chapter 19

Query Optimization

In this chapter,[1] we will assume that the reader is already familiar with the strategies for query processing in relational DBMSs that we discussed in the previous chapter. The goal of query optimization is to select the best possible strategy for query evaluation. As we said before, the term *optimization* is a misnomer because the chosen execution plan may not always be the most optimal plan possible. The primary goal is to arrive at the most efficient and cost-effective plan using the available information about the schema and the content of relations involved, and to do so in a reasonable amount of time. Thus a proper way to describe **query optimization** would be that it is an activity conducted by a query optimizer in a DBMS to select the best available strategy for executing the query.

This chapter is organized as follows. In Section 19.1 we describe the notation for mapping of the queries from SQL into query trees and graphs. Most RDBMSs use an internal representation of the query as a tree. We present heuristics to transform the query into a more efficient equivalent form followed by a general procedure for applying those heuristics. In Section 19.2, we discuss the conversion of queries into execution plans. We discuss nested subquery optimization. We also present examples of query transformation in two cases: merging of views in Group By queries and transformation of Star Schema queries that arise in data warehouses. We also briefly discuss materialized views. Section 19.3 is devoted to a discussion of selectivity and result-size estimation and presents a cost-based approach to optimization. We revisit the information in the system catalog that we presented in Section 18.3.4 earlier and present histograms. Cost models for selection and join operation are presented in Sections 19.4 and 19.5. We discuss the join ordering problem, which is a critical one, in some detail in Section 19.5.3. Section 19.6 presents an example of cost-based optimization. Section 19.7 discusses some additional issues related to

[1]The substantial contribution of Rafi Ahmed to this chapter is appreciated.

query optimization. Section 19.8 is devoted to a discussion of query optimization in data warehouses. Section 19.9 gives an overview of query optimization in Oracle. Section 19.10 briefly discusses semantic query optimization. We end the chapter with a summary in Section 19.11.

19.1 Query Trees and Heuristics for 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 19.1.1, we reiterate the query tree and query graph notations that we introduced earlier in the context of relational algebra and calculus in Sections 8.3.5 and 8.6.5, 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 19.1.2, 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 19.1.3 discusses the generation of query execution plans.

19.1.1 Notation for Query Trees and Query Graphs

A **query tree** is a tree data structure that corresponds to an extended relational algebra expression. It represents the input relations of the query as *leaf nodes* of the tree, and it 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.

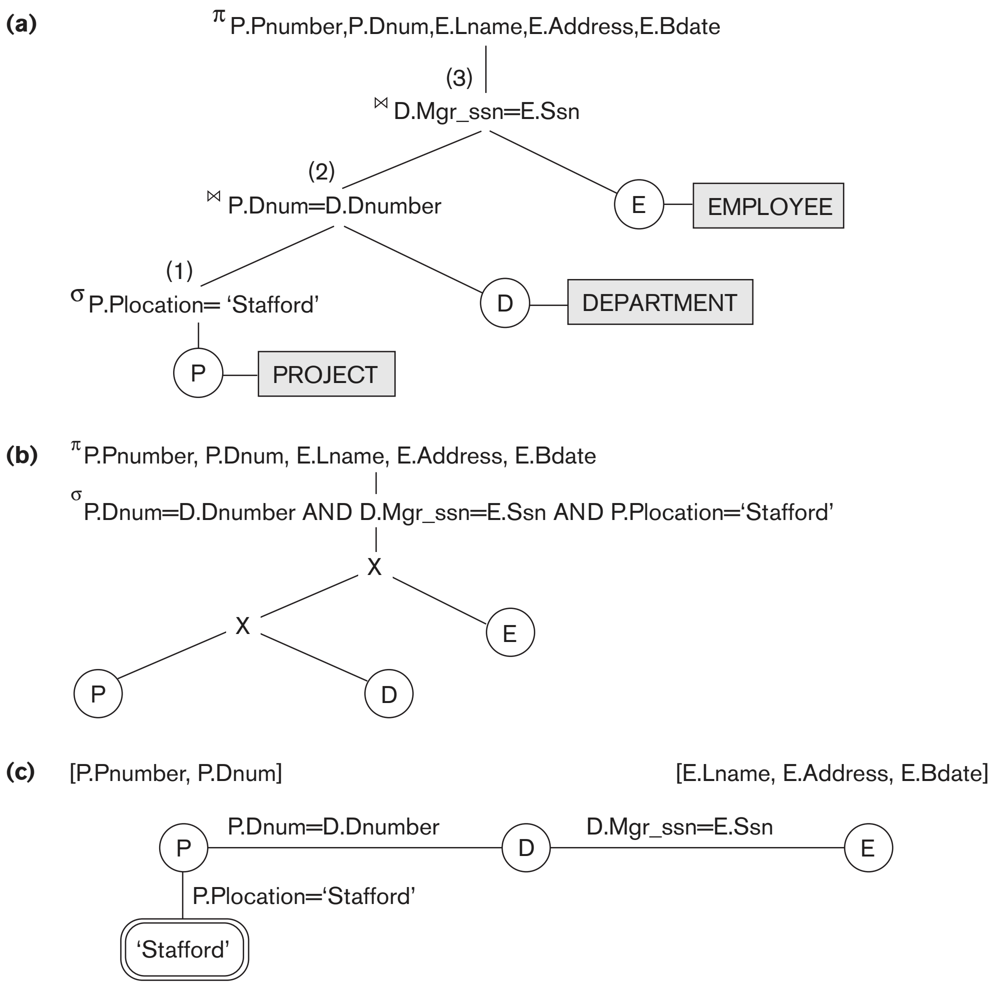

(a)
π P.Pnumber,P.Dnum,E.Lname,E.Address,E.Bdate
(3)
⋈ D.Mgr_ssn=E.Ssn
(2)
⋈ P.Dnum=D.Dnumber
E
EMPLOYEE
(1)
σ P.Plocation= 'Stafford'
D
DEPARTMENT
P
PROJECT
(b)
π P.Pnumber, P.Dnum, E.Lname, E.Address, E.Bdate
σ P.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation='Stafford'
X
X
E
P
D
(c)
[P.Pnumber, P.Dnum]
[E.Lname, E.Address, E.Bdate]
P.Dnum=D.Dnumber
D.Mgr_ssn=E.Ssn
P
D
E
P.Plocation='Stafford'
'Stafford'

Figure 19.1
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.

Figure 19.1(a) shows a query tree (the same as shown in Figure 6.9) for query Q2 in Chapters 6 to 8: For every project located in 'Stafford', retrieve the project number, the controlling department number, and the department manager's last name, address, and birthdate. This query is specified on the COMPANY relational schema in Figure 5.5 and corresponds to the following relational algebra expression:

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

This corresponds to the following SQL query:

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

In Figure 19.1(a), 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 19.1(a) must begin execution before node (2) because some resulting tuples of operation (1) must be available before we can begin executing operation (2). Similarly, node (2) must begin executing and producing results before node (3) can start execution, and so on.

As we can see, the query tree represents a specific order of operations for executing a query. A more neutral data structure for representation of a query is the **query graph** notation. Figure 19.1(c) (the same as shown in Figure 6.13) shows the query graph for query Q2. Relations in the query are represented by **relation nodes**, which are displayed as single circles. Constant values, typically from the query selection conditions, are represented by **constant nodes**, which are displayed as double circles or ovals. Selection and join conditions are represented by the graph **edges**, as shown in Figure 19.1(c). 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.[2] Although some optimization techniques were based on query graphs such as those originally in the INGRES DBMS, 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.

19.1.2 Heuristic Optimization of Query Trees

In general, many different relational algebra expressions—and hence many different query trees—can be **semantically equivalent**; that is, they can represent the *same query and produce the same results*.[3]

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 19.1(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

[2]Hence, a query graph corresponds to a *relational calculus* expression as shown in Section 8.6.5.

[3]The same query may also be stated in various ways in a high-level query language such as SQL (see Chapters 7 and 8).

the projection on the SELECT clause attributes. Such a **canonical query tree** represents a relational algebra expression that is *very inefficient if executed directly,* because of the CARTESIAN PRODUCT (×) operations. For example, if the PROJECT, DEPARTMENT, and EMPLOYEE relations had record sizes of 100, 50, and 150 bytes and contained 100, 20, and 5,000 tuples, respectively, the result of the CARTESIAN PRODUCT would contain 10 million tuples of record size 300 bytes each. However, this canonical query tree in Figure 19.1(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 extended 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 5.5: *Find the last names of employees born after 1957 who work on a project named 'Aquarius'.* This query can be specified in SQL as follows:

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

The initial query tree for Q is shown in Figure 19.2(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 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 19.2(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 19.2(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 as a selection with a JOIN operation, as shown in Figure 19.2(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 19.2(e). This reduces the attributes (columns) of the intermediate relations, whereas the SELECT operations reduce the number of tuples (records).

Figure 19.2
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.

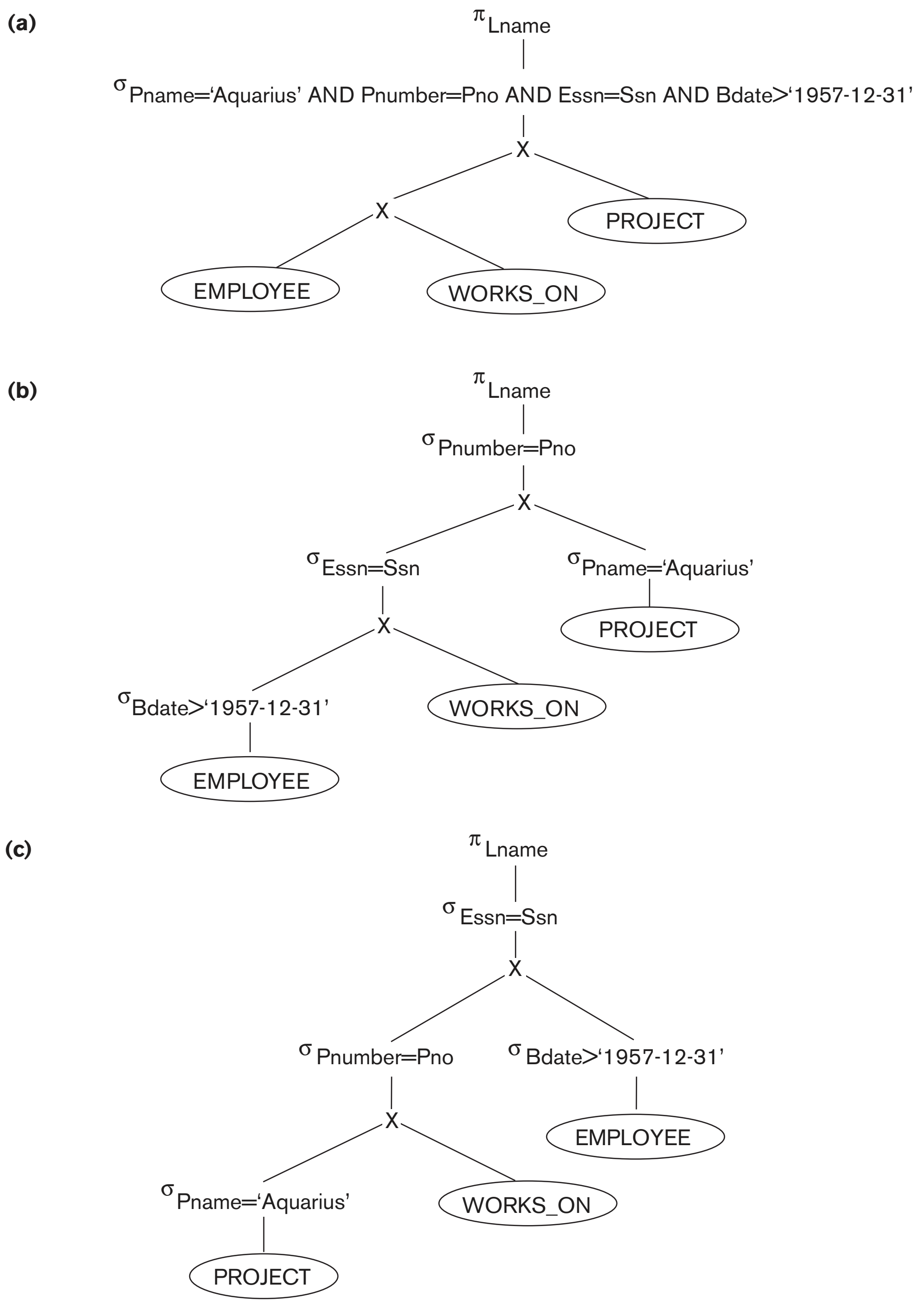
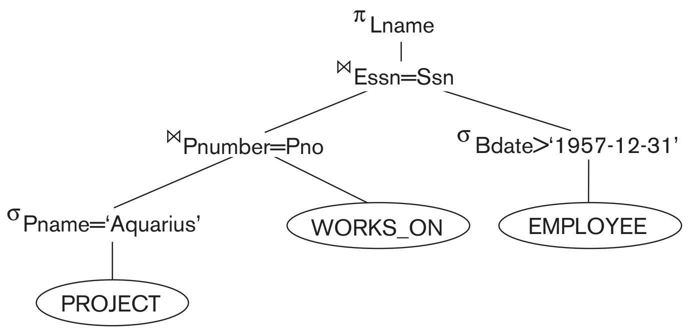
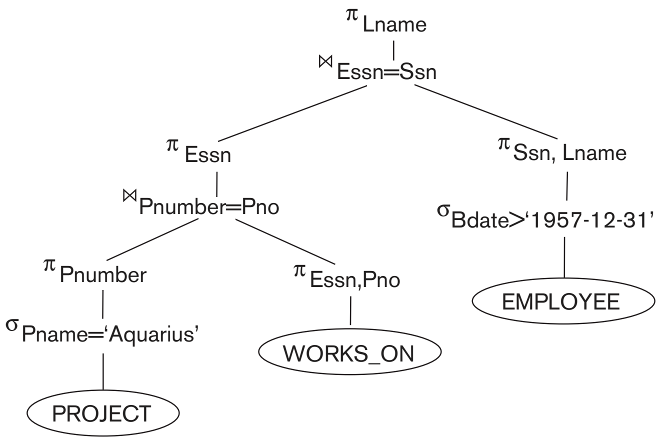


Figure 19.2 (continued)
Steps in converting a query tree during heuristic optimization. (d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations. (e) Moving PROJECT operations down the query tree.

(d)



(e)



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 5.1.2 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 ⋈ (and ×).** The join operation is commutative, as is the × 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 ⋈ (or ×).** If all the attributes in the selection condition c involve only the attributes of one of the relations being joined—say, R—the two operations can be commuted as follows:

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

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

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

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

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

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

   If the join condition c contains additional attributes not in L, these must be added to the projection list, and a final π operation is needed. For example, if attributes

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

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

For $\times$, 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 both occurrences of θ stand for the same operation that is 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 (σ, $\times$) sequence into $\bowtie$.** If the condition c of a σ that follows a $\times$ corresponds to a join condition, convert the (σ, $\times$) sequence into a $\bowtie$ as follows:

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

13. **Pushing σ in conjunction with set difference.**

    $\sigma_c (R - S) = \sigma_c (R) - \sigma_c (S)$

    However, σ may be applied to only one relation:

    $\sigma_c (R - S) = \sigma_c (R) - S$

14. **Pushing σ to only one argument in $\cap$.**

    If in the condition σ_c all attributes are from relation R, then:

    $\sigma_c (R \cap S) = \sigma_c (R) \cap S$

15. **Some trivial transformations.**

    If S is empty, then $R \cup S = R$

    If the condition c in σ_c is true for the entire R, then $\sigma_c (R) = R$.

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 (De Morgan'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 19.2. 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, 13, 14 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.[4] Another possibility is to define the most restrictive SELECT as the one with the smallest selectivity; this is more practical because estimates of selectivities are often available in the DBMS catalog. Second, make sure that the ordering of leaf nodes does not cause CARTESIAN PRODUCT operations; for example, if the two relations with the most restrictive SELECT do not have a direct join condition between them, it may be desirable to change the order of leaf nodes to avoid Cartesian products.[5]
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.

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

[5]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.

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

In our example, Figure 19.2(b) shows the tree in Figure 19.2(a) after applying steps 1 and 2 of the algorithm; Figure 19.2(c) shows the tree after step 3; Figure 19.2(d) after step 4; and Figure 19.2(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.

19.2 Choice of Query Execution Plans

19.2.1 Alternatives for Query Evaluation

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 7, whose corresponding relational algebra expression is

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

The query tree is shown in Figure 19.3. To convert this into an execution plan, the optimizer might choose an index search for the SELECT operation on DEPARTMENT (assuming one exists), an index-based nested-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

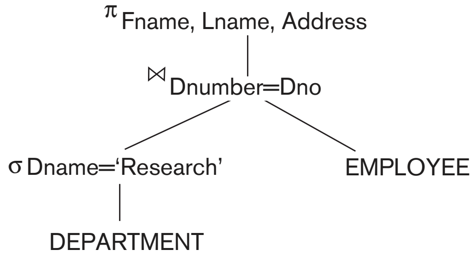


Figure 19.3
A query tree for query Q1.

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. We discussed pipelining as a strategy for query processing in Section 18.7. For example, as the selected tuples from DEPARTMENT are produced by the SELECT operation, they are placed in a buffer; the JOIN operation algorithm then consumes 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.

We discussed in Section 19.1 the possibility of converting query trees into equivalent trees so that the evaluation of the query is more efficient in terms of its execution time and overall resources consumed. There are more elaborate transformations of queries that are possible to optimize, or rather to "improve." Transformations can be applied either in a heuristic-based or cost-based manner.

As we discussed in Sections 7.1.2 and 7.1.3, nested subqueries may occur in the WHERE clause as well as in the FROM clause of SQL queries. In the WHERE clause, if an inner block makes a reference to the relation used in the outer block, it is called a correlated nested query. When a query is used within the FROM clause to define a resulting or derived relation, which participates as a relation in the outer query, it is equivalent to a view. Both these types of nested subqueries are handled by the optimizer, which transforms them and rewrites the entire query. In the next two subsections, we consider these two variations of query transformation and rewriting with examples. We will call them nested subquery optimization and subquery (view) merging transformation. In Section 19.8, we revisit this topic in the context of data warehouses and illustrate star transformation optimizations.

19.2.2 Nested Subquery Optimization

We discussed nested queries in Section 7.1.2. Consider the query:

```
SELECT E1.Fname, E1.Lname
FROM EMLOYEE E1
WHERE E1.Salary = ( SELECT MAX (Salary)
                    FROM EMPLOYEE E2)
```

In the above nested query, there is a query block inside an outer query block. Evaluation of this query involves executing the nested query first, which yields a single value of the maximum salary M in the EMPLOYEE relation; then the outer block is simply executed with the selection condition Salary = M. The maximum salary could be obtained just from the highest value in the index on salary (if one exists) or from the catalog if it is up-to-date. The outer query is evaluated based on the same index. If no index exists, then linear search would be needed for both.

We discussed correlated nested SQL queries in Section 7.1.3. In a correlated subquery, the inner query contains a reference to the outer query via one or more variables. The subquery acts as a function that returns a set of values for each value of this variable or combination of variables.

Suppose in the database of Figure 5.5, we modify the DEPARTMENT relation as:

DEPARTMENT (Dnumber, Dname, Mgr_ssn, Mgr_start_date, Zipcode)

Consider the query:

```
SELECT Fname, Lname, Salary
FROM EMPLOYEE E
WHERE EXISTS ( SELECT *
               FROM DEPARTMENT D
               WHERE D.Dnumber = E.Dno AND D.Zipcode=30332);
```

In the above, the nested subquery takes the E.Dno, the department where the employee works, as a parameter and returns a true or false value as a function depending on whether the department is located in zip code 30332. The naïve strategy for evaluating the query is to evaluate the inner nested subquery for every tuple of the outer relation, which is inefficient. Wherever possible, SQL optimizer tries to convert queries with nested subqueries into a join operation. The join can then be evaluated with one of the options we considered in Section 18.4. The above query would be converted to

```
SELECT Fname, Lname, Salary
FROM EMPLOYEE E, DEPARTMENT D
WHERE WHERE D.Dnumber = E.Dno AND D.Zipcode=30332
```

The process of removing the nested query and converting the outer and inner query into one block is called **unnesting**. Here inner join is used, since D.Dnumber is unique and the join is an equi-join; this guarantees that a tuple from relation Employee will match with at most one tuple from relation Department. We showed in Chapter 7 that the query Q16, which has a subquery connected with the IN connector, was also unnested into a single block query involving a join. In general, queries involving a nested subquery connected by IN or ANY connector in SQL can always be converted into a single block query. Other techniques used include creation of temporary result tables from subqueries and using them in joins.

We repeat the example query shown in Section 18.1. (Note that the IN operator is equivalent to the =ANY operator.):

```
Q (SJ) :
SELECT COUNT(*)
FROM DEPARTMENT D
WHERE D.Dnumber IN ( SELECT E.Dno
                     FROM EMPLOYEE E
                     WHERE E.Salary > 200000)
```

In this case again, there are two options for the optimizer:

1. Evaluate the nested subquery for each outer tuple; it is inefficient to do so.
2. Unnest the subquery using **semi-join**, which is much more efficient than option 1. In Section 18.1, we used this alternative to introduce and define the semi-join operator. Note that for unnesting this subquery, which refers to expressing it as a single block, inner join *cannot* be used, since in inner join a tuple of DEPARTMENT may match more than one tuple of EMPLOYEE and thus produce wrong results. It is easy to see that a nested subquery acts as a **filter** and thus it cannot, unlike inner join, produce more rows than there are in the DEPARTMENT table. Semi-join simulates this behavior.

The process we described as **unnesting** is sometimes called **decorrelation**. We showed another example in Section 18.1 using the connector "NOT IN", which was converted into a single block query using the operation **anti-join**. Optimization of complex nested subqueries is difficult and requires techniques that can be quite involved. We illustrate two such techniques in Section 19.2.3 below. Unnesting is a powerful optimization technique and is used widely by SQL optimizers.

19.2.3 Subquery (View) Merging Transformation

There are instances where a subquery appears in the FROM clause of a query and amounts to including a derived relation, which is similar to a predefined view that is involved in the query. This FROM clause subquery is often referred to as an inline view. Sometimes, an actual view defined earlier as a separate query is used as one of the argument relations in a new query. In such cases, the transformation of the query can be referred to as a view-merging or subquery merging transformation. The techniques of view merging discussed here apply equally to both inline and predefined views,

Consider the following three relations:

EMP (<u>Ssn</u>, Fn, Ln, Dno)
DEPT (<u>Dno</u>, Dname, Dmgrname, Bldg_id)
BLDG (<u>Bldg_id,</u> No_storeys, Addr, Phone)

The meaning of the relations is self-explanatory; the last one represents buildings where departments are located; the phone refers to a phone number for the building lobby.

The following query uses an inline view in the FROM clause; it retrieves for employees named "John" the last name, address and phone number of building where they work:

```
SELECT E.Ln, V.Addr, V.Phone
FROM EMP E, ( SELECT D.Dno, D.Dname, B.Addr, B.Phone
              FROM DEPT D, BLDG B
              WHERE D.Bldg_id = B.Bldg_id ) V
WHERE V.Dno = E.Dno AND E.Fn = "John";
```

The above query joins the EMP table with a view called V that provides the address and phone of the building where the employee works. In turn, the view joins the two tables DEPT and BLDG. This query may be executed by first temporarily materializing the view and then joining it with the EMP table. The optimizer is then constrained to consider the join order E, V or V, E; and for computing the view, the join orders possible are D, B and B, D. Thus the total number of join order candidates is limited to 4. Also, index-based join on E, V is precluded since there is no index on V on the join column Dno. The **view-merging** operation merges the tables in the view with the tables from the outer query block and produces the following query:

```
SELECT E.Ln, B.Addr, B.Phone
FROM EMP E, DEPT D, BLDG B
WHERE D.Bldg_id = B.Bldg_id AND D.Dno = E.Dno AND E.Fn = "John";
```

With the merged query block above, three tables appear in the FROM clause, thus affording eight possible join orders and indexes on Dno in DEPT, and Bldg_id in BLDG can be used for index-based nested loop joins that were previously excluded. We leave it to the reader to develop execution plans with and without merging to see the comparison.

In general, views containing select-project-join operations are considered simple views and they can always be subjected to this type of view-merging. Typically, view merging enables additional options to be considered and results in an execution plan that is better than one without view merging. Sometimes other optimizations are enabled, such as dropping a table in the outer query if it is used within the view. View-merging may be invalid under certain conditions where the view is more complex and involves DISTINCT, OUTER JOIN, AGGREGATION, GROUP BY set operations, and so forth. We next consider a possible situation of GROUP-BY view-merging.

GROUP-BY View-Merging: When the view has additional constructs besides select-project-join as we mentioned above, merging of the view as shown above may or may not be desirable. Delaying the Group By operation after performing joins may afford the advantage of reducing the data subjected to grouping in case the joins have low join selectivity. Alternately, performing early Group By may be advantageous by reducing the amount of data subjected to subsequent joins. The optimizer would typically consider execution plans with and without merging and compare their cost to determine the viability of doing the merging. We illustrate with an example.

Consider the following relations:

```
SALES (Custid, Productid, Date, Qty_sold)
CUST (Custid, Custname, Country, Cemail)
PRODUCT (Productid, Pname, Qty_onhand)
```

The query: List customers from France who have bought more than 50 units of a product "Ring_234" may be set up as follows:

A view is created to count total quantity of any item bought for the <Custid, Productid> pairs:

```
CREATE VIEW CP_BOUGHT_VIEW AS
SELECT SUM (S.Qty_sold) as Bought, S.Custid, S.Productid
FROM SALES S
GROUP BY S.Custid, S.Productid;
```

Then the query using this view becomes:

```
QG: SELECT C.Custid, C.Custname, C.Cemail
FROM CUST C, PRODUCT P, CP_BOUGHT_VIEW V1
WHERE P.Productid = V1.Productid AND C.Custid = V1.Custid AND V1.
Bought >50
AND Pname = "Ring_234" AND C.Country = "France";
```

The view V1 may be evaluated first and its results temporarily materialized, then the query QG may be evaluated using the materialized view as one of the tables in the join. By using the merging transformation, this query becomes:

```
QT: SELECT C.Custid, C.Custname, C.Cemail
FROM CUST C, PRODUCT P, SALES S
WHERE P.Productid = S.Productid AND C.Custid = S.Custid AND
          Pname = "Ring_234" AND C.Country = "France"
GROUP BY, P.Productid, P.rowid, C.rowid, C.Custid, C.Custname, C.Cemail
HAVING SUM (S.Qty_sold) > 50;
```

After merging, the resulting query QT is much more efficient and cheaper to execute. The reasoning is as follows. Before merging, the view V1 does grouping on the entire SALES table and materializes the result, and it is expensive to do so. In the transformed query, the grouping is applied to the join of the three tables; in this operation, a single product tuple is involved from the PRODUCT table, thus filtering the data from SALES considerably. The join in QT after transformation may be slightly more expensive in that the whole SALES relation is involved rather than the aggregated view table CP_BOUGHT_VIEW in QG. Note, however, that the GROUP-BY operation in V1 produces a table whose cardinality is not considerably smaller than the cardinality of SALES, because the grouping is on <Custid, Productid>, which may not have high repetition in SALES. Also note the use of P.rowid and C.rowid, which refer to the unique row identifiers that are added to maintain equivalence with the original query. We reiterate that the decision to merge GROUP-BY views must be made by the optimizer based on estimated costs.

19.2.4 Materialized Views

We discussed the concept of views in Section 7.3 and also introduced the concept of materialization of views. A view is defined in the database as a query, and a **materialized view** stores the results of that query. Using materialized views to avoid some of the computation involved in a query is another query optimization technique. A materialized view may be stored temporarily to allow more queries to be processed against it or permanently, as is common in data warehouses (see Chapter 29). A materialized view constitutes derived data because its content can be computed as a result of processing the defining query of the materialized view. The main idea behind materialization is that it is much cheaper to read it when needed and query against it than to recompute it from scratch. The savings can be significant when the view involves costly operations like join, aggregation, and so forth.

Consider, for example, view V2 in Section 7.3, which defines the view as a relation by joining the DEPARTMENT and EMPLOYEE relations. For every department, it computes the total number of employees and the total salary paid to employees in that department. If this information is frequently required in reports or queries, this view may be permanently stored. The materialized view may contain data related only to a fragment or sub-expression of the user query. Therefore, an involved algorithm is needed to replace only the relevant fragments of the query with one or more materialized views and compute the rest of the query in a conventional way. We also mentioned in Section 7.3 three update (also known as refresh) strategies for updating the view:

- Immediate update, which updates the view as soon as any of the relations participating in the view are updated
- Lazy update, which recomputes the view only upon demand
- Periodic update (or deferred update), which updates the view later, possibly with some regular frequency

When immediate update is in force, it constitutes a large amount of overhead to keep the view updated when any of the underlying base relations have a change in the form of insert, delete, and modify. For example, deleting an employee from the database, or changing the salary of an employee, or hiring a new employee affects the tuple corresponding to that department in the view and hence would require the view V2 in Section 7.3 to be immediately updated. These updates are handled sometimes manually by programs that update all views defined on top of a base relation whenever the base relation is updated. But there is obviously no guarantee that all views may be accounted for. Triggers (see Section 7.2) that are activated upon an update to the base relation may be used to take action and make appropriate changes to the materialized views. The straightforward and naive approach is to recompute the entire view for every update to any base table and is prohibitively costly. Hence incremental view maintenance is done in most RDBMSs today. We discuss that next.

Incremental View Maintenance. The basic idea behind incremental view maintenance is that instead of creating the view from scratch, it can be updated incrementally by accounting for only the changes that occurred since the last time it was created/updated. The trick is in figuring out exactly what is the net change to the materialized view based on a set of inserted or deleted tuples in the base relation. We describe below the general approaches to incremental view maintenance for views involving join, selection, projection, and a few types of aggregation. To deal with modification, we can consider these approaches as a combination of delete of the old tuple followed by an insert of the new tuple. Assume a view *V* defined over relations *R* and *S*. The respective instances are *v*, *r*, and *s*.

Join: If a view contains inner join of relations *r* and *s*, $v_{\text{old}} = r \bowtie s$, and there is a new set of tuples inserted: r_i, in *r*, then the new value of the view contains $(r \cup r_i) \bowtie s$. The incremental change to the view can be computed as $v_{\text{new}} = r \bowtie s \cup r_i \bowtie \text{s}$. Similarly, deleting a set of tuples r_d from *r* results in the new view as $v_{\text{new}} = r \bowtie s - r_d \bowtie s$. We will have similar expressions symmetrically when s undergoes addition or deletion.

Selection: If a view is defined as $V = \sigma_{\text{C}}$ R with condition C for selection, when a set of tuples r_i are inserted into *r*, the view can be modified as $v_{\text{new}} = v_{old} \cup \sigma_{\text{C}}\, r_i$. On the other hand, upon deletion of tuples r_d from *r*, we get $v_{\text{new}} = v_{\text{old}} - \sigma_{\text{C}}\, r_d$.

Projection: Compared to the above strategy, projection requires additional work. Consider the view defined as V = $\pi_{Sex,\ Salary}$R, where R is the EMPLOYEE relation, and suppose the following <Sex, Salary> pairs exist for Salary of 50,000 in r in three distinct tuples: t_5 contains <M, 50000>, t_{17} contains <M, 50000> and t_{23} contains <F, 50000>. The view v therefore contains <M, 50000> and <F, 50000> as two tuples derived from the three tuples of *r*. If tuple t_5 were to be deleted from *r*, it would have no effect on the view. However, if t_{23} were to be deleted from *r*, the tuple <F, 50000> would have to be removed from the view. Similarly, if another new tuple t_{77} containing <M, 50000> were to be inserted in the relation r, it also would have no effect on the view. Thus, view maintenance of projection views requires a count to be maintained in addition to the actual columns in the view. In the above example, the original count values are 2 for <M, 50000> and 1 for <F, 50000>. Each time an insert to the base relation results in contributing to the view, the count is incremented; if a deleted tuple from the base relation has been represented in the view, its count is decremented. When the count of a tuple in the view reaches zero, the tuple is actually dropped from the view. When a new inserted tuple contributes to the view, its count is set to 1. Note that the above discussion assumes that SELECT DISTINCT is being used in defining the view to correspond to the project (π) operation. If the multiset version of projection is used with no DISTINCT, the counts would still be used. There is an option to display the view tuple as many times as its count in case the view must be displayed as a multiset.

Intersection: If the view is defined as V= R ∩ S, when a new tuple r_i is inserted, it is compared against the **s** relation to see if it is present there. If present, it is inserted in *v*, else not. If tuple r_d is deleted, it is matched against the view *v* and, if present there, it is removed from the view.

Aggregation (Group By): For aggregation, let us consider that GROUP BY is used on column G in relation R and the view contains (SELECT G, aggregate-function (A)). The view is a result of some aggregation function applied to attribute A, which corresponds to (see Section 8.4.2):

$$_{G}\Im_{\text{Aggregate-function}}(A)$$

We consider a few aggregate-functions below:

- **Count:** For keeping the count of tuples for each group, if a new tuple is inserted in r, and if it has a value for G = g1, and if g1 is present in the view, then its count is incremented by 1. If there is no tuple with the value g1 in the view, then a new tuple is inserted in the view: <g1, 1>. When the tuple being deleted has the value G = g1, its count is decremented by 1. If the count of g1 reaches zero after deletion in the view, that tuple is removed from the view.
- **Sum:** Suppose the view contains (G, sum(A)). There is a count maintained for each group in the view. If a tuple is inserted in the relation r and has (g1, x1) under the columns R.G and R.A, and if the view does not have an entry for g1, a new tuple <g1, x1> is inserted in the view and its count is set to 1. If there is already an entry for g1 as <g1, s1> in the old view, it is modified as <g1, s1 + x1> and its count is incremented by 1. For the deletion from base relation of a tuple with R.G, R.A being <g1, x1>, if the count of the corresponding group g1 is 1, the tuple for group g1 would be removed from the view. If it is present and has count higher than 1, the count would be decremented by 1 and the sum s1 would be decremented to s1– x1.
- **Average:** The aggregate function cannot be maintained by itself without maintaining the sum and the count functions and then computing the average as sum divided by count. So both the sum and count functions need to be maintained and incrementally updated as discussed above to compute the new average.
- **Max and Min:** We can just consider Max. Min would be symmetrically handled. Again for each group, the (g, max(a), count) combination is maintained, where max(a) represents the maximum value of R.A in the base relation. If the inserted tuple has R.A value lower than the current max(a) value, or if it has a value equal to max(a) in the view, only the count for the group is incremented. If it has a value greater than max(a), the max value in the view is set to the new value and the count is incremented. Upon deletion of a tuple, if its R.A value is less than the max(a), only the count is decremented. If the R.A value matches the max(a), the count is decremented by 1; so the tuple that represented the max value of A has been deleted. Therefore, a new max must be computed for A for the group that requires substantial amount of work. If the count becomes 0, that group is removed from the view because the deleted tuple was the last tuple in the group.

We discussed incremental materialization as an optimization technique for maintaining views. However, we can also look upon materialized views as a way to reduce the effort in certain queries. For example, if a query has a component, say, R $\bowtie$ S or π_LR that is available as a view, then the query may be modified to use the view and avoid doing unnecessary computation. Sometimes an opposite situation happens. A view V is used in the query Q, and that view has been materialized as v; let us say the view includes R $\bowtie$ S; however, no access structures like indexes are available on v. Suppose that indexes are available on certain attributes, say, A of the component relation R and that the query Q involves a selection condition on A. In such cases, the query against the view can benefit by using the index on a component relation, and the view is replaced by its defining query; the relation representing the materialized view is not used at all.

19.3 Use of Selectivities in Cost-Based Optimization

A query optimizer does not depend solely on heuristic rules or query transformations; 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**, rather than ad-hoc 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 18.1 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**.[6] 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 19.3.1, we discuss the components of query execution cost. In Section 19.3.2, we discuss the type of information needed in cost functions. This information is kept in the DBMS catalog. In Section 19.3.3, we describe histograms that are used to keep details on the value distributions of important attributes.

The decision-making process during query optimization is nontrivial and has multiple challenges. We can abstract the overall cost-based query optimization approach in the following way:

- For a given subexpression in the query, there may be multiple equivalence rules that apply. The process of applying equivalences is a cascaded one; it

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

does not have any limit and there is no definitive convergence. It is difficult to conduct this in a space-efficient manner.

- It is necessary to resort to some quantitative measure for evaluation of alternatives. By using the space and time requirements and reducing them to some common metric called cost, it is possible to devise some methodology for optimization.
- Appropriate search strategies can be designed by keeping the cheapest alternatives and pruning the costlier alternatives.
- The scope of query optimization is generally a query block. Various table and index access paths, join permutations (orders), join methods, group-by methods, and so on provide the alternatives from which the query optimizer must chose.
- In a global query optimization, the scope of optimization is multiple query blocks.[7]

19.3.1 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 23), it would also include the cost of transferring tables and results among various computers during query evaluation.

[7]We do not discuss global optimization in this sense in the present chapter. Details may be found in Ahmed et al. (2006).

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 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 23), communication cost must be minimized. 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. This 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.

19.3.2 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. These were mentioned in Section 18.3.4, and we utilized them while illustrating the various implementation algorithms for relational operations. 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.

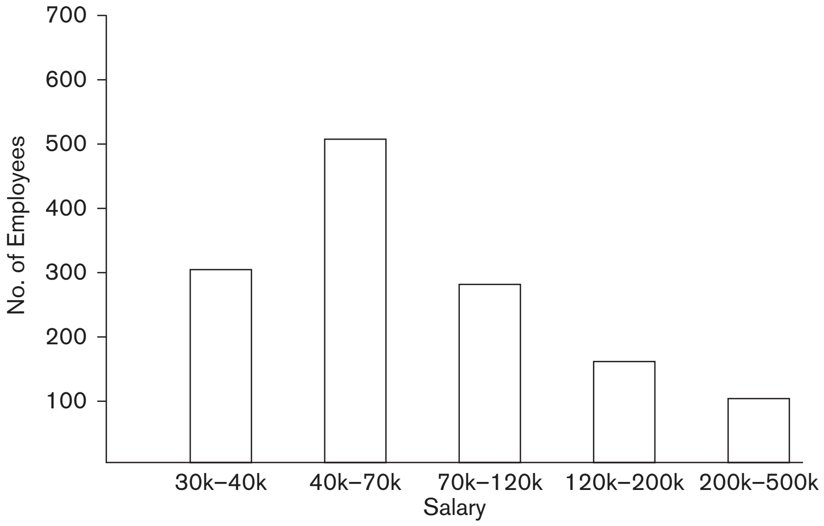
Another important parameter is the **number of distinct values NDV (A, R)** of an attribute in relation *R* and the attribute **selectivity (*sl*)**, which is the fraction of records satisfying an equality condition on the attribute. This allows estimation of the **selection cardinality** (s = ***sl*∗*r***) of an attribute, which is the *average* number of records that will satisfy an equality selection condition on that attribute.

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. To help with estimating the size of the results of queries, it is important to have as good an estimate of the distribution of values as possible. To that end, most systems store a histogram.

19.3.3 Histograms

Histograms are tables or data structures maintained by the DBMS to record information about the distribution of data. It is customary for most RDBMSs to store histograms for most of the important attributes. Without a histogram, the best assumption is that values of an attribute are uniformly distributed over its range from high to low. Histograms divide the attribute over important ranges (called buckets) and store the total number of records that belong to that bucket in that relation. Sometimes they may also store the number of distinct values in each bucket as well. An implicit assumption is made sometimes that among the distinct values within a bucket there is a uniform distribution. All these assumptions are oversimplifications that rarely hold. So keeping a histogram with a finer granularity (i.e., larger number of buckets) is always useful. A couple of variations of histograms are common: in **equi-width** histograms, the range of values is divided into equal subranges. In **equi-height** histograms, the buckets are so formed that each one contains roughly the same number of records. Equi-height histograms are considered better since they keep fewer numbers of more frequently occurring values in one bucket and more numbers of less frequently occurring ones in a different bucket. So the uniform distribution assumption within a bucket seems to hold better. We show an example of a histogram for salary information in a company in Figure 19.4. This histogram divides the salary range into five buckets that may correspond to the important sub-ranges over which the queries may be likely because they belong to certain types of employees. It is neither an equi-width nor an equi-height histogram.

Figure 19.4
Histogram of salary in the relation EMPLOYEE.



19.4 Cost Functions for SELECT Operation

We now provide cost functions for the selection algorithms S1 to S8 discussed in Section 18.3.1 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. To reiterate, the following notation is used in the formulas hereafter:

C_{Si}: Cost for method S*i* in block accesses
r_X: Number of records (tuples) in a relation X
b_X: Number of blocks occupied by relation X (also referred to as b)
bfr_X: Blocking factor (i.e., number of records per block) in relation X
sl_A: Selectivity of an attribute A for a given condition
sA: Selection cardinality of the attribute being selected ($= sl_A * r$)
xA: Number of levels of the index for attribute A
$b_{I1}A$: Number of first-level blocks of the index on attribute A
NDV (A, X): Number of distinct values of attribute A in relation X

Note: In using the above notation in formulas, we have omitted the relation name or attribute name when it is obvious.

- **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 16.8).
- **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 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, with an equality (i.e., <attribute = value>) selection condition, 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 17.5). For range queries, 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. For a range condition such as $v1 < A < v2$, the selection cardinality s must be computed from the histogram or as a default, under the uniform distribution assumption; then the cost would be computed based on whether or not A is a key or nonkey with a B$^+$-tree index on A. (We leave this as an exercise for the reader to compute under the different conditions.)
- **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.
- **S9—Selection using a bitmap index.** (See Section 17.5.2.) Depending on the nature of selection, if we can reduce the selection to a set of equality conditions, each equating the attribute with a value (e.g., $A = \{7, 13, 17, 55\}$), then a bit vector for each value is accessed which is r bits or $r/8$ bytes long. A number of bit vectors may fit in one block. Then, if s records qualify, s blocks are accessed for the data records.
- **S10—Selection using a functional index.** (See Section 17.5.3.) This works similar to S6 except that the index is based on a function of multiple attributes; if that function is appearing in the SELECT clause, the corresponding index may be utilized.

Cost-Based Optimization Approach. 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. **Dynamic programming** is an optimization technique[8] in which subproblems are solved only once. This technique is applicable when a problem may be broken down into subproblems that themselves have subproblems. We will visit the dynamic programming approach when we discuss join ordering in Section 19.5.5. We do not discuss optimization algorithms here; rather, we use a simple example to illustrate how cost estimates may be used.

19.4.1 Example of Optimization of Selection Based on Cost Formulas:

Suppose that the EMPLOYEE file in Figure 5.5 has $r_E = 10{,}000$ records stored in $b_E = 2{,}000$ 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 $sl_{\text{Salary}} = 20/10000 = 0.002$.)
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 *NDV (Dno, EMPLOYEE)* = 125 distinct values for Dno, so the selectivity of Dno is $sl_{\text{Dno}} = (1/\ NDV\ (Dno, EMPLOYEE)) = 0.008$, and the selection cardinality is $s_{\text{Dno}} = (r_E * sl_{\text{Dno}}) = (r_E/NDV\ (Dno, EMPLOYEE)) = 80$.
4. A secondary index on Sex, with $x_{\text{Sex}} = 1$. There are *NDV (Sex, EMPLOYEE)* = 2 values for the Sex attribute, so the average selection cardinality is $s_{\text{Sex}} = (r_E/NDV\ (Sex, EMPLOYEE)) = 5000$. (Note that in this case, a histogram giving the percentage of male and female employees may be useful, unless the percentages are approximately equal.)

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

OP1: $\sigma_{\text{Ssn='123456789'}}$ (EMPLOYEE)
OP2: $\sigma_{\text{Dno>5}}$(EMPLOYEE)
OP3: $\sigma_{\text{Dno=5}}$(EMPLOYEE)
OP4: $\sigma_{\text{Dno=5 AND SALARY>30000 AND Sex='F'}}$ (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 $C_{S1b} = (b_E/2) = 1{,}000$

[8]For a detailed discussion of dynamic programming as a technique of optimization, the reader may consult an algorithm textbook such as Corman et al. (2003).

(average cost for a selection on a key attribute). For OP1 we can use either method S1 or method S6*a*; the cost estimate for S6*a* is $C_{S6a} = x_{\text{Ssn}} + 1 = 4 + 1 = 5$, and it is chosen over method S1, whose average cost is $C_{S1b} = 1{,}000$. For OP2 we can use either method S1 (with estimated cost $C_{S1a} = 2{,}000$) or method S6*b* (with estimated cost $C_{S6b} = x_{\text{Dno}} + (b_{I1\text{Dno}}/2) + (r_E /2) = 2 + (4/2) + (10{,}000/2) = 5{,}004$), so we choose the linear search approach for OP2. For OP3 we can use either method S1 (with estimated cost $C_{S1a} = 2{,}000$) or method S6a (with estimated cost $C_{S6a} = x_{\text{Dno}} + s_{\text{Dno}} = 2 + 80 = 82$), so we choose method S6*a*.

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 > 30000) 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. Consider the Dno = 5 condition in OP3 above; Dno has 125 values and hence a B$^+$-tree index would be appropriate. Instead, if we had an attribute Zipcode in EMPLOYEE and if the condition were Zipcode = 30332 and we had only five zip codes, bitmap indexing could be used to know what records qualify. Assuming uniform distribution, $s_{\text{Zipcode}} = 2{,}000$. This would result in a cost of 2,000 for bitmap indexing.

19.5 Cost Functions for the JOIN Operation

To develop reasonably accurate cost functions for JOIN operations, we must 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 $js \le (1/|S|)$.

Hence a **simple formula** to use for join selectivity is:

$js = 1/\max(\text{NDV}(A, R), \text{NDV}(B,S))$

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, which we call **join cardinality** ***(jc)***.

$jc = |(R_c\, S)| = js * |R| * |S|.$

We can now give some sample *approximate* cost functions for estimating the cost of some of the join algorithms given in Section 18.4. 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 19.4. If n_B main memory buffer blocks 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—Index-based nested-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,[9] 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

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

[9]*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.

For a primary index, we get

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

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. This is an optimistic estimate, and typically *h* ranges from 1.2 to 1.5 in practical situations.

- **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 18.2 to estimate the sorting cost.
- **J4—Partition–hash join (or just 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). As we showed in Section 18.4, the cost of this join can be approximated to:

  $C_{J4} = 3 * (b_R + b_S) + ((js * |R| * |S|)/bfr_{RS})$

19.5.1 Join Selectivity and Cardinality for Semi-Join and Anti-Join

We consider these two important operations, which are used when unnesting certain queries. In Section 18.1 we showed examples of subqueries that are transformed into these operations. The goal of these operations is to avoid the unnecessary effort of doing exhaustive pairwise matching of two tables based on the join condition. Let us consider the join selectivity and cardinality of these two types of joins.

Semi-Join

```
SELECT COUNT(*)
FROM T1
WHERE T1.X IN (SELECT T2.Y
    FROM T2);
```

Unnesting of the query above leads to semi-join. (In the following query, the notation “*S*=” for semi-join is nonstandard.)

```
SELECT COUNT(*)
FROM T1, T2
WHERE T1.X S= T2.Y;
```

The join selectivity of the semi-join above is given by:

$js = \text{MIN}(1,\text{NDV}(Y, T2)/\text{NDV}(X, T1))$

The join cardinality of the semi-join is given by:

$jc = |T1| * js$

Anti-Join Consider the following query:

```
SELECT COUNT (*)
FROM T1
WHERE T1.X NOT IN (SELECT T2.Y
FROM T2);
```

Unnesting of the query above leads to anti-join.[10] (In the following query, the notation "*A*=" for anti-join is nonstandard.)

```
SELECT COUNT(*)
FROM T1, T2
WHERE T1.X A= T2.Y;
```

The join selectivity of the anti-join above is given by:

$js = 1 - \text{MIN}(1,\text{NDV}(T2.y)/\text{NDV}(T1.x))$

The join cardinality of the anti-join is given by:

$jc = |T1|*js$

19.5.2 Example of Join Optimization Based on Cost Formulas

Suppose that we have the EMPLOYEE file described in the example in the previous section, and assume that the DEPARTMENT file in Figure 5.5 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 $js_{OP6} = (1/|\text{DEPARTMENT}|) = 1/125$[11] because Dnumber is a key of DEPARTMENT. Also assume that the blocking factor for the resulting join file is $bfr_{ED} = 4$ records

[10] Note that in order for anti-join to be used in the NOT IN subquery, both the join attributes, T1.X and T2.Y, must have non-null values. For a detailed discussion, consult Bellamkonda et al. (2009).

[11] Note that this coincides with our other formula: = 1/ max (NDV (Dno, EMPLOYEEE), NDV (Dnumber, DEPARTMENT) = 1/max (125,125) = 1/125.

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:

$$C_{J1} = b_E + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED})$$
$$= 2{,}000 + (2{,}000 * 13) + (((1/125) * 10{,}000 * 125)/4) = 30{,}500$$

2. Using method J1 with DEPARTMENT as outer loop:

$$C_{J1} = b_D + (b_E * b_D) + ((js_{OP6} * r_E * r_D)/bfr_{ED})$$
$$= 13 + (13 * 2{,}000) + (((1/125) * 10{,}000 * 125/4) = 28{,}513$$

3. Using method J2 with EMPLOYEE as outer loop:

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

4. Using method J2 with DEPARTMENT as outer loop:

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

5. Using method J4 gives:

$$C_{J4} = 3* (b_D + b_E) + ((js_{OP6} * r_E * r_D)/bfr_{ED})$$
$$= 3* (13+2{,}000) + 2{,}500 = 8{,}539$$

Case 5 has the lowest cost estimate and will be chosen. Notice that in case 2 above, if 15 memory buffer blocks (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 $b_E + b_D + ((js_{OP6} * r_E * r_D)/bfr_{ED})$ or 4,513, as discussed in Section 18.4. If some other number of main memory buffers was available, say $n_B = 10$, then the cost for case 2 would be calculated as follows, which would also give better performance than case 4:

$$C_{J1} = b_D + (\lceil b_D/(n_B - 2)\rceil * b_E) + ((js * |R| * |S|)/bfr_{RS})$$
$$= 13 + (\lceil 13/8 \rceil * 2{,}000) + (((1/125) * 10{,}000 * 125/4) = 28{,}513$$
$$= 13 + (2 * 2{,}000) + 2{,}500 = 6{,}513$$

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

19.5.3 Multirelation Queries and JOIN Ordering Choices

The algebraic transformation rules in Section 19.1.2 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 block that joins n relations will often have $n - 1$ join operations, and hence can have a large number of different join orders. In general, for a query block that has n relations, there are n! join orders; Cartesian products are included in this total number. 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 join tree** is a binary tree in which the right child of each non–leaf node is always a base relation. The optimizer would choose the particular left-deep join tree with the lowest estimated cost. Two examples of left-deep trees are shown in Figure 19.5(a). (Note that the trees in Figure 19.2 are also left-deep trees.) A **right-deep join tree** is a binary tree where the left child of every leaf node is a base relation (Figure 19.5(b)).

A **bushy join tree** is a binary tree where the left or right child of an internal node may be an internal node. Figure 19.5(b) shows a right-deep join tree whereas Figure 19.5(c) shows a bushy one using four base relations. Most query optimizers consider left-deep join trees as the preferred join tree and then choose one among the n! possible join orderings, where n is the number of relations. We discuss the join ordering issue in more detail in Sections 19.5.4 and 19.5.5. The left-deep tree has exactly one shape, and the join orders for N tables in a left-deep tree are given by N!. In contrast, the shapes of a bushy tree are given by the following recurrence relation (i.e., recursive function), with $S(n)$ defined as follows: $S(1) = 1$.

$$S(n) = \sum_{i=1}^{n-1} S(i) * S(n - \mathrm{i})$$

The above recursive equation for $S(n)$ can be explained as follows. It states that, for i between 1 and $N - 1$ as the number of leaves in the left subtree, those leaves may be rearranged in $S(i)$ ways. Similarly, the remaining $N - i$ leaves in the right subtree can be rearranged in $S(N - i)$ ways. The number of permutations of the bushy trees is given by:

$$P(n) = n! * S(n) = (2n - 2)!/(n - 1)!$$

Table 19.1 shows the number of possible left-deep (or right-deep) join trees and bushy join trees for joins of up to seven relations.

It is clear from Table 19.1 that the possible space of alternatives becomes rapidly unmanageable if all possible bushy tree alternatives were to be considered. In certain

Table19.1 Number of Permutations of Left-Deep and Bushy Join Trees of n Relations

No. of Relations N	No. of Left-Deep Trees N!	No. of Bushy Shapes $S(N)$	No. of Bushy Trees $(2N-2)!/(N-1)!$
2	2	1	2
3	6	2	12
4	24	5	120
5	120	14	1,680
6	720	42	30,240
7	5,040	132	665,280

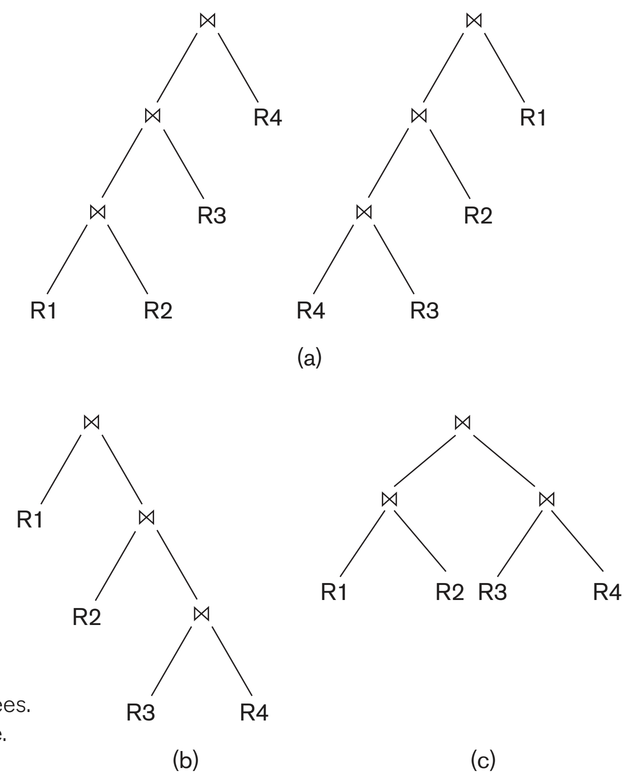


Figure 19.5
(a) Two left-deep join query trees.
(b) A right-deep join query tree.
(c) A bushy query tree.

cases like complex versions of snowflake schemas (see Section 29.3), approaches to considering bushy tree alternatives have been proposed.[12]

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 an index-based nested-loop join. One advantage of left-deep (or right-deep) trees is that they are amenable to pipelining, as discussed in Section 18.7. For instance, consider the first left-deep tree in Figure 19.5(a) and assume that the join algorithm is the index-based nested-loop method; in this case, a disk page of tuples of the outer relation is used to probe the inner relation for 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 18.7 and 19.2), the join results could be materialized and stored as temporary relations. The key idea from

[12]As a representative case for bushy trees, refer to Ahmed et al. (2014).

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.

19.5.4 Physical Optimization

For a given logical query plan based on the heuristics we have been discussing so far, each operation needs a further decision in terms of executing the operation by a specific algorithm at the physical level. This is referred to as **physical optimization**. If this optimization is based on the relative cost of each possible implementation, we call it cost-based physical optimization. The two sets of approaches to this decision making may be broadly classified as top-down and bottom-up approaches. In the **top-down** approach, we consider the options for implementing each operation working our way down the tree and choosing the best alternative at each stage. In the **bottom-up** approach, we consider the operations working up the tree, evaluating options for physical execution, and choosing the best at each stage. Theoretically, both approaches amount to evaluation of the entire space of possible implementation solutions to minimize the cost of evaluation; however, the bottom-up strategy lends itself naturally to pipelining and hence is used in commercial RDBMSs. Among the most important physical decisions is the ordering of join operations, which we will briefly discuss in Section 19.5.5. There are certain heuristics applied at the physical optimization stage that make elaborate cost computations unnecessary. These heuristics include:

- For selections, use index scans wherever possible.
- If the selection condition is conjunctive, use the selection that results in the smallest cardinality first.
- If the relations are already sorted on the attributes being matched in a join, then prefer sort-merge join to other join methods.
- For union and intersection of more than two relations, use the associative rule; consider the relations in the ascending order of their estimated cardinalities.
- If one of the arguments in a join has an index on the join attribute, use that as the inner relation.
- If the left relation is small and the right relation is large and it has index on the joining column, then try index-based nested-loop join.
- Consider only those join orders where there are no Cartesian products or where all joins appear before Cartesian products.

The following are only some of the types of physical level heuristics used by the optimizer. If the number of relations is small (typically less than 6) and, therefore, possible implementations options are limited, then most optimizers would elect to apply a cost-based optimization approach directly rather than to explore heuristics.

19.5.5 Dynamic Programming Approach to Join Ordering

We saw in Section 19.5.3 that there are many possible ways to order n relations in an n-way join. Even for $n = 5$, which is not uncommon in practical applications, the possible permutations are 120 with left-deep trees and 1,680 with bushy trees. Since bushy trees expand the solution space tremendously, left-deep trees are generally preferred (over both bushy and right-deep trees). They have multiple advantages: First, they work well with the common algorithms for join, including nested-loop, index-based nested-loop, and other one-pass algorithms. Second, they can generate **fully pipelined plans** (i.e., plans where all joins can be evaluated using pipelining). Note that inner tables must always be materialized because in the join implementation algorithms, the entire inner table is needed to perform the matching on the join attribute. This is not possible with right-deep trees.

The common approach to evaluate possible permutations of joining relations is a greedy heuristic approach called dynamic programming. **Dynamic programming** is an optimization technique[13] where subproblems are solved only once, and it is applicable when a problem may be broken down into subproblems that themselves have subproblems. A typical dynamic programming algorithm has the following characteristics[14]:

1. The structure of an optimal solution is developed.
2. The value of the optimal solution is recursively defined.
3. The optimal solution is computed and its value developed in a bottom-up fashion.

Note that the solution developed by this procedure is an optimal solution and not the absolute optimal solution. To consider how dynamic programming may be applied to the join order selection, consider the problem of ordering a 5-way join of relations $r1$, $r2$, $r3$, $r4$, $r5$. This problem has 120 (=5!) possible left-deep tree solutions. Ideally, the cost of each of them can be estimated and compared and the best one selected. Dynamic programming takes an approach that breaks down this problem to make it more manageable. We know that for three relations, there are only six possible left-deep tree solutions. Note that if all possible bushy tree join solutions were to be evaluated, there would be 12 of them. We can therefore consider the join to be broken down as:

$$r1 \bowtie r2 \bowtie r3 \bowtie r4 \bowtie r5 = (r1 \bowtie r2 \bowtie r3) \bowtie r4 \bowtie r5$$

The 6 (= 3!) possible options of $(r1 \bowtie r2 \bowtie r3)$ may then be combined with the 6 possible options of taking the result of the first join, say, temp1, and then considering the next join:

$$(\text{temp1} \bowtie r4 \bowtie r5)$$

If we were to consider the 6 options for evaluating temp1 and, for each of them, consider the 6 options of evaluating the second join $(\text{temp1} \bowtie r4 \bowtie r5)$, the possible

[13] For a detailed discussion of dynamic programming as a technique of optimization, the reader may consult an algorithm textbook such as Corman et al. (2003).

[14] Based on Chapter 16 in Corman et al. (2003).

solution space has 6 * 6 = 36 alternatives. This is where dynamic programming can be used to do a sort of greedy optimization. It takes the "optimal" plan for evaluating temp1 and does *not* revisit that plan. So the solution space now reduces to only 6 options to be considered for the second join. Thus the total number of options considered becomes 6 + 6 instead of 120 (=5!) in the nonheuristic exhaustive approach.

The order in which the result of the join is generated is also important for finding the best overall order of joins since for using sort-merge join with the next relation, it plays an important role. The ordering beneficial for the next join is considered an **interesting join order**. This approach was first proposed in System R at IBM Research.[15] Besides the join attributes of the later join, System R also included grouping attributes of a later GROUP BY or a sort order at the root of the tree among interesting sort orders. For example, in the case we discussed above, the interesting join orders for the temp1 relation will include those that match the join attribute(s) required to join with either *r*4 or with *r*5. The dynamic programming algorithm can be extended to consider best join orders for each interesting sort order. The number of subsets of n relations is 2^n (for $n = 5$ it is 32; $n = 10$ gives 1,024, which is still manageable), and the number of interesting join orders is small. The complexity of the extended dynamic programming algorithm to determine the optimal left-deep join tree permutation has been shown to be $O(3^n)$.

19.6 Example to Illustrate Cost-Based Query Optimization

We will consider query Q2 and its query tree shown in Figure 19.1(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 19.6. The LOW_VALUE and HIGH_VALUE statistics have been normalized for clarity. The tree in Figure 19.1(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

[15]See the classic reference in this area by Selinger et al. (1979).

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 19.6, 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 use of the PROJ_PLOC index—so the optimizer must compare the estimated costs of these two options. The statistical information on the PROJ_PLOC index (see Figure 19.6) shows the number of index levels $x = 2$ (root plus leaf levels). The index is nonunique

Figure 19.6
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.

(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 19.6 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 2,000/100 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 must be estimated. We can use a single-loop join on TEMP2 since in this case the index EMP_SSN (see Figure 19.6) 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.

19.7 Additional Issues Related to Query Optimization

In this section, we will discuss a few issues of interest that we have not been able to discuss earlier.

19.7.1 Displaying the System's Query Execution Plan

Most commercial RDBMSs have a provision to display the execution plan produced by the query optimizer so that DBA-level personnel can view such execution plans and try to understand the descision made by the optimizer.[16] The common syntax is some variation of EXPLAIN <query>.

- **Oracle** uses

  EXPLAIN PLAN FOR
  <SQL Query>

The query may involve INSERT, DELETE, and UPDATE statements; the output goes into a table called PLAN_TABLE. An appropriate SQL query is written to read the PLAN_TABLE. Alternately, Oracle provides two scripts UTLXPLS.SQL and UTLXPLP.SQL to display the plan table output for serial and parallel execution, respectively.

- **IBM DB2** uses

  EXPLAIN PLAN SELECTION [additional options] FOR <SQL-query>

There is no plan table. The PLAN SELECTION is a command to indicate that the explain tables should be loaded with the explanations during the plan selection phase. The same statement is also used to explain XQUERY statements.

- **SQL SERVER** uses

  SET SHOWPLAN_TEXT ON or SET SHOWPLAN_XML ON or SET SHOWPLAN_ALL ON

The above statements are used before issuing the TRANSACT-SQL, so the plan output is presented as text or XML or in a verbose form of text corresponding to the above three options.

- **PostgreSQL** uses

  EXPLAIN [set of options] <query>.where the options include ANALYZE, VERBOSE, COSTS, BUFFERS, TIMING, etc.

19.7.2 Size Estimation of Other Operations

In Sections 19.4 and 19.5, we discussed the SELECTION and JOIN operations and size estimation of the query result when the query involves those operations. Here we consider the size estimation of some other operations.

Projection: For projection of the form π_{List} (R) expressed as SELECT <attribute-list> FROM R, since SQL treats it as a multiset, the estimated number of tuples in the result is $|R|$. If the DISTINCT option is used, then size of π_A (R) is NDV (A, R).

[16]We have just illustrated this facility without describing the syntactic details of each system.

Set Operations: If the arguments for an intersection, union, or set difference are made of selections on the same relation, they can be rewritten as conjunction, disjunction, or negation, respectively. For example, $\sigma_{c1}(R) \cap \sigma_{c2}(R)$ can be rewritten as $\sigma_{c1 \text{ AND } c2}(R)$; and $\sigma_{c1}(R) \cup \sigma_{c2}(R)$ can be rewritten as $\sigma_{c1 \text{ OR } c2}(R)$. The size estimation can be made based on the selectivity of conditions c1 and c2. Otherwise, the estimated upper bound on the size of $r \cap s$ is the minimum of the sizes of r and s; the estimated upper bound on the size of $r \cup s$ is the sum of their sizes.

Aggregation: The size of $_{G}\Im_{\text{Aggregate-function}}(A)$ R is NDV (G, R) since there is one group for each unique value of G.

Outer Join : the size of R LEFT OUTER JOIN S would be $|R \bowtie S|$ plus |R anti-join S|. Similarly, the size of R FULL OUTER JOIN S would be $|r \bowtie s|$ plus |r anti-join s| plus |s anti-join r|. We discussed anti-join selectivity estimation in Section 19.5.1.

19.7.3 Plan Caching

In Chapter 2, we referred to parametric users who run the same queries or transactions repeatedly, but each time with a different set of parameters. For example, a bank teller uses an account number and some function code to check the balance in that account. To run such queries or transactions repeatedly, the query optimizer computes the best plan when the query is submitted for the first time and caches the plan for future use. This storing of the plan and reusing it is referred to as **plan caching**. When the query is resubmitted with different constants as parameters, the same plan is reused with the new parameters. It is conceivable that the plan may need to be modified under certain situations; for example, if the query involves report generation over a range of dates or range of accounts, then, depending on the amount of data involved, different strategies may apply. Under a variation called **parametric query optimization**, a query is optimized without a certain set of values for its parameters and the optimizer outputs a number of plans for different possible value sets, all of which are cached. As a query is submitted, the parameters are compared to the ones used for the various plans and the cheapest among the applicable plans is used.

19.7.4 Top-*k* Results Optimization

When the output of a query is expected to be large, sometimes the user is satisfied with only the top-*k* results based on some sort order. Some RDBMSs have a **limit** *K* **clause** to limit the result to that size. Similarly, hints may be specified to inform the optimizer to limit the generation of the result. Trying to generate the entire result and then presenting only the top-*k* results by sorting is a naive and inefficient strategy. Among the suggested strategies, one uses generation of results in a sorted order so that it can be stopped after *K* tuples. Other strategies, such as introducing additional selection conditions based on the estimated highest value, have been proposed. Details are beyond our scope here. The reader may consult the bibliographic notes for details.

19.8 An Example of Query Optimization in Data Warehouses

In this section, we introduce another example of query transformation and rewriting as a technique for query optimization. In Section 19.2, we saw examples of query transformation and rewriting. Those examples dealt with nested subqueries and used heuristics rather than cost-based optimization. The subquery (view) merging example we showed can be considered a heuristic transformation; but the group-by view merging uses cost-based optimization as well. In this section, we consider a transformation of star-schema queries in data warehouses based on cost considerations. These queries are commonly used in data warehouse applications that follow the star schema. (See Section 29.3 for a discussion of star schemas.)

We will refer to this procedure as s**tar-transformation optimization**. The star schema contains a collection of tables; it gets its name because of the schema's resemblance to a star-like shape whose center contains one or more fact tables (relations) that reference multiple dimension tables (relations). The fact table contains information about the relationships (e.g., sales) among the various dimension tables (e.g., customer, part, supplier, channel, year, etc.) and measure columns (e.g., amount_sold, etc.). Consider the representative query called QSTAR given below. Assume that D1, D2, D3 are aliases for the dimension tables DIM1, DIM2, DIM3, whose primary keys are, respectively, D1.Pk, D2.Pk, and D3.Pk**.** These dimensions have corresponding foreign key attributes in the fact table FACT with alias F—namely, F.Fk1, F.Fk2, F.Fk3—on which joins can be defined. The query creates a grouping on attributes D1.X, D2.Y and produces a sum of the so-called "measure" attribute (see Section 29.3) F.M from the fact table F. There are conditions on attributes A, B, C in DIM1, DIM2, DIM3, respectively:

```
Query QSTAR:
SELECT D1.X, D2.Y, SUM (F.M)
FROM FACT F, DIM1 D1, DIM2 D2, DIM3 D3
WHERE F.Fk1 = D1.Pk and F.Fk2 = D2.Pk and F.Fk3 = D3.Pk and
      D1.A > 5 and D2.B < 77 and D3.C = 11
GROUP BY D1.X, D2.Y
```

The fact table is generally very large in comparison with the dimension tables. QSTAR is a typical star query, and its fact table tends to be generally very large and joined with several tables of small dimension tables. The query may also contain single-table filter predicates on other columns of the dimension tables, which are generally restrictive. The combination of these filters helps to significantly reduce the data set processed from the fact table (such as D1.A > 5 in the above query). This type of query generally does grouping on columns coming from dimension tables and aggregation on measure columns coming from the fact table.

The goal of star-transformation optimization is to access only this reduced set of data from the fact table and avoid using a full table scan on it. Two types of star-transformation optimizations are possible: (A) classic star transformation, and

(B) bitmap index star transformation. Both these optimizations are performed on the basis of comparative costs of the original and the transformed queries.

A. **Classic Star Transformation**
   In this optimization, a Cartesian product of the dimension tables is performed first after applying the filters (such as D1.A > 5) to each dimension table. Note that generally there are no join predicates between dimension tables. The result of this Cartesian product is then joined with the fact table using B-tree indexes (if any) on the joining keys of the fact table.

B. **Bitmap Index Star Transformation**
   The requirement with this optimization is that there must be bitmap[17] indexes on the fact-table joining keys referenced in the query. For example, in QSTAR, there must be bitmap indexes (see Section 17.5.2) on FACT.Fk1, FACT.Fk2, and FACT.Fk3 attributes; each bit in the bitmap corresponds to a row in the fact table. The bit is set if the key value of the attribute appears in a row of the fact table. The given query QSTAR is transformed into Q2STAR as shown below.

Q2STAR:

```
SELECT D1.X, D2.Y, SUM (F.M)
FROM FACT F, DIM1 D1, DIM2 D2
WHERE F.Fk1 = D1.Pk and F.Fk2 = D2.Pk and D1.A > 5 and D2.B < 77 and
    F.Fk1 IN (SELECT D1.Pk
              FROM DIM1 D1
              WHERE D1.A > 5) AND
    F.Fk2 IN (SELECT D2.Pk
              FROM DIM2 D2
              WHERE D2.B < 77) AND
    F.Fk3 IN (SELECT D3.pk
              FROM DIM3 D3
              WHERE D3.C = 11)
GROUP BY D1.X, D2.Y;
```

The bitmap star transformation adds subquery predicates corresponding to the dimension tables. Note that the subqueries introduced in Q2STAR may be looked upon as a set membership operation; for example, F.Fk1 IN (5, 9, 12, 13, 29 …).

When driven by bitmap AND and OR operations of the key values supplied by the dimension subqueries, only the relevant rows from the fact table need to be retrieved. If the filter predicates on the dimension tables and the intersection of the fact table joining each dimension table filtered out a significant subset of the fact table rows, then this optimization would prove to be much more efficient than a brute force full-table scan of the fact table.

[17]In some cases, the B-tree index keys can be converted into bitmaps, but we will not discuss this technique here.

The following operations are performed in Q2STAR in order to access and join the FACT table.

1. By iterating over the key values coming from a dimension subquery, the bitmaps are retrieved for a given key value from a bitmap index on the FACT table.
2. For a subquery, the bitmaps retrieved for various key values are merged (OR-ed).
3. The merged bitmaps for each dimension subqueries are AND-ed; that is, a conjunction of the joins is performed.
4. From the final bitmap, the corresponding tuple-ids for the FACT table are generated.
5. The FACT table rows are directly retrieved using the tuple-ids.

Joining Back: The subquery bitmap trees filter the fact table based on the filter predicates on the dimension tables; therefore, it may still be necessary to join the dimension tables back to the relevant rows in the fact table using the original join predicates. The join back of a dimension table can be avoided if the column(s) selected from the subquery are unique and the columns of the dimension table are not referenced in the SELECT and GROUP-BY clauses. Note that in Q2STAR, the table DIM3 is not joined back to the FACT table, since it is not referenced in the SELECT and GROUP-BY clauses, and DIM3.Pk is unique.

19.9 Overview of Query Optimization in Oracle[18]

This section provides a broad overview of various features in Oracle query processing, including query optimization, execution, and analytics.[19]

19.9.1 Physical Optimizer

The Oracle physical optimizer is cost based and was introduced in Oracle 7.1. The scope of the physical optimizer is a single query block. The physical optimizer examines alternative table and index access paths, operator algorithms, join orderings, join methods, parallel execution distribution methods, and so on. It chooses the execution plan with the lowest estimated cost. The estimated query cost is a relative number proportional to the expected elapsed time needed to execute the query with the given execution plan.

The physical optimizer calculates this cost based on object statistics (such as table cardinalities, number of distinct values in a column, column high and low values, data distribution of column values), the estimated usage of resources (such as I/O and CPU time), and memory needed. Its estimated cost is an internal metric that

[18]This section is contributed by Rafi Ahmed of Oracle Corporation.

[19]Support for analytics was introduced in Oracle 10.2.

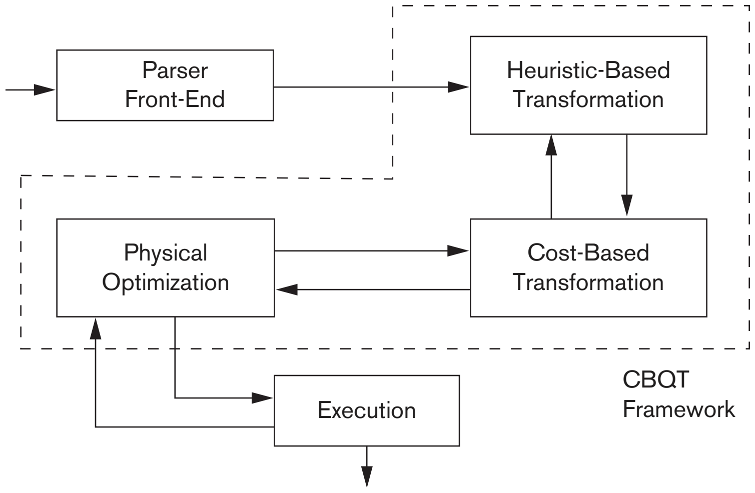


Figure 19.7
Cost-based query transformation framework (based on Ahmed et al., 2006).

roughly corresponds to the run time and the required resources. The goal of cost-based optimization in Oracle is to find the best trade-off between the lowest run time and the least resource utilization.

19.9.2 Global Query Optimizer

In traditional RDBMSs, query optimization consists of two distinct logical and physical optimization phases. In contrast, Oracle has a global query optimizer, where logical transformation and physical optimization phases have been integrated to generate an optimal execution plan for the entire query tree. The architecture of the Oracle query processing is illustrated in Figure 19.7.

Oracle performs a multitude of query transformations, which change and transform the user queries into equivalent but potentially more optimal forms. Transformations can be either heuristic-based or cost-based. The cost-based query transformation (CBQT) framework[20] introduced in Oracle 10g provides efficient mechanisms for exploring the state space generated by applying one or more transformations. During cost-based transformation, an SQL statement, which may comprise multiple query blocks, is copied and transformed and its cost is computed using the physical optimizer. This process is repeated multiple times, each time applying a new set of possibly interdependent transformations; and, at the end, one or more transformations are selected and applied to the original SQL statement, if those transformations result in an optimal execution plan. To deal with the combinatorial explosion, the CBQT framework provides efficient strategies for searching the state space of various transformations.

The availability of the general framework for cost-based transformation has made it possible for other innovative transformations to be added to the vast repertoire of

[20]As presented in Ahmed et al. (2006).

Oracle's query transformation techniques. Major among these transformations are group-by and distinct subquery merging (in the FROM clause of the query), subquery unnesting, predicate move-around, common subexpression elimination, join predicate push down, OR expansion, subquery coalescing, join factorization, subquery removal through window function, star transformation, group-by placement, and bushy join trees.[21]

The cost-based transformation framework of Oracle 10g is a good example of the sophisticated approach taken to optimize SQL queries.

19.9.3 Adaptive Optimization

Oracle's physical optimizer is adaptive and uses a feedback loop from the execution level to improve on its previous decisions. The optimizer selects the most optimal execution plan for a given SQL statement using the cost model, which relies on object statistics (e.g., number of rows, distribution of column values, etc.) and system statistics (e.g., I/O bandwidth of the storage subsystem). The optimality of the final execution plan depends primarily on the accuracy of the statistics fed into the cost model as well as on the sophistication of the cost model itself. In Oracle, the feedback loop shown in Figure 19.7 establishes a bridge between the execution engine and the physical optimizer. The bridge brings valuable statistical information to enable the physical optimizer to assess the impact of its decisions and make better decisions for the current and future executions. For example, based on the estimated value of table cardinality, the optimizer may choose the index-based nested-loop join method. However, during the execution phase, the actual table cardinality may be detected to diverge significantly from the estimated value. This information may trigger the physical optimizer to revise its decision and dynamically change the index access join method to the hash join method.

19.9.4 Array Processing

One of the critical deficiencies of SQL implementations is its lack of support for *N*-dimensional array-based computation. Oracle has made extensions for analytics and OLAP features; these extensions have been integrated into the Oracle RDBMS engine.[22] We will illustrate the need for OLAP queries when we discuss data warehousing in Chapter 29. These SQL extensions involving array-based computations for complex modeling and optimizations include access structures and execution strategies for processing these computations efficiently. The computation clause (details are beyond our scope here) allows the Oracle RDBMS to treat a table as a multidimensional array and specify a set of formulas over it. The formulas replace multiple joins and UNION operations that must be performed for equivalent computation with current ANSI SQL (where ANSI stands for

[21]More details can be found in Ahmed et al. (2006, 2014).

[22]See Witkowski et al. (2003) for more details.

American National Standards Institute). The computation clause not only allows for ease of application development but also offers the Oracle RDBMS an opportunity to perform better optimization.

19.9.5 Hints

An interesting addition to the Oracle query optimizer is the capability for an application developer to specify hints (also called query *annotations* or *directives* in other systems) to the optimizer. Hints are embedded in the text of an SQL statement. Hints are commonly used to address the infrequent cases where the optimizer chooses a suboptimal plan. The idea is that an application developer occasionally might need to override the optimizer decisions based on cost or cardinality mis-estimations. For example, consider the EMPLOYEE table shown in Figure 5.6. The Sex column of that table has only two distinct values. If there are 10,000 employees, then the optimizer, in the absence of a histogram on the Sex column, would estimate that half are male and half are female, assuming a uniform data distribution. If a secondary index exists, it would more than likely not be used. However, if the application developer knows that there are only 100 male employees, a hint could be specified in an SQL query whose WHERE-clause condition is Sex = 'M' so that the associated index would be used in processing the query. Various types of hints can be specified for different operations; these hints include but are not limited to the following:

- The access path for a given table
- The join order for a query block
- A particular join method for a join between tables
- The enabling or disabling of a transformation

19.9.6 Outlines

In Oracle RDBMSs, outlines are used to preserve execution plans of SQL statements or queries. Outlines are implemented and expressed as a collection of hints, because hints are easily portable and comprehensible. Oracle provides an extensive set of hints that are powerful enough to specify any execution plan, no matter how complex. When an outline is used during the optimization of an SQL statement, these hints are applied at appropriate stages by the optimizer (and other components). Every SQL statement processed by the Oracle optimizer automatically generates an outline that can be displayed with the execution plan. Outlines are used for purposes such as plan stability, what-if analysis, and performance experiments.

19.9.7 SQL Plan Management

Execution plans for SQL statements have a significant impact on the overall performance of a database system. New optimizer statistics, configuration parameter changes, software updates, introduction of new query optimization and processing techniques, and hardware resource utilizations are among a multitude of factors that may cause the Oracle query optimizer to generate a new execution plan for the same SQL queries or statements. Although most of the changes in the execution plans are beneficial or benign, a few execution plans may turn out to be suboptimal, which can have a negative impact on system performance.

In Oracle 11g, a novel feature called SQL plan management (SPM) was introduced[23] for managing execution plans for a set of queries or workloads. SPM provides stable and optimal performance for a set of SQL statements by preventing new suboptimal plans from being executed while allowing other new plans to be executed if they are verifiably better than the previous plans. SPM encapsulates an elaborate mechanism for managing the execution plans of a set of SQL statements, for which the user has enabled SPM. SPM maintains the previous execution plans in the form of stored outlines associated with texts of SQL statements and compares the performances of the old and new execution plans for a given SQL statement before permitting them to be used by the user. SPM can be configured to work automatically, or it can be manually controlled for one or more SQL statements.

19.10 Semantic Query Optimization

A different approach to query optimization, called **semantic query optimization**, has been suggested. This technique, which may be used in combination with the techniques discussed previously, uses constraints specified on the database schema—such as unique attributes and other more complex constraints—to modify one query into another query that is more efficient to execute. We will not discuss this approach in detail but we will illustrate it with a simple example. Consider the SQL query:

```
SELECT   E.Lname, M.Lname
FROM     EMPLOYEE AS E, EMPLOYEE AS M
WHERE    E.Super_ssn=M.Ssn AND E.Salary > M.Salary
```

This query retrieves the names of employees who earn more than their supervisors. Suppose that we had a constraint on the database schema that stated that no employee can earn more than his or her direct supervisor. If the semantic query optimizer checks for the existence of this constraint, it does not need to execute the query because it knows that the result of the query will be empty. This may save considerable time if the constraint checking can be done efficiently. However, searching through many constraints to find those that are applicable to a given query and that may semantically optimize it can also be time-consuming.

Consider another example:

```
SELECT Lname, Salary
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.Dno = DEPARTMENT.Dnumber and
EMPLOYEE.Salary>100000
```

[23]See Ziauddin et al. (2008).

In this example, the attributes retrieved are only from one relation: EMPLOYEE; the selection condition is also on that one relation. However, there is a referential integrity constraint that Employee.Dno is a foreign key that refers to the primary key Department.Dnumber. Therefore, this query can be transformed by removing the DEPARTMENT relation from the query and thus avoiding the inner join as follows:

```
SELECT Lname, Salary
FROM EMPLOYEE
WHERE EMPLOYEE.Dno IS NOT NULL and EMPLOYEE.Salary>100000
```

This type of transformation is based on the primary-key/foreign-key relationship semantics, which are a constraint between the two relations.

With the inclusion of active rules and additional metadata in database systems (see Chapter 26), semantic query optimization techniques are being gradually incorporated into DBMSs.

19.11 Summary

In the previous chapter, we presented the strategies for query processing used by relational DBMSs. We considered algorithms for various standard relational operators, including selection, projection, and join. We also discussed other types of joins, including outer join, semi-join, and anti-join, and we discussed aggregation as well as external sorting. In this chapter, our goal was to focus on query optimization techniques used by relational DBMSs. In Section 19.1 we introduced the notation for query trees and graphs and described heuristic approaches to query optimization; these approaches use heuristic rules and algebraic techniques to improve the efficiency of query execution. We showed how a query tree that represents a relational algebra expression can be heuristically optimized by reorganizing the tree nodes and transforming the tree into another equivalent query tree that is more efficient to execute. We also gave equivalence-preserving transformation rules and a systematic procedure for applying them to a query tree. In Section 19.2 we described alternative query evaluation plans, including pipelining and materialized evaluation. Then we introduced the notion of query transformation of SQL queries; this transformation optimizes nested subqueries. We also illustrated with examples of merging subqueries occurring in the FROM clause, which act as derived relations or views. We also discussed the technique of materializing views.

We discussed in some detail the cost-based approach to query optimization in Section 19.3. We discussed information maintained in catalogs that the query optimizer consults. We also discussed histograms to maintain distribution of important attributes. We showed how cost functions are developed for some database access algorithms for selection and join in Sections 19.4 and 19.5, respectively. We illustrated with an example in Section 19.6 how these cost functions are used to estimate the costs of different execution strategies. A number of additional issues such as display of query plans, size estimation of results, plan caching and top-k results optimization were discussed in Section 19.7. Section 19.8

was devoted to a discussion of how typical queries in data warehouses are optimized. We gave an example of cost-based query transformation in data warehouse queries on the so-called star schema. In Section 19.9 we presented a detailed overview of the Oracle query optimizer, which uses a number of additional techniques, details of which were beyond our scope. Finally, in Section 19.10 we mentioned the technique of semantic query optimization, which uses the semantics or integrity constraints to simplify the query or completely avoid accessing the data or the actual execution of the query.

Review Questions

19.1. What is a query execution plan?

19.2. What is meant by the term *heuristic optimization*? Discuss the main heuristics that are applied during query optimization.

19.3. How does a query tree represent a relational algebra expression? What is meant by an execution of a query tree? Discuss the rules for transformation of query trees, and identify when each rule should be applied during optimization.

19.4. How many different join orders are there for a query that joins 10 relations? How many left-deep trees are possible?

19.5. What is meant by *cost-based query optimization*?

19.6. What is the optimization approach based on dynamic programming? How is it used during query optimization?

19.7. What are the problems associated with keeping views materialized?

19.8. What is the difference between *pipelining* and *materialization*?

19.9. Discuss the cost components for a cost function that is used to estimate query execution cost. Which cost components are used most often as the basis for cost functions?

19.10. Discuss the different types of parameters that are used in cost functions. Where is this information kept?

19.11. What are semi-join and anti-join? What are the join selectivity and join cardinality parameters associated with them? Provide appropriate formulas.

19.12. List the cost functions for the SELECT and JOIN methods discussed in Sections19.4 and 19.5.

19.13. What are the special features of query optimization in Oracle that we did not discuss in the chapter?

19.14. What is meant by *semantic query optimization*? How does it differ from other query optimization techniques?

Exercises

19.15. Develop cost functions for the PROJECT, UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT algorithms discussed in Section 19.4.

19.16. Develop cost functions for an algorithm that consists of two SELECTs, a JOIN, and a final PROJECT, in terms of the cost functions for the individual operations.

19.17. Develop a pseudo-language-style algorithm for describing the dynamic programming procedure for join-order selection.

19.18. Calculate the cost functions for different options of executing the JOIN operation OP7 discussed in Section 19.4.

19.19. Develop formulas for the hybrid hash-join algorithm for calculating the size of the buffer for the first bucket. Develop more accurate cost estimation formulas for the algorithm.

19.20. Estimate the cost of operations OP6 and OP7 using the formulas developed in Exercise 19.19.

19.21. Compare the cost of two different query plans for the following query:

$$\sigma_{\text{Salary} < 40000}(\text{EMPLOYEE} \bowtie_{\text{Dno=Dnumber}} \text{DEPARTMENT})$$

Use the database statistics shown in Figure 19.6.

Selected Bibliography

This bibliography provides literature references for the topics of query processing and optimization. We discussed query processing algorithms and strategies in the previous chapter, but it is difficult to separate the literature that addresses optimization from the literature that addresses query processing strategies and algorithms. Hence, the bibliography is consolidated.

A detailed algorithm for relational algebra optimization is given by Smith and Chang (1975). The Ph.D. thesis of Kooi (1980) provides a foundation for query processing techniques. A survey paper by Jarke and Koch (1984) gives a taxonomy of query optimization and includes a bibliography of work in this area. A survey by Graefe (1993) discusses query execution in database systems and includes an extensive bibliography.

Whang (1985) discusses query optimization in OBE (Office-By-Example), which is a system based on the language QBE. Cost-based optimization was introduced in the SYSTEM R experimental DBMS and is discussed in Astrahan et al. (1976). Selinger et al. (1979) is a classic paper that discussed cost-based optimization of multiway joins in SYSTEM R. Join algorithms are discussed in Gotlieb (1975), Blasgen and Eswaran (1976), and Whang et al. (1982). Hashing algorithms for implementing joins are described and analyzed in DeWitt et al. (1984), Bratbergsengen