- **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]
- **S3a—Using a primary index.** If the selection condition involves an equality comparison on a **key attribute** with a primary index—for example, Ssn = '123456789' in OP1—use the primary index to retrieve the record. Note that this condition retrieves a single record (at most).
- **S3b—Using a hash key.** If the selection condition involves an equality comparison on a **key attribute** with a hash key—for example, Ssn = '123456789' in OP1—use the 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 **nonkey 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 19.8, we discuss how to develop formulas that estimate the access cost of these search methods in terms of the number of block accesses and access time. Method S1 (**linear search**) applies to any file, but all the other methods depend on having the appropriate access path on the attribute used in the selection condition. Method S2 (**binary search**) requires the file to be sorted on the search attribute. The methods that use an index (S3a, S4, S5, and S6) are generally referred to as **index searches**, and they require the appropriate index to exist on the search attribute. 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

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

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 select condition has an access path that 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 select condition.
- **S8—Conjunctive selection using a composite index.** If two or more attributes are involved in equality conditions in the conjunctive select 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 select 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 select 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] In general, method S9 assumes that each of the indexes is on a *nonkey field* of the file, because if one of the conditions is an equality condition on a key field, only one record will satisfy the whole condition.

Whenever a single condition specifies the selection—such as OP1, OP2, or OP3—the DBMS can only check whether or not an access path exists on the attribute involved in that condition. If an access path (such as index or hash key or sorted file) 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 19.8) and choosing the method with the least estimated cost.

Selectivity of a Condition. When the optimizer is choosing between multiple simple conditions in a conjunctive select condition, it typically considers the

[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.

selectivity of each condition. The **selectivity (*sl*)** 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 one. *Zero selectivity* means none of the records in the file satisfies the selection condition, and a selectivity of one means that all the records in the file satisfy the condition. In general, the selectivity will not be either of these two extremes, but will be a fraction that estimates the percentage of file records that will be retrieved.

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 a nonkey attribute with *i distinct values, s* can be estimated by $(|r(R)|/i)/|r(R)|$ or $1/i$, assuming that the records are evenly or **uniformly 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 *sl* is estimated to be $|r(R)| * sl$. The smaller this estimate is, the higher the desirability of using that condition first to retrieve records. In certain cases, the actual distribution of records among the various distinct values of the attribute is kept by the DBMS in the form of a *histogram*, in order to get more accurate estimates of the number of records that satisfy a particular condition.

Disjunctive Selection Conditions. 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′:

$$\text{OP4}'\text{: } \sigma_{\text{Dno=5 OR Salary > 30000 OR Sex='F'}}(\text{EMPLOYEE})$$

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* simple condition in the disjunction 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 will discuss in Section 19.8. The optimizer chooses the access method with the lowest estimated cost.

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

19.3.2 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 just these two here since we are only giving an overview of query processing and optimization. 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 in Figure 3.5 once more—specifically, to the EMPLOYEE, DEPARTMENT, and PROJECT relations. The algorithms we discuss next are for a join operation of the form:

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

where A and B are the **join attributes**, which should be 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 sample operations:

$$\text{OP6: EMPLOYEE} \bowtie_{\text{Dno=Dnumber}} \text{DEPARTMENT}$$
$$\text{OP7: DEPARTMENT} \bowtie_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE}$$

Methods for Implementing Joins.

- **J1—Nested-loop join (or nested-block join).** This is the default (brute force) algorithm, as it does not require any special access paths on either file in the join. 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, attribute B of file S—retrieve each record t in R (loop over file R), and then use the access structure (such as an index or a hash key) 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 19.2). 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

[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*.

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 19.3(a). We use $R(i)$ to refer to the ith record in file 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—Partition-hash join.** The records of files R and S are partitioned into smaller files. The partitioning of each file is done using the same hashing function h on the join attribute A of R (for partitioning file R) and B of S (for partitioning file S). First, a single pass through the file with fewer records (say, R) hashes its records to the various partitions of R; this is called the **partitioning phase**, since the records of R are partitioned into the hash buckets. In the simplest case, we assume that the smaller file can fit entirely in main memory after it is partitioned, so that the partitioned subfiles of R are all kept in main memory. The collection of records with the same value of $h(A)$ are placed in the same partition, which is a **hash bucket** in a hash table in main memory. In the second phase, called the **probing phase**, a single pass through the other file (S) then hashes each of its records using the same hash function $h(B)$ to *probe* the appropriate bucket, and that record is combined with all matching records from R in that bucket. This simplified description of partition-hash join assumes that the smaller of the two files *fits entirely into memory buckets* after the first phase. We will discuss the general case of partition-hash join that does 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 number of buffers in memory, the number of blocks read in from the file can be adjusted.

How Buffer Space and Choice of Outer-Loop File Affect Performance of Nested-Loop Join. The buffer space available has an important effect on some of the 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). Recall that we assume that each memory buffer is the same size as one disk block. 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 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 that are currently in main memory for matching records. This reduces the total number of block accesses. An extra buffer in main memory is needed to contain the resulting records after they are joined, and the contents of this result buffer can be appended to the **result file**—the disk file that will contain the join result—whenever it is filled. This result buffer block then is reused to hold additional join result records.

Figure 19.3
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 $T \leftarrow \pi_{<\text{attribute list}>}(R)$.

```
(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 ← k, j ← l
          }
    }

(b) create a tuple t[<attribute list>] in T′ for each tuple t in R;
       (* T′ contains the projection results 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 *)
```

(continues)

Figure 19.3 (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 $T \leftarrow R - S$.

(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(j) 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;
```

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 formulas for the number of disk blocks that are read from disk to main memory:

Total number of blocks accessed (read) for outer-loop file = b_E

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

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

Hence, we get the following total number of block read 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 its contents are appended to the result file, and then refilled with join result records.[10]

If the result file of the join operation has b_{RES} disk blocks, each block is written once to disk, so an additional b_{RES} block accesses (writes) 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.

How the Join Selection Factor Affects Join Performance. Another factor that affects the performance of a join, particularly the single-loop method J2, is the fraction of records in one 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) will be joined with a *single* EMPLOYEE record, but many EMPLOYEE records (the 5,950 of them that do not manage a department) will not be joined with any record from DEPARTMENT.

Suppose that secondary indexes exist on both the attributes Ssn of EMPLOYEE and Mgr_ssn of DEPARTMENT, with the number of index levels $x_{\text{Ssn}} = 4$ and $x_{\text{Mgr\_ssn}} = 2$,

[10] If we reserve two buffers for the result file, double buffering can be used to speed the algorithm (see Section 17.3).

[11] This is different from the *join selectivity*, which we will discuss in Section 19.8.

respectively. We have two options for implementing method J2. The first retrieves each EMPLOYEE record and then uses the index on Mgr_ssn of DEPARTMENT to find a matching DEPARTMENT record. In this case, no matching record will be found for employees who do not manage a department. The number of block accesses for this case is approximately:

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

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 = Mgr_ssn is 1 (every record in DEPARTMENT will be joined), whereas the join selection factor of EMPLOYEE with respect to the same join condition is (50/6000), or 0.008 (only 0.8 percent of the records in EMPLOYEE will be joined). 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 (single) 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, a sorted copy of each file must be created specifically for performing the join operation. If we roughly 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]

General Case for Partition-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, the partitions of both files 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: the general case of *partition-hash join* and a variation called *hybrid hash-join algorithm*, which has been shown to be quite efficient.

In the general case of **partition-hash join**, each file is first partitioned into M partitions using the same **partitioning hash function** on the join attributes. Then, each

[12]We can use the more accurate formulas from Section 19.2 if we know the number of available buffers for sorting.

pair of corresponding 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 $R_1, R_2, ..., R_M$, and S into the M partitions $S_1, S_2, ..., S_M$. The property of each pair of corresponding partitions R_i, S_i with respect to the join operation 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 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. During partitioning of a file, M in-memory buffers are allocated to store the records that hash to each partition, and one additional buffer is needed to hold one block at a time of the input file being partitioned. Whenever the in-memory buffer for a partition gets filled, its contents are appended to a **disk subfile** that stores the partition. The partitioning phase has *two iterations.* After the first iteration, the first file R is partitioned into the subfiles $R_1, R_2, ..., R_M$, 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, two corresponding 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.

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

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.

Hybrid Hash-Join. 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 partitioning 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 and stored to disk. Hence, at the end of the second pass of the partitioning phase, all records that hash to the first partition have been joined. At this point, 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 on disk and then rereading them a second time during the joining phase.

19.4 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 can be 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 19.3(b). 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 records already in the bucket; if it is a duplicate, it is not inserted in the bucket. It is useful to recall here that in SQL queries, the default is not to eliminate duplicates from the query result; duplicates are eliminated from the query result only if the keyword DISTINCT is included.

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. Also, each record in the result includes 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 for $R \times S$ will have $n * m$ records and each record will have $j + k$ attributes. Hence, it is important to avoid the CARTESIAN PRODUCT operation and to substitute other operations such as join during query optimization (see Section 19.7).

The other three set operations—UNION, INTERSECTION, and SET DIFFERENCE[14]—apply only to **type-compatible** (or 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 sorted relations.* Figure 19.3(c) 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 first scanned and then partitioned into an in-memory hash table with buckets, and the records in the other table are then scanned one at a time and 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.

[14]SET DIFFERENCE is called EXCEPT in SQL.

In SQL, there are two variations of these set operations. The operations UNION, INTERSECTION, and EXCEPT (the SQL keyword for the SET DIFFERENCE operation) apply to traditional sets, where no duplicate records exist in the result. The operations UNION ALL, INTERSECTION ALL, and EXCEPT ALL apply to multisets (or bags), and duplicates are fully considered. Variations of the above algorithms can be used for the multiset operations in SQL. We leave these as an exercise for the reader.

19.5 Implementing Aggregate Operations and OUTER JOINs

19.5.1 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) B^+-tree index on Salary exists for the EMPLOYEE relation, then the optimizer can decide on using the Salary index to search for the largest Salary value in the index by following the *rightmost* pointer in each index node from the root to the rightmost leaf. That node would include 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 function can be handled in a similar manner, except that the *leftmost* pointer in the index 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 AVERAGE and SUM aggregate functions, 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 value must be used for a correct computation. This can be done if the *number of records associated with each value* in the index is stored in each index entry. For the COUNT aggregate function, the number of values can be also computed from the index in a similar manner. If a COUNT(*) function is applied to a whole relation, the number of records currently in each relation are typically stored in the catalog, and so the result can be retrieved directly from the catalog.

When a GROUP BY clause is used in a query, the aggregate operator must be applied separately to each group of tuples as partitioned by the grouping attribute. 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 sample query, the set of EMPLOYEE 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 18) 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.

19.5.2 Implementing OUTER JOINs

In Section 6.4, the *outer join operation* was discussed, with its three variations: left outer join, right outer join, and full outer join. We also discussed in Chapter 5 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.

Theoretically, outer join can also 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.
   $\text{TEMP1} \leftarrow \pi_{\text{Lname, Fname, Dname}} (\text{EMPLOYEE} \bowtie \text{Dno=Dnumber DEPARTMENT})$
2. Find the EMPLOYEE tuples that do not appear in the (inner) JOIN result.
   $\text{TEMP2} \leftarrow \pi_{\text{Lname, Fname}} (\text{EMPLOYEE}) - \pi_{\text{Lname, Fname}} (\text{TEMP1})$
3. Pad each tuple in TEMP2 with a NULL Dname field.
   $\text{TEMP2} \leftarrow \text{TEMP2} \times \text{NULL}$

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

   RESULT ← TEMP1 ∪ TEMP2

The cost of the outer join as computed above would be the sum of the costs of the associated steps (inner join, projections, set difference, 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.

19.6 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 corresponds 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 19.7.2, 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.

19.7 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 scanner and parser of an SQL query first generate a data structure that corresponds to an *initial query representation,* which is then optimized according to heuristic rules. This leads to an *optimized query representation*, which corresponds to the query execution strategy. 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, 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.

In Section 19.7.1 we reiterate the query tree and query graph notations that we introduced earlier in the context of relational algebra and calculus in Sections 6.3.5 and 6.6.5, respectively. 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*. Then in Section 19.7.2 we show how heuristic optimization rules are applied to convert an initial 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 tree. We also discuss the equivalence of various relational algebra expressions. Finally, Section 19.7.3 discusses the generation of query execution plans.

19.7.1 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 order of execution of operations *starts at the leaf nodes*, which represents the input database relations for the query, and *ends at the root node*, which represents the final operation of the query. The execution terminates when the root node operation is executed and produces the result relation for the query.

Figure 19.4a shows a query tree (the same as shown in Figure 6.9) for query Q2 in Chapters 4 to 6: 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 COMPANY relational schema in Figure 3.5 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';
```

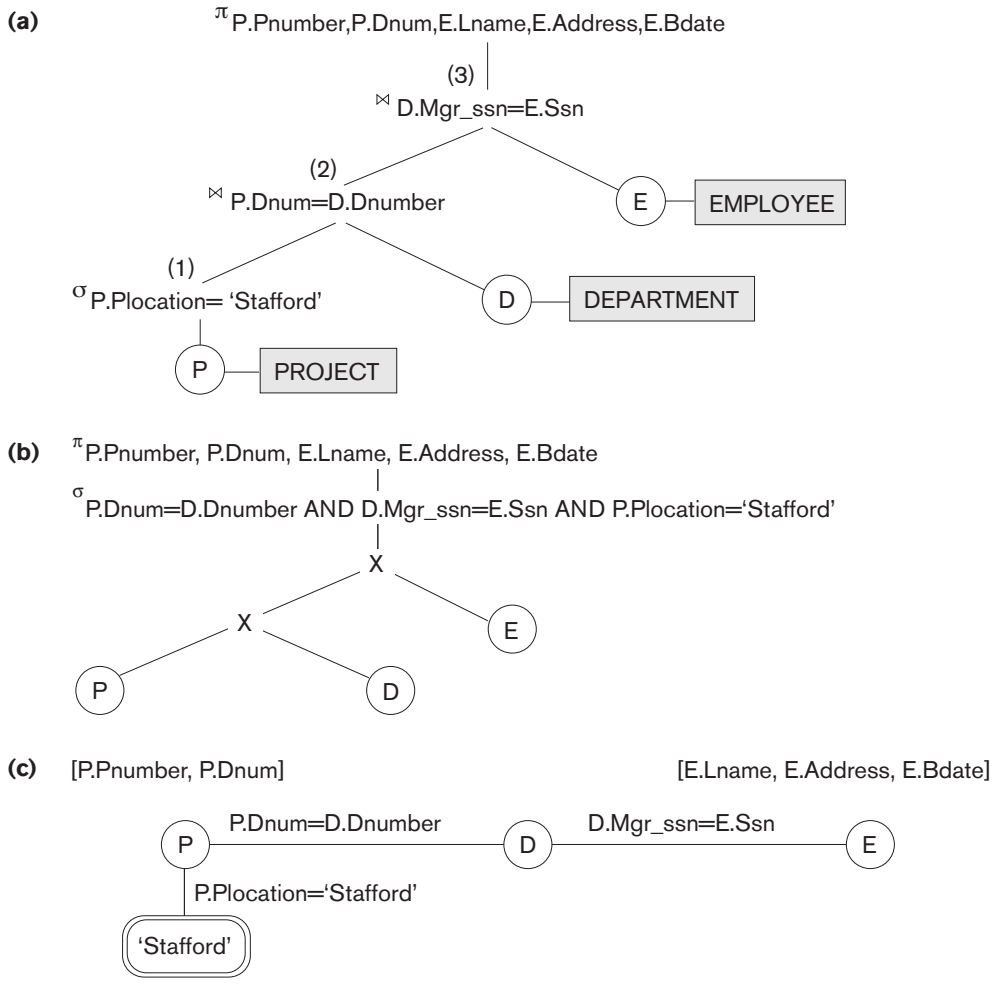


Figure 19.4
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

In Figure 19.4a, the leaf nodes P, D, and E represent the three relations PROJECT, DEPARTMENT, and EMPLOYEE, respectively, and the internal tree nodes represent the *relational algebra operations* of the expression. When this query tree is executed, the node marked (1) in Figure 19.4a 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 data structure for representation of a query is the **query graph** notation. Figure 19.4c (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 19.4c. 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.

19.7.2 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 represent 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 19.4(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 ($\times$) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5,000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each. However, the initial query tree in Figure 19.4(b) is in a simple standard form that can be easily created from the SQL query. It will never be executed. The heuristic query optimizer will transform this initial query tree into an equivalent **final query tree** that is efficient to execute.

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

Example of Transforming a Query. Consider the following query Q on the database in Figure 3.5: *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:

---

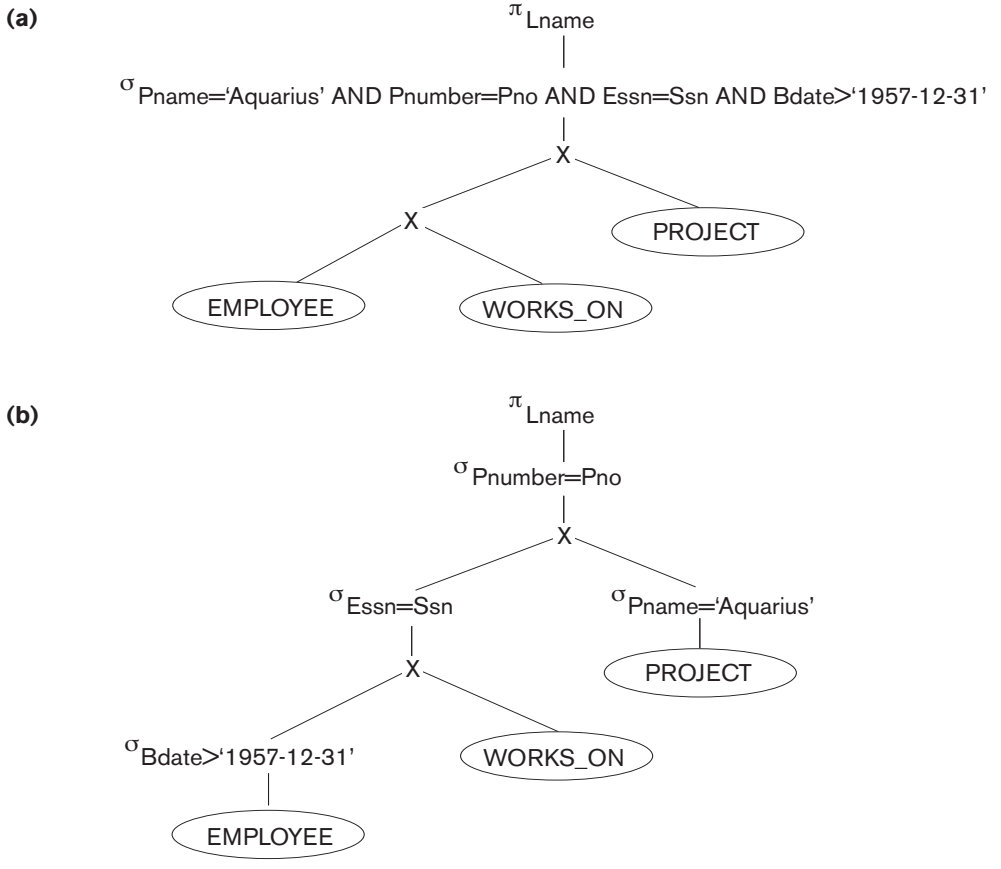
[15]Hence, a query graph corresponds to a *relational calculus* expression as shown in Section 6.6.5.

[16]The same query may also be stated in various ways in a high-level query language such as SQL (see Chapters 4 and 5).

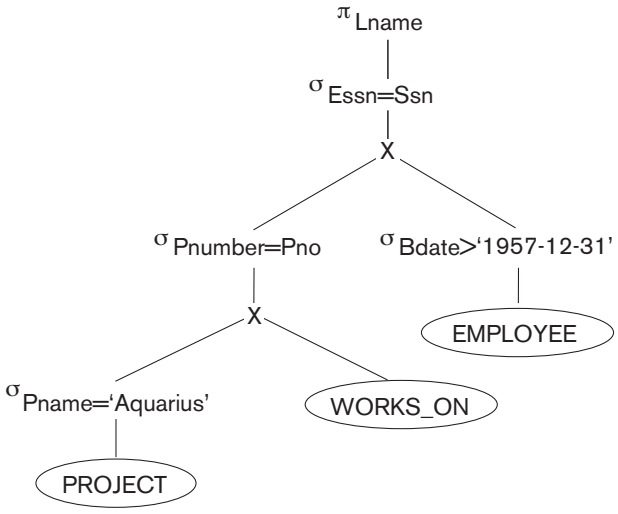
```
Q:  SELECT  Lname
    FROM    EMPLOYEE, WORKS_ON, PROJECT
    WHERE   Pname='Aquarius' AND Pnumber=Pno AND Essn=Ssn
            AND Bdate > '1957-12-31';
```

The initial query tree for Q is shown in Figure 19.5(a). Executing this tree directly first creates a very large file containing the CARTESIAN PRODUCT of the entire EMPLOYEE, WORKS_ON, and PROJECT files. That is why the initial query tree is never executed, but is transformed into another equivalent tree that is efficient to

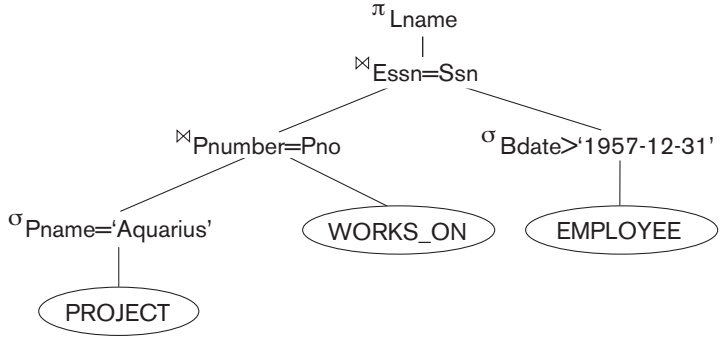
Figure 19.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.



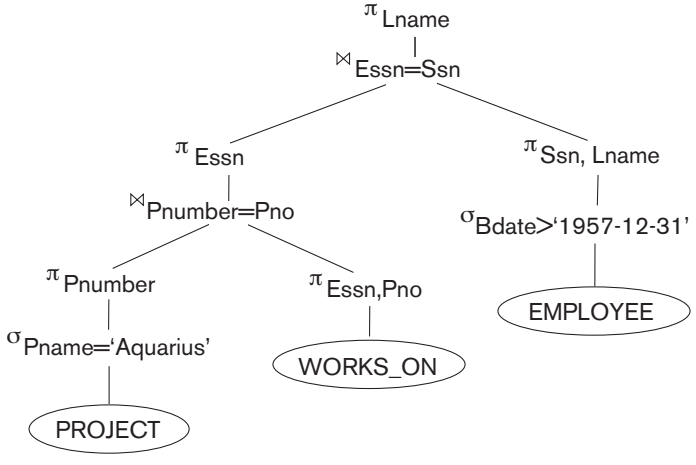
(c)


π Lname
σ Essn=Ssn
X
σ Pnumber=Pno
σ Bdate>'1957-12-31'
X
EMPLOYEE
σ Pname='Aquarius'
WORKS_ON
PROJECT

(d)


π Lname
⋈ Essn=Ssn
⋈ Pnumber=Pno
σ Bdate>'1957-12-31'
σ Pname='Aquarius'
WORKS_ON
EMPLOYEE
PROJECT

(e)


π Lname
⋈ Essn=Ssn
π Essn
π Ssn, Lname
⋈ Pnumber=Pno
σ Bdate>'1957-12-31'
π Pnumber
π Essn,Pno
EMPLOYEE
σ Pname='Aquarius'
WORKS_ON
PROJECT

execute. This particular 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 19.5(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 19.5(c). This uses the information 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 19.5(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 19.5(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 an equivalent 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. For query optimization purposes, 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 to be equivalent. In Section 3.1.2 we gave an alternative definition of *relation* that makes the 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 join 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 the 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 $(c_1 \text{ AND } c_2)$, 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 $\bowtie$ is replaced by a $\times$ operation.

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

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

   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 $A_{n+1}, ..., A_{n+k}$ of R and $B_{m+1}, ..., B_{m+p}$ of S are involved in the join condition c but are not in the projection list L, the operations commute as follows:

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

   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 standard rules from Boolean algebra (DeMorgan's laws):

NOT $(c_1$ **AND** $c_2) \equiv$ (**NOT** c_1) **OR** (**NOT** c_2)

NOT $(c_1$ **OR** $c_2) \equiv$ (**NOT** c_1) **AND** (**NOT** c_2)

Additional transformations discussed in Chapters 4, 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 a final tree that is more efficient to execute (in most cases). The algorithm will lead to transformations similar to those discussed in our example in Figure 19.5. 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. If the condition involves attributes from *only one table*, which means that it represents a *selection condition*, the operation is moved all the way to the leaf node that represents this table. If the condition involves attributes from *two tables*, which means that it represents a *join condition*, the condition is moved to a location down the tree after the two tables are combined.
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

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

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 19.5(b) shows the tree in Figure 19.5(a) after applying steps 1 and 2 of the algorithm; Figure 19.5(c) shows the tree after step 3; Figure 19.5(d) after step 4; and Figure 19.5(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. 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—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. The latter rule is accomplished through reordering the leaf nodes of the tree among themselves while avoiding Cartesian products, and adjusting the rest of the tree appropriately.

19.7.3 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 4, whose corresponding relational algebra expression is

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

[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.

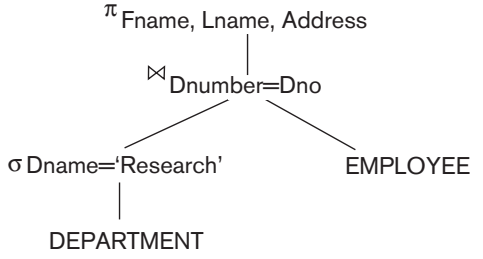


Figure 19.6
A query tree for query Q1.

The query tree is shown in Figure 19.6. To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation on DEPARTMENT (assuming one exists), a single-loop join algorithm that loops over the records in the result of the SELECT operation on DEPARTMENT for the join operation (assuming an index exists on the Dno attribute of EMPLOYEE), and a scan of the JOIN result for input to the PROJECT operator. Additionally, the approach taken for executing the query may specify a materialized or a pipelined evaluation, although in general a pipelined evaluation is preferred whenever feasible.

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 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.

19.8 Using Selectivity and Cost Estimates in Query Optimization

A query optimizer does not depend solely on heuristic rules; it also estimates and compares the costs of executing a query using different execution strategies and algorithms, and it then chooses the strategy with the *lowest cost estimate*. For this approach to work, accurate *cost estimates* are required so that different strategies can be compared fairly and realistically. In addition, the optimizer 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 19.1 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.

This approach is generally referred to as **cost-based query optimization**.[19] 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 (absolute best) one. In Section 19.8.1 we discuss the components of query execution cost. In Section 19.8.2 we discuss the type of information needed in cost functions. This information is kept in the DBMS catalog. In Section 19.8.3 we give examples of cost functions for the SELECT operation, and in Section 19.8.4 we discuss cost functions for two-way JOIN operations. Section 19.8.5 discusses multiway joins, and Section 19.8.6 gives an example.

19.8.1 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 transferring (reading and writing) data blocks between secondary disk storage and main memory buffers. This is also known as *disk I/O (input/output) cost*. The cost of searching for records in a disk file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.
2. **Disk storage cost.** This is the cost of storing on disk any intermediate files that are generated by an execution strategy for the query.
3. **Computation cost.** This is the cost of performing in-memory operations on the records within the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join or a sort operation, and performing computations on field values. This is also known as *CPU (central processing unit) cost*.
4. **Memory usage cost.** This is the cost pertaining to the number of main memory buffers needed during query execution.
5. **Communication cost.** This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated. In distributed databases (see Chapter 25), it would also include the cost of transferring tables and results among various computers during query evaluation.

For large databases, the main emphasis is often on minimizing the access cost to secondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk

[19]This approach was first used in the optimizer for the SYSTEM R in an experimental DBMS developed at IBM (Selinger et al. 1979).

and main memory buffers. For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the emphasis is on minimizing computation cost. In distributed databases, where many sites are involved (see Chapter 25), communication cost must be minimized also. It is difficult to include all the cost components in a (weighted) cost function because of the difficulty of assigning suitable weights to the cost components. That is why some cost functions consider a single factor only—disk access. In the next section we discuss some of the information that is needed for formulating cost functions.

19.8.2 Catalog Information Used in Cost Functions

To estimate the costs of various execution strategies, we must keep track of any information that is needed for the cost functions. This information may be stored in the DBMS catalog, where it is accessed by the query optimizer. First, we must know the size of each file. For a file whose records are all of the same type, the **number of records (tuples)** (***r***), the (average) **record size** (***R***), and the **number of file blocks** (***b***) (or close estimates of them) are needed. The **blocking factor** (***bfr***) for the file may also be needed. We must also keep track of the *primary file organization* for each file. The primary file organization records may be *unordered*, *ordered* by an attribute with or without a primary or clustering index, or *hashed* (static hashing or one of the dynamic hashing methods) on a key attribute. Information is also kept on all primary, secondary, or clustering indexes and their indexing attributes. The **number of levels** (***x***) of each multilevel index (primary, secondary, or clustering) is needed for cost functions that estimate the number of block accesses that occur during query execution. In some cost functions the **number of first-level index blocks** (b_{I1}) is needed.

Another important parameter is the **number of distinct values** (***d***) of an attribute and the attribute **selectivity** (***sl***), which is the fraction of records satisfying an equality condition on the attribute. This allows estimation of the **selection cardinality** (***s*** = ***sl*** * ***r***) of an attribute, which is the *average* number of records that will satisfy an equality selection condition on that attribute. For a *key attribute,* $d = r$, $sl = 1/r$ and $s = 1$. For a *nonkey attribute,* by making an assumption that the d distinct values are uniformly distributed among the records, we estimate $sl = (1/d)$ and so $s = (r/d)$.[20]

Information such as the number of index levels is easy to maintain because it does not change very often. However, other information may change frequently; for example, the number of records r in a file changes every time a record is inserted or deleted. The query optimizer will need reasonably close but not necessarily completely up-to-the-minute values of these parameters for use in estimating the cost of various execution strategies.

For a nonkey attribute with d distinct values, it is often the case that the records are not uniformly distributed among these values. For example, suppose that a company has 5 departments numbered 1 through 5, and 200 employees who are distrib-

[20]More accurate optimizers store *histograms* of the distribution of records over the data values for an attribute.

uted among the departments as follows: (1, 5), (2, 25), (3, 70), (4, 40), (5, 60). In such cases, the optimizer can store a **histogram** that reflects the distribution of employee records over different departments in a table with the two attributes (Dno, Selectivity), which would contain the following values for our example: (1, 0.025), (2, 0.125), (3, 0.35), (4, 0.2), (5, 0.3). The selectivity values stored in the histogram can also be estimates if the employee table changes frequently.

In the next two sections we examine how some of these parameters are used in cost functions for a cost-based query optimizer.

19.8.3 Examples of Cost Functions for SELECT

We now give cost functions for the selection algorithms S1 to S8 discussed in Section 19.3.1 in terms of *number of block transfers* between memory and disk. Algorithm S9 involves an intersection of record pointers after they have been retrieved by some other means, such as algorithm S6, and so the cost function will be based on the cost for S6. These cost functions are estimates that ignore computation time, storage cost, and other factors. The cost for method S*i* is referred to as C_{Si} block accesses.

- **S1—Linear search (brute force) approach.** We search all the file blocks to retrieve all records satisfying the selection condition; hence, $C_{S1a} = b$. For an *equality condition on a key attribute*, only half the file blocks are searched *on the average* before finding the record, so a rough estimate for $C_{S1b} = (b/2)$ if the record is found; if no record is found that satisfies the condition, $C_{S1b} = b$.
- **S2—Binary search.** This search accesses approximately $C_{S2} = \log_2 b + \lceil (s/bfr) \rceil - 1$ file blocks. This reduces to $\log_2 b$ if the equality condition is on a unique (key) attribute, because $s = 1$ in this case.
- **S3a—Using a primary index to retrieve a single record.** For a primary index, retrieve one disk block at each index level, plus one disk block from the data file. Hence, the cost is one more disk block than the number of index levels: $C_{S3a} = x + 1$.
- **S3b—Using a hash key to retrieve a single record.** For hashing, only one disk block needs to be accessed in most cases. The cost function is approximately $C_{S3b} = 1$ for static hashing or linear hashing, and it is 2 disk block accesses for extendible hashing (see Section 17.8).
- **S4—Using an ordering index to retrieve multiple records.** If the comparison condition is >, >=, <, or <= on a key field with an ordering index, roughly half the file records will satisfy the condition. This gives a cost function of $C_{S4} = x + (b/2)$. This is a very rough estimate, and although it may be correct on the average, it may be quite inaccurate in individual cases. A more accurate estimate is possible if the distribution of records is stored in a histogram.
- **S5—Using a clustering index to retrieve multiple records.** One disk block is accessed at each index level, which gives the address of the first file disk block in the cluster. Given an equality condition on the indexing attribute, *s*

records will satisfy the condition, where s is the selection cardinality of the indexing attribute. This means that $\lceil (s/bfr) \rceil$ file blocks will be in the cluster of file blocks that hold all the selected records, giving $C_{S5} = x + \lceil (s/bfr) \rceil$.

- **S6—Using a secondary (B^+-tree) index.** For a secondary index on a key (unique) attribute, the cost is $x + 1$ disk block accesses. For a secondary index on a nonkey (nonunique) attribute, s records will satisfy an equality condition, where s is the selection cardinality of the indexing attribute. However, because the index is nonclustering, each of the records may reside on a different disk block, so the (worst case) cost estimate is $C_{S6a} = x + 1 + s$. The additional *1* is to account for the disk block that contains the record pointers after the index is searched (see Figure 18.5). If the comparison condition is >, >=, <, or <= and half the file records are assumed to satisfy the condition, then (very roughly) half the first-level index blocks are accessed, plus half the file records via the index. The cost estimate for this case, approximately, is $C_{S6b} = x + (b_{I1}/2) + (r/2)$. The $r/2$ factor can be refined if better selectivity estimates are available through a histogram. The latter method C_{S6b} can be very costly.
- **S7—Conjunctive selection.** We can use either S1 or one of the methods S2 to S6 discussed above. In the latter case, we use one condition to retrieve the records and then check in the main memory buffers whether each retrieved record satisfies the remaining conditions in the conjunction. If multiple indexes exist, the search of each index can produce a set of record pointers (record ids) in the main memory buffers. The intersection of the sets of record pointers (referred to in S9) can be computed in main memory, and then the resulting records are retrieved based on their record ids.
- **S8—Conjunctive selection using a composite index.** Same as S3*a*, S5, or S6*a*, depending on the type of index.

Example of Using the Cost Functions. In a query optimizer, it is common to enumerate the various possible strategies for executing a query and to estimate the costs for different strategies. An optimization technique, such as dynamic programming, may be used to find the optimal (least) cost estimate efficiently, without having to consider all possible execution strategies. We do not discuss optimization algorithms here; rather, we use a simple example to illustrate how cost estimates may be used. Suppose that the EMPLOYEE file in Figure 3.5 has $r_E = 10{,}000$ records stored in $b_E = 2000$ disk blocks with blocking factor $bfr_E = 5$ records/block and the following access paths:

1. A clustering index on Salary, with levels $x_{\text{Salary}} = 3$ and average selection cardinality $s_{\text{Salary}} = 20$. (This corresponds to a selectivity of $sl_{\text{Salary}} = 0.002$).
2. A secondary index on the key attribute Ssn, with $x_{\text{Ssn}} = 4$ ($s_{\text{Ssn}} = 1$, $sl_{\text{Ssn}} = 0.0001$).
3. A secondary index on the nonkey attribute Dno, with $x_{\text{Dno}} = 2$ and first-level index blocks $b_{I1\text{Dno}} = 4$. There are $d_{\text{Dno}} = 125$ distinct values for Dno, so the selectivity of Dno is $sl_{\text{Dno}} = (1/d_{\text{Dno}}) = 0.008$, and the selection cardinality is $s_{\text{Dno}} = (r_E * sl_{\text{Dno}}) = (r_E/d_{\text{Dno}}) = 80$.

4. A secondary index on Sex, with $x_{\text{Sex}} = 1$. There are $d_{\text{Sex}} = 2$ values for the Sex attribute, so the average selection cardinality is $s_{\text{Sex}} = (r_E/d_{\text{Sex}}) = 5000$. (Note that in this case, a histogram giving the percentage of male and female employees may be useful, unless they are approximately equal.)

We illustrate the use of cost functions with the following examples:

OP1: $\sigma_{\text{Ssn='123456789'}}(\text{EMPLOYEE})$

OP2: $\sigma_{\text{Dno}>5}(\text{EMPLOYEE})$

OP3: $\sigma_{\text{Dno}=5}(\text{EMPLOYEE})$

OP4: $\sigma_{\text{Dno}=5 \text{ AND SALARY}>30000 \text{ AND Sex='F'}}(\text{EMPLOYEE})$

The cost of the brute force (linear search or file scan) option S1 will be estimated as $C_{S1a} = b_E = 2000$ (for a selection on a nonkey attribute) or $C_{S1b} = (b_E/2) = 1000$ (average cost for a selection on a key attribute). For OP1 we can use either method *S*1 or method *S6a*; the cost estimate for *S6a* is $C_{S6a} = x_{\text{Ssn}} + 1 = 4 + 1 = 5$, and it is chosen over method *S*1, whose average cost is $C_{S1b} = 1000$. For OP2 we can use either method *S*1 (with estimated cost $C_{S1a} = 2000$) or method S6*b* (with estimated cost $C_{S6b} = x_{\text{Dno}} + (b_{I1\text{Dno}}/2) + (r_E/2) = 2 + (4/2) + (10{,}000/2) = 5004$), so we choose the linear search approach for OP2. For OP3 we can use either method *S*1 (with estimated cost $C_{S1a} = 2000$) or method *S6a* (with estimated cost $C_{S6a} = x_{\text{Dno}} + s_{\text{Dno}} = 2 + 80 = 82$), so we choose method *S6a*.

Finally, consider OP4, which has a conjunctive selection condition. We need to estimate the cost of using any one of the three components of the selection condition to retrieve the records, plus the linear search approach. The latter gives cost estimate $C_{S1a} = 2000$. Using the condition (Dno = 5) first gives the cost estimate $C_{S6a} = 82$. Using the condition (Salary > 30,000) first gives a cost estimate $C_{S4} = x_{\text{Salary}} + (b_E/2) = 3 + (2000/2) = 1003$. Using the condition (Sex = 'F') first gives a cost estimate $C_{S6a} = x_{\text{Sex}} + s_{\text{Sex}} = 1 + 5000 = 5001$. The optimizer would then choose method *S6a* on the secondary index on Dno because it has the lowest cost estimate. The condition (Dno = 5) is used to retrieve the records, and the remaining part of the conjunctive condition (Salary > 30,000 AND Sex = 'F') is checked for each selected record after it is retrieved into memory. Only the records that satisfy these additional conditions are included in the result of the operation.

19.8.4 Examples of Cost Functions for JOIN

To develop reasonably accurate cost functions for JOIN operations, we need to have an estimate for the size (number of tuples) of the file that results *after* the JOIN operation. This is usually kept as a ratio of the size (number of tuples) of the resulting join file to the size of the CARTESIAN PRODUCT file, if both are applied to the same input files, and it is called the **join selectivity** (***js***). If we denote the number of tuples of a relation *R* by |*R*|, we have:

$$js = |(R \bowtie_c S)| / |(R \times S)| = |(R \bowtie_c S)| / (|R| * |S|)$$

If there is no join condition *c,* then *js* = 1 and the join is the same as the CARTESIAN PRODUCT. If no tuples from the relations satisfy the join condition, then *js* = 0. In

general, $0 \le js \le 1$. For a join where the condition c is an equality comparison $R.A = S.B$, we get the following two special cases:

1. If A is a key of R, then $|(R \bowtie_c S)| \le |S|$, so $js \le (1/|R|)$. This is because each record in file S will be joined with at most one record in file R, since A is a key of R. A special case of this condition is when attribute B is a *foreign key* of S that references the *primary key* A of R. In addition, if the foreign key B has the NOT NULL constraint, then $js = (1/|R|)$, and the result file of the join will contain $|S|$ records.
2. If B is a key of S, then $|(R \bowtie_c S)| \le |R|$, so $js \le (1/|S|)$.

Having an estimate of the join selectivity for commonly occurring join conditions enables the query optimizer to estimate the size of the resulting file after the join operation, given the sizes of the two input files, by using the formula $|(R \bowtie_c S)| = js * |R| * |S|$. We can now give some sample *approximate* cost functions for estimating the cost of some of the join algorithms given in Section 19.3.2. The join operations are of the form:

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

where A and B are domain-compatible attributes of R and S, respectively. Assume that R has b_R blocks and that S has b_S blocks:

- **J1—Nested-loop join.** Suppose that we use R for the outer loop; then we get the following cost function to estimate the number of block accesses for this method, assuming *three memory buffers.* We assume that the blocking factor for the resulting file is bfr_{RS} and that the join selectivity is known:

  $C_{J1} = b_R + (b_R * b_S) + ((js * |R| * |S|)/bfr_{RS})$

  The last part of the formula is the cost of writing the resulting file to disk. This cost formula can be modified to take into account different numbers of memory buffers, as presented in Section 19.3.2. If n_B main memory buffers are available to perform the join, the cost formula becomes:

  $C_{J1} = b_R + (\lceil b_R/(n_B - 2) \rceil * b_S) + ((js * |R| * |S|)/bfr_{RS})$
- **J2—Single-loop join (using an access structure to retrieve the matching record(s)).** If an index exists for the join attribute B of S with index levels x_B, we can retrieve each record s in R and then use the index to retrieve all the matching records t from S that satisfy $t[B] = s[A]$. The cost depends on the type of index. For a secondary index where s_B is the selection cardinality for the join attribute B of S,[21] we get:

  $C_{J2a} = b_R + (|R| * (x_B + 1 + s_B)) + ((js * |R| * |S|)/bfr_{RS})$

  For a clustering index where s_B is the selection cardinality of B, we get

  $C_{J2b} = b_R + (|R| * (x_B + (s_B/bfr_B))) + ((js * |R| * |S|)/bfr_{RS})$

  For a primary index, we get

[21] *Selection cardinality* was defined as the average number of records that satisfy an equality condition on an attribute, which is the average number of records that have the same value for the attribute and hence will be joined to a single record in the other file.

$C_{J2c} = b_R + (|R| * (x_B + 1)) + ((js * |R| * |S|)/bfr_{RS})$

If a hash key exists for one of the two join attributes—say, B of S—we get

$C_{J2d} = b_R + (|R| * h) + ((js * |R| * |S|)/bfr_{RS})$

where $h \geq 1$ is the average number of block accesses to retrieve a record, given its hash key value. Usually, h is estimated to be *1* for static and linear hashing and *2* for extendible hashing.

- **J3—Sort-merge join.** If the files are already sorted on the join attributes, the cost function for this method is

  $C_{J3a} = b_R + b_S + ((js * |R| * |S|)/bfr_{RS})$

  If we must sort the files, the cost of sorting must be added. We can use the formulas from Section 19.2 to estimate the sorting cost.

Example of Using the Cost Functions. Suppose that we have the EMPLOYEE file described in the example in the previous section, and assume that the DEPARTMENT file in Figure 3.5 consists of $r_D = 125$ records stored in $b_D = 13$ disk blocks. Consider the following two join operations:

OP6: EMPLOYEE $\bowtie_{\text{Dno=Dnumber}}$ DEPARTMENT
OP7: DEPARTMENT $\bowtie_{\text{Mgr\_ssn=Ssn}}$ EMPLOYEE

Suppose that we have a primary index on Dnumber of DEPARTMENT with $x_{\text{Dnumber}} = 1$ level and a secondary index on Mgr_ssn of DEPARTMENT with selection cardinality $s_{\text{Mgr\_ssn}} = 1$ and levels $x_{\text{Mgr\_ssn}} = 2$. Assume that the join selectivity for OP6 is $js_{\text{OP6}} = (1/|\text{DEPARTMENT}|) = 1/125$ because Dnumber is a key of DEPARTMENT. Also assume that the blocking factor for the resulting join file is $bfr_{\text{ED}} = 4$ records per block. We can estimate the worst-case costs for the JOIN operation OP6 using the applicable methods J1 and J2 as follows:

1. Using method J1 with EMPLOYEE as outer loop:

   $C_{J1} = b_E + (b_E * b_D) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED})$
   $= 2000 + (2000 * 13) + (((1/125) * 10{,}000 * 125)/4) = 30{,}500$

2. Using method J1 with DEPARTMENT as outer loop:

   $C_{J1} = b_D + (b_E * b_D) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED})$
   $= 13 + (13 * 2000) + (((1/125) * 10{,}000 * 125/4) = 28{,}513$

3. Using method J2 with EMPLOYEE as outer loop:

   $C_{J2c} = b_E + (r_E * (x_{\text{Dnumber}} + 1)) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED}$
   $= 2000 + (10{,}000 * 2) + (((1/125) * 10{,}000 * 125/4) = 24{,}500$

4. Using method J2 with DEPARTMENT as outer loop:

   $C_{J2a} = b_D + (r_D * (x_{\text{Dno}} + s_{\text{Dno}})) + ((js_{\text{OP6}} * r_E * r_D)/bfr_{ED})$
   $= 13 + (125 * (2 + 80)) + (((1/125) * 10{,}000 * 125/4) = 12{,}763$

Case 4 has the lowest cost estimate and will be chosen. Notice that in case 2 above, if 15 memory buffers (or more) were available for executing the join instead of just 3, 13 of them could be used to hold the entire DEPARTMENT relation (outer loop

relation) in memory, one could be used as buffer for the result, and one would be used to hold one block at a time of the EMPLOYEE file (inner loop file), and the cost for case 2 could be drastically reduced to just $b_E + b_D + ((\ js_{\mathrm{OP6}} * r_{\mathrm{E}} * r_{\mathrm{D}})/bfr_{\mathrm{ED}})$ or 4,513, as discussed in Section 19.3.2. If some other number of main memory buffers was available, say $n_{\mathrm{B}} = 10$, then the cost for case 2 would be calculated as follows, which would also give better performance than case 4:

$$\begin{aligned} C_{\mathrm{J1}} &= b_D + (\lceil b_D/(n_B - 2)\rceil * b_E) + ((js * |R| * |S|)/bfr_{RS}) \\ &= 13 + (\lceil 13/8\rceil * 2000) + (((1/125) * 10{,}000 * 125/4) = 28{,}513 \\ &= 13 + (2 * 2000) + 2500 = 6{,}513 \end{aligned}$$

As an exercise, the reader should perform a similar analysis for OP7.

19.8.5 Multiple Relation Queries and JOIN Ordering

The algebraic transformation rules in Section 19.7.2 include a commutative rule and an associative rule for the join operation. With these rules, many equivalent join expressions can be produced. As a result, the number of alternative query trees grows very rapidly as the number of joins in a query increases. A query that joins n relations will often have $n - 1$ join operations, and hence can have a large number of different join orders. Estimating the cost of every possible join tree for a query with a large number of joins will require a substantial amount of time by the query optimizer. Hence, some pruning of the possible query trees is needed. Query optimizers typically limit the structure of a (join) query tree to that of left-deep (or right-deep) trees. A **left-deep tree** is a binary tree in which the right child of each nonleaf node is always a base relation. The optimizer would choose the particular left-deep tree with the lowest estimated cost. Two examples of left-deep trees are shown in Figure 19.7. (Note that the trees in Figure 19.5 are also left-deep trees.)

With left-deep trees, the right child is considered to be the inner relation when executing a nested-loop join, or the probing relation when executing a single-loop join. One advantage of left-deep (or right-deep) trees is that they are amenable to pipelining, as discussed in Section 19.6. For instance, consider the first left-deep tree in Figure 19.7 and assume that the join algorithm is the single-loop method; in this case, a disk page of tuples of the outer relation is used to probe the inner relation for

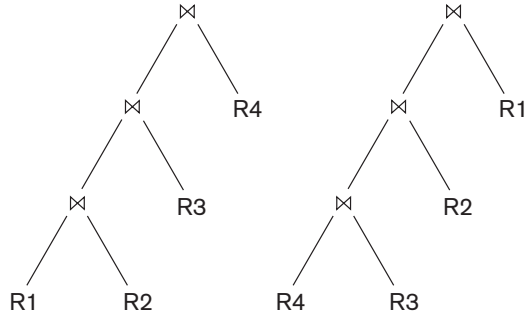


Figure 19.7
Two left-deep (JOIN) query trees.

matching tuples. As resulting tuples (records) are produced from the join of R1 and R2, they can be used to probe R3 to locate their matching records for joining. Likewise, as resulting tuples are produced from this join, they could be used to probe R4. Another advantage of left-deep (or right-deep) trees is that having a base relation as one of the inputs of each join allows the optimizer to utilize any access paths on that relation that may be useful in executing the join.

If materialization is used instead of pipelining (see Sections 19.6 and 19.7.3), the join results could be materialized and stored as temporary relations. The key idea from the optimizer's standpoint with respect to join ordering is to find an ordering that will reduce the size of the temporary results, since the temporary results (pipelined or materialized) are used by subsequent operators and hence affect the execution cost of those operators.

19.8.6 Example to Illustrate Cost-Based Query Optimization

We will consider query Q2 and its query tree shown in Figure 19.4(a) to illustrate cost-based query optimization:

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

Suppose we have the information about the relations shown in Figure 19.8. The LOW_VALUE and HIGH_VALUE statistics have been normalized for clarity. The tree in Figure 19.4(a) is assumed to represent the result of the algebraic heuristic optimization process and the start of cost-based optimization (in this example, we assume that the heuristic optimizer does not push the projection operations down the tree).

The first cost-based optimization to consider is join ordering. As previously mentioned, we assume the optimizer considers only left-deep trees, so the potential join orders—without CARTESIAN PRODUCT—are:

1. PROJECT ⋈ DEPARTMENT ⋈ EMPLOYEE
2. DEPARTMENT ⋈ PROJECT ⋈ EMPLOYEE
3. DEPARTMENT ⋈ EMPLOYEE ⋈ PROJECT
4. EMPLOYEE ⋈ DEPARTMENT ⋈ PROJECT

Assume that the selection operation has already been applied to the PROJECT relation. If we assume a materialized approach, then a new temporary relation is created after each join operation. To examine the cost of join order (1), the first join is between PROJECT and DEPARTMENT. Both the join method and the access methods for the input relations must be determined. Since DEPARTMENT has no index according to Figure 19.8, the only available access method is a table scan (that is, a linear search). The PROJECT relation will have the selection operation performed before the join, so two options exist: table scan (linear search) or utilizing its PROJ_PLOC index, so the optimizer must compare their estimated costs.

Figure 19.8
Sample statistical information for relations in Q2. (a) Column information. (b) Table information. (c) Index information.

(a)

Table_name	Column_name	Num_distinct	Low_value	High_value
PROJECT	Plocation	200	1	200
PROJECT	Pnumber	2000	1	2000
PROJECT	Dnum	50	1	50
DEPARTMENT	Dnumber	50	1	50
DEPARTMENT	Mgr_ssn	50	1	50
EMPLOYEE	Ssn	10000	1	10000
EMPLOYEE	Dno	50	1	50
EMPLOYEE	Salary	500	1	500

(b)

Table_name	Num_rows	Blocks
PROJECT	2000	100
DEPARTMENT	50	5
EMPLOYEE	10000	2000

(c)

Index_name	Uniqueness	Blevel*	Leaf_blocks	Distinct_keys
PROJ_PLOC	NONUNIQUE	1	4	200
EMP_SSN	UNIQUE	1	50	10000
EMP_SAL	NONUNIQUE	1	50	500

*Blevel is the number of levels without the leaf level.

The statistical information on the PROJ_PLOC index (see Figure 19.8) shows the number of index levels $x = 2$ (root plus leaf levels). The index is nonunique (because Plocation is not a key of PROJECT), so the optimizer assumes a uniform data distribution and estimates the number of record pointers for each Plocation value to be 10. This is computed from the tables in Figure 19.8 by multiplying Selectivity * Num_rows, where Selectivity is estimated by 1/Num_distinct. So the cost of using the index and accessing the records is estimated to be 12 block accesses (2 for the index and 10 for the data blocks). The cost of a table scan is estimated to be 100 block accesses, so the index access is more efficient as expected.

In the materialized approach, a temporary file TEMP1 of size 1 block is created to hold the result of the selection operation. The file size is calculated by determining the blocking factor using the formula Num_rows/Blocks, which gives 2000/100 or 20 rows per block. Hence, the 10 records selected from the PROJECT relation will fit

into a single block. Now we can compute the estimated cost of the first join. We will consider only the nested-loop join method, where the outer relation is the temporary file, TEMP1, and the inner relation is DEPARTMENT. Since the entire TEMP1 file fits in the available buffer space, we need to read each of the DEPARTMENT table's five blocks only once, so the join cost is six block accesses plus the cost of writing the temporary result file, TEMP2. The optimizer would have to determine the size of TEMP2. Since the join attribute Dnumber is the key for DEPARTMENT, any Dnum value from TEMP1 will join with at most one record from DEPARTMENT, so the number of rows in TEMP2 will be equal to the number of rows in TEMP1, which is 10. The optimizer would determine the record size for TEMP2 and the number of blocks needed to store these 10 rows. For brevity, assume that the blocking factor for TEMP2 is five rows per block, so a total of two blocks are needed to store TEMP2.

Finally, the cost of the last join needs to be estimated. We can use a single-loop join on TEMP2 since in this case the index EMP_SSN (see Figure 19.8) can be used to probe and locate matching records from EMPLOYEE. Hence, the join method would involve reading in each block of TEMP2 and looking up each of the five Mgr_ssn values using the EMP_SSN index. Each index lookup would require a root access, a leaf access, and a data block access (x+1, where the number of levels x is 2). So, 10 lookups require 30 block accesses. Adding the two block accesses for TEMP2 gives a total of 32 block accesses for this join.

For the final projection, assume pipelining is used to produce the final result, which does not require additional block accesses, so the total cost for join order (1) is estimated as the sum of the previous costs. The optimizer would then estimate costs in a similar manner for the other three join orders and choose the one with the lowest estimate. We leave this as an exercise for the reader.

19.9 Overview of Query Optimization in Oracle

The Oracle DBMS[22] provides two different approaches to query optimization: rule-based and cost-based. With the rule-based approach, the optimizer chooses execution plans based on heuristically ranked operations. Oracle maintains a table of 15 ranked access paths, where a lower ranking implies a more efficient approach. The access paths range from table access by ROWID (the most efficient)—where ROWID specifies the record's physical address that includes the data file, data block, and row offset within the block—to a full table scan (the least efficient)—where all rows in the table are searched by doing multiblock reads. However, the rule-based approach is being phased out in favor of the cost-based approach, where the optimizer examines alternative access paths and operator algorithms and chooses the execution plan with the lowest estimated cost. The estimated query cost is proportional to the expected elapsed time needed to execute the query with the given execution plan.

[22]The discussion in this section is primarily based on version 7 of Oracle. More optimization techniques have been added to subsequent versions.