation.

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 (see Chapter 6).

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 15.4b. 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 15.4b 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. We first 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 15.5a. 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 15.5b 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 15.5c. This uses the information that PNUMBER is a key attribute of the project relation, and hence the SELECT operation on the

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

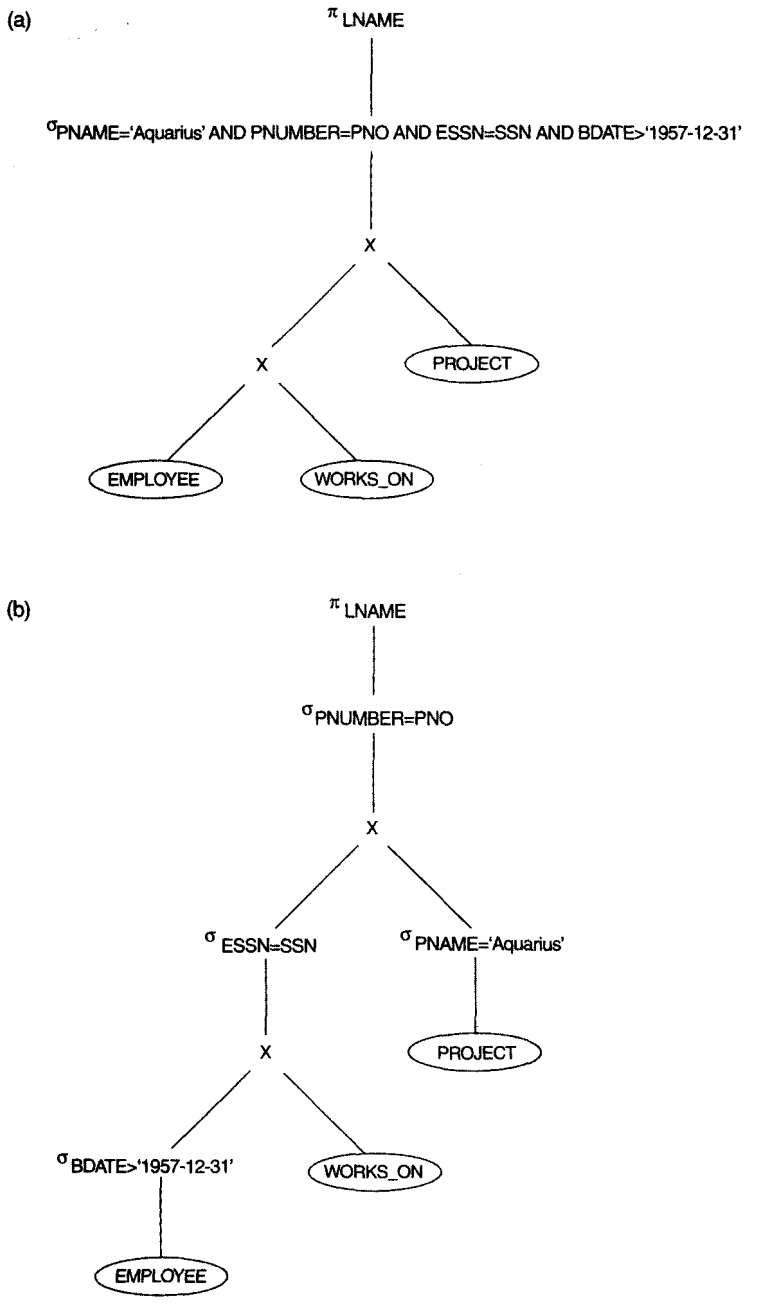

FIGURE 15.5 Steps in converting a query tree during heuristic optimization. (a) Initial (canonical) query tree for SQL query Q. (b) Moving SELECT operations down the query tree.

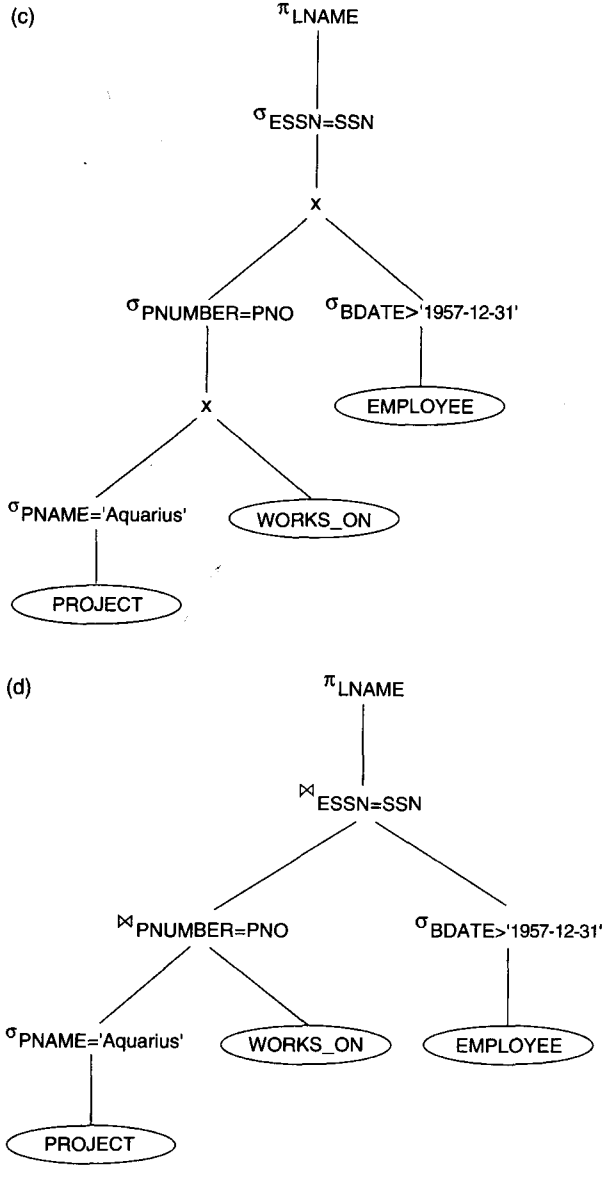

FIGURE 15.5(CONTINUED) Steps in converting a query tree during heuristic optimization. (c) Applying the more restrictive SELECT operation first. (d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.

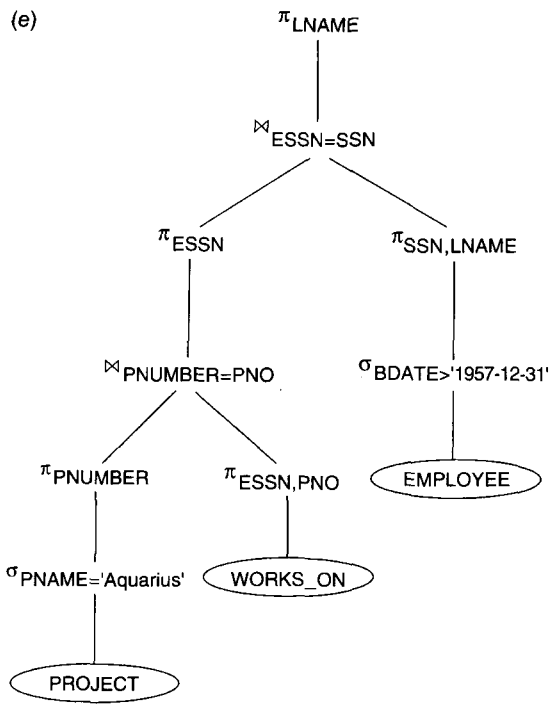

FIGURE 15.5(CONTINUED) Steps in converting a query tree during heuristic optimization. (e) Moving PROJECT operations down the query tree.

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 15.5d. 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 15.5e. 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 now 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:

   $\sigma_{c1 \text{ AND } c2 \text{ AND } \ldots \text{ AND } cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$

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

   $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$

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

   $\pi_{List1}(\pi_{List2}(\ldots(\pi_{Listn}(R))\ldots)) \equiv \pi_{List1}(R)$

4. Commuting σ with π: If the selection condition c involves only those attributes A1, ..., An in the projection list, the two operations can be commuted:

   $\pi_{A1, A2, \ldots, An}(\sigma_c(R)) \equiv \sigma_c(\pi_{A1, A2, \ldots, An}(R))$

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 $\bowtie$ (or $\times$): 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 (c1 AND c2), where condition c1 involves only the attributes of R and condition $c2$ involves only the attributes of S, the operations commute as follows:

   $\sigma_c(R \bowtie S) \equiv (\sigma_{c1}(R)) \bowtie (\sigma_{c2}(S))$

   The same rules apply if the $\bowtie$ is replaced by a $\times$ operation.

7. Commuting π with $\bowtie$ (or $\times$): 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:

   $\pi_L(R \bowtie_c S) \equiv (\pi_{A1, \ldots, An}(R)) \bowtie_c (\pi_{B1, \ldots, Bm}(S))$

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:

$$\pi_L (R \bowtie_c S) \equiv \pi_L ((\pi_{A1, \ldots, An, An+1, \ldots, An+k}(R)) \bowtie_c (\pi_{B1, \ldots, Bm, Bm+1, \ldots, Bm+p} (S)))$$

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

8. Commutativity of set operations: The set operations $\cup$ and $\cap$ are commutative but $-$ is not.
9. Associativity of $\bowtie$, $\times$, $\cup$, and $\cap$: 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 $\cup$, $\cap$, 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 $\cup$:

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

12. Converting a $(\sigma, \times)$ sequence into $\bowtie$: If the condition c of a σ that follows a $\times$ corresponds to a join condition, convert the $(\sigma, \times)$ sequence into a $\bowtie$ 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 (c1 AND c2) ≡ (NOT c1) OR (NOT c2)
NOT (c1 OR c2) ≡ (NOT c1) AND (NOT c2)
```

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 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

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.

as far down the tree as possible. In addition, 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

$$\pi_{\text{FNAME, LNAME, ADDRESS}}(\sigma_{\text{DNAME='RESEARCH'}}(\text{DEPARTMENT}) \bowtie_{\text{DNUMBER=DNO}} \text{EMPLOYEE})$$

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. In addition, 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 tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm would then consume

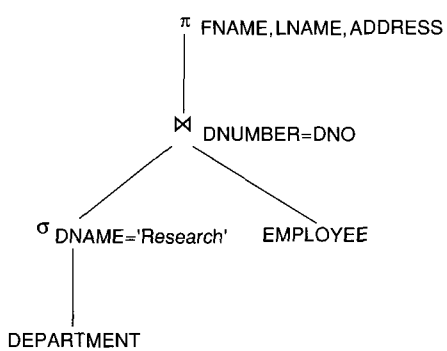

FIGURE 15.6 A query tree for query Q1.

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.

15.8 Using Selectivity and Cost Estimates in Query Optimization

A query optimizer should not depend solely on heuristic rules; it should also estimate and compare the costs of executing a query using different execution strategies and should choose the strategy with the *lowest cost estimate*. For this approach to work, accurate cost estimates are required so that different strategies are compared fairly and realistically. In addition, we must limit the number of execution strategies to be considered; otherwise, too much time will be spent making cost estimates for the many possible execution strategies. Hence, this approach is more suitable for **compiled queries** where the optimization is done at compile time and the resulting execution strategy code is stored and executed directly at runtime. For **interpreted queries,** where the entire process shown in Figure 15.1 occurs at runtime, a full-scale optimization may slow down the response time. A more elaborate optimization is indicated for compiled queries, whereas a partial, less time-consuming optimization works best for interpreted queries.

We call this approach **cost-based query optimization,**[19] and it uses traditional optimization techniques that search the *solution space* to a problem for a solution that minimizes an objective (cost) function. The cost functions used in query optimization are estimates and not exact cost functions, so the optimization may select a query execution strategy that is not the optimal one. In Section 15.8.1 we discuss the components of query execution cost. In Section 15.8.2 we discuss the type of information needed in cost functions. This information is kept in the DBMS catalog. In Section 15.8.3 we give examples of cost functions for the SELECT operation, and in Section 15.8.4 we discuss cost functions for two-way JOIN operations. Section 15.8.5 discusses multiway joins, and Section 15.8.6 gives an example.

15.8.1 Cost Components for Query Execution

The cost of executing a query includes the following components:

1. *Access cost to secondary storage:* This is the cost of searching for, reading, and writing data blocks that reside on secondary storage, mainly on disk. The cost of searching for records in a file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, factors

19. This approach was first used in the optimizer for the SYSTEM R experimental DBMS developed at IBM.