part 8

Query Processing and Optimization

This page intentionally left blank

chapter 18

Strategies for Query Processing[1]

In this chapter, we discuss the techniques used internally by a DBMS to process high-level queries. A query expressed in a high-level query language such as SQL must first be scanned, parsed, and validated.[2] The **scanner** identifies the query tokens—such as SQL keywords, attribute names, and relation names—that appear in the text of the query, whereas the **parser** checks the query syntax to determine whether it is formulated according to the syntax rules (rules of grammar) of the query language. The query must also be **validated** by checking that all attribute and relation names are valid and semantically meaningful names in the schema of the particular database being queried. An internal representation of the query is then created, usually as a tree data structure called a **query tree**. It is also possible to represent the query using a graph data structure called a **query graph**, which is generally a **directed acyclic graph (DAG)**. The DBMS must then devise an **execution strategy** or **query plan** for retrieving the results of the query from the database files. A query has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as **query optimization**.

We defer a detailed discussion of query optimization to the next chapter. In this chapter, we will primarily focus on how queries are processed and what algorithms are used to perform individual operations within the query. Figure 18.1 shows the different steps of processing a high-level query. The **query optimizer** module has the task of producing a good execution plan, and the **code generator** generates the code to execute that plan. The **runtime database processor** has the task of running (executing) the query code, whether in compiled or interpreted mode, to produce the query result. If a runtime error results, an error message is generated by the runtime database processor.

---

[1]We appreciate Rafi Ahmed's contributions in updating this chapter.

[2]We will not discuss the parsing and syntax-checking phase of query processing here; this material is discussed in compiler texts.

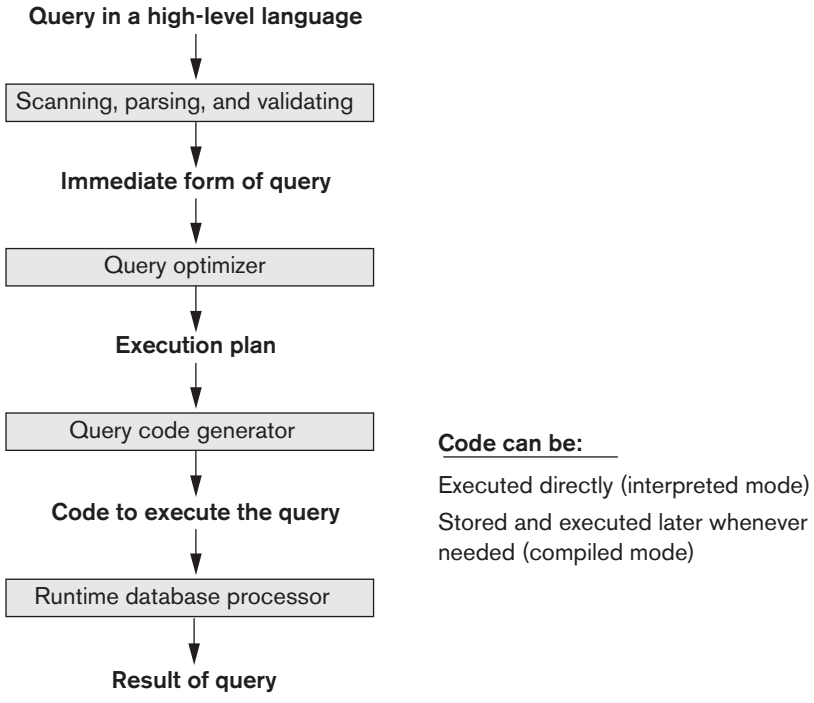


Figure 18.1
Typical steps when processing a high-level query.

The term *optimization* is actually a misnomer because in some cases the chosen execution plan is not the optimal (or absolute best) strategy—it is just a *reasonably efficient or the best available strategy* for executing the query. Finding the optimal strategy is usually too time-consuming—except for the simplest of queries. In addition, trying to find the optimal query execution strategy requires accurate and detailed information about the size of the tables and distributions of things such as column values, which may not be always available in the DBMS catalog. Furthermore, additional information such as the size of the expected result must be derived based on the predicates in the query. Hence, *planning of a good execution strategy* may be a more accurate description than *query optimization.*

For lower-level navigational database languages in legacy systems—such as the network DML or the hierarchical DL/1 the programmer must choose the query execution strategy while writing a database program. If a DBMS provides only a navigational language, there is a *limited opportunity* for extensive query optimization by the DBMS; instead, the programmer is given the capability to choose the query execution strategy. On the other hand, a high-level query language—such as SQL for relational DBMSs (RDBMSs) or OQL (see Chapter 12) for object DBMSs (ODBMSs)—is more declarative in nature because it specifies what the intended results of the query are rather than identifying the details of *how* the result should be obtained. Query optimization is thus necessary for queries that are specified in a high-level query language.

We will concentrate on describing query processing and optimization in the *context of an RDBMS* because many of the techniques we describe have also been adapted for other types of database management systems, such as ODBMSs.[3] A relational DBMS must systematically evaluate alternative query execution strategies and choose a reasonably efficient or near-optimal strategy. Most DBMSs have a number of general database access algorithms that implement relational algebra operations such as SELECT or JOIN (see Chapter 8) or combinations of these operations. Only execution strategies that can be implemented by the DBMS access algorithms and that apply to the particular query, as well as to the *particular physical database design*, can be considered by the query optimization module.

This chapter is organized as follows. Section 18.1 starts with a general discussion of how SQL queries are typically translated into relational algebra queries and additional operations and then optimized. Then we discuss algorithms for implementing relational algebra operations in Sections 18.2 through 18.6. In Section 18.7, we discuss the strategy for execution called pipelining. Section 18.8 briefly reviews the strategy for parallel execution of the operators. Section 18.9 summarizes the chapter.

In the next chapter, we will give an overview of query optimization strategies. There are two main techniques of query optimization that we will be discussing. The first technique is based on **heuristic rules** for ordering the operations in a query execution strategy that works well in most cases but is not guaranteed to work well in every case. The rules typically reorder the operations in a query tree. The second technique involves **cost estimation** of different execution strategies and choosing the execution plan that minimizes estimated cost. The topics covered in this chapter require that the reader be familiar with the material presented in several earlier chapters. In particular, the chapters on SQL (Chapters 6 and 7), relational algebra (Chapter 8), and file structures and indexing (Chapters 16 and 17) are a prerequisite to this chapter. Also, it is important to note that the topic of query processing and optimization is vast, and we can only give an introduction to the basic principles and techniques in this and the next chapter. Several important works are mentioned in the Bibliography of this and the next chapter.

18.1 Translating SQL Queries into Relational Algebra and Other Operators

In practice, SQL is the query language that is used in most commercial RDBMSs. An SQL query is first translated into an equivalent extended relational algebra expression—represented as a query tree data structure—that is then optimized. Typically, SQL queries are decomposed into *query blocks*, which form the basic units that can be translated into the algebraic operators and optimized. A **query block** contains a single SELECT-FROM-WHERE expression, as well as GROUP BY

[3]There are some query processing and optimization issues and techniques that are pertinent only to ODBMSs. However, we do not discuss them here because we give only an introduction to query processing in this chapter and we do not discuss query optimization until Chapter 19.

and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as separate query blocks. Because SQL includes aggregate operators—such as MAX, MIN, SUM, and COUNT—these operators must also be included in the extended algebra, as we discussed in Section 8.4.

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

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ( SELECT MAX (Salary)
                 FROM   EMPLOYEE
                 WHERE  Dno=5 );
```

This query retrieves the names of employees (from any department in the company) who earn a salary that is greater than the *highest salary in department 5*. The query includes a nested subquery and hence would be decomposed into two blocks. The inner block is:

```
( SELECT MAX (Salary)
  FROM   EMPLOYEE
  WHERE  Dno=5 )
```

This retrieves the highest salary in department 5. The outer query block is:

```
SELECT  Lname, Fname
FROM    EMPLOYEE
WHERE Salary > c
```

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

$$\Im_{\text{MAX Salary}}(\sigma_{\text{Dno=5}}(\text{EMPLOYEE}))$$

and the outer block into the expression:

$$\pi_{\text{Lname,Fname}}(\sigma_{\text{Salary>c}}(\text{EMPLOYEE}))$$

The *query optimizer* would then choose an execution plan for each query block. Notice that in the above example, the inner block needs to be evaluated only once to produce the maximum salary of employees in department 5, which is then used—as the constant c—by the outer block. We called this a *nested subquery block* (*which is uncorrelated to the outer query block*) in Section 7.1.2. It is more involved to optimize the more complex *correlated nested subqueries* (see Section 7.1.3), where a tuple variable from the outer query block appears in the WHERE-clause of the inner query block. Many techniques are used in advanced DBMSs to unnest and optimize correlated nested subqueries.

18.1.1 Additional Operators Semi-Join and Anti-Join

Most RDBMSs currently process SQL queries arising from various types of enterprise applications that include ad hoc queries, standard canned queries with parameters,

and queries for report generation. Additionally, SQL queries originate from OLAP (online analytical processing) applications on data warehouses (we discuss data warehousing in detail in Chapter 29). Some of these queries are transformed into operations that are not part of the standard relational algebra we discussed in Chapter 8. Two commonly used operations are **semi-join** and **anti-join**. Note that both these operations are a type of join. Semi-join is generally used for unnesting EXISTS, IN, and ANY subqueries.[4] Here we represent semi-join by the following non-standard syntax: T1.X *S* = T2.Y, where T1 is the left table and T2 is the right table of the semi-join. The semantics of semi-join are as follows: A row of T1 is returned as soon as T1.X finds a match with any value of T2.Y without searching for further matches. This is in contrast to finding all possible matches in inner join.

Consider a slightly modified version of the schema in Figure 5.5 as follows:

EMPLOYEE (Ssn, Bdate, Address, Sex, Salary, Dno)
DEPARTMENT (Dnumber, Dname, Dmgrssn, Zipcode)

where a department is located in a specific zip code.

Let us consider the following query:

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

Here we have a nested query which is joined by the connector **IN**.

To remove the nested query:

```
( SELECT  E.Dno
  FROM    EMPLOYEE E    WHERE E.Salary > 200000)
```

is called as **unnesting**. It leads to the following query with an operation called **semi-join,**[5] which we show with a non-standard notation "*S*=" below:

```
SELECT COUNT(*)
FROM     EMPLOYEE E, DEPARTMENT D
WHERE  D.Dnumber S= E.Dno and E.Salary > 200000;
```

The above query is counting the number of departments that have employees who make more than $200,000 annually. Here, the operation is to find the department whose Dnumber attribute matches the value(s) for the Dno attribute of Employee with that high salary.

[4]In some cases where duplicate rows are not relevant, inner join can also be used to unnest EXISTS and ANY subqueries.

[5]Note that this semi-join operator is not the same as that used in distributed query processing.

In algebra, alternate notations exist. One common notation is shown in the following figure.

Semi-join

Now consider another query:

```
Q (AJ) :  SELECT COUNT(*)
FROM      EMPLOYEE
WHERE     EMPLOYEE.Dno NOT IN (SELECT   DEPARTMENT.Dnumber
                               FROM     DEPARTMENT
                               WHERE    Zipcode =30332)
```

The above query counts the number of employees who *do not* work in departments located in zip code 30332. Here, the operation is to find the employee tuples whose Dno attribute does *not* match the value(s) for the Dnumber attribute in DEPARTMENT for the given zip code. We are only interested in producing a count of such employees, and performing an inner join of the two tables would, of course, produce wrong results. In this case, therefore, the **anti-join** operator is used while unnesting this query.

Anti-join is used for unnesting NOT EXISTS, NOT IN, and ALL subqueries. We represent anti-join by the following nonstandard syntax: T1.x *A* = T2.y, where T1 is the left table and T2 is the right table of the anti-join. The semantics of anti-join are as follows: A row of T1 is rejected as soon as T1.x finds a match with any value of T2.y. A row of T1 is returned, only if T1.x does not match with any value of T2.y.

In the following result of unnesting, we show the aforementioned anti-join with the nonstandard symbol "*A*=" in the following:

```
SELECT COUNT(*)
FROM      EMPLOYEE, DEPARTMENT
WHERE     EMPLOYEE.Dno A= DEPARTMENT AND Zipcode =30332
```

In algebra, alternate notations exist. One common notation is shown in the following figure.

Anti-join

18.2 Algorithms for External Sorting

Sorting is one of the primary algorithms used in query processing. For example, whenever an SQL query specifies an ORDER BY-clause, the query result must be sorted. Sorting is also a key component in sort-merge algorithms used for JOIN and

other operations (such as UNION and INTERSECTION), and in duplicate elimination algorithms for the PROJECT operation (when an SQL query specifies the DISTINCT option in the SELECT clause). We will discuss one of these algorithms in this section. Note that sorting of a particular file may be avoided if an appropriate index—such as a primary or clustering index (see Chapter 17)—exists on the desired file attribute to allow ordered access to the records of the file.

External sorting refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files.[6] The typical external sorting algorithm uses a **sort-merge strategy**, which starts by sorting small subfiles—called **runs**—of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn. The sort-merge algorithm, like other database algorithms, requires *buffer space* in main memory, where the actual sorting and merging of the runs is performed. The basic algorithm, outlined in Figure 18.2, consists of two phases: the sorting phase and the merging phase. The buffer space in main memory is part of the **DBMS cache**—an area in the computer's main memory that is controlled by the DBMS. The buffer space is divided into individual buffers, where each **buffer** is the same size in bytes as the size of one disk block. Thus, one buffer can hold the contents of exactly *one disk block*.

In the **sorting phase**, runs (portions or pieces) of the file that can fit in the available buffer space are read into main memory, sorted using an *internal* sorting algorithm, and written back to disk as temporary sorted subfiles (or runs). The size of each run and the **number of initial runs (n_R)** are dictated by the **number of file blocks (b)** and the **available buffer space (n_B)**. For example, if the number of available main memory buffers $n_B = 5$ disk blocks and the size of the file $b = 1{,}024$ disk blocks, then $n_R = \lceil (b/n_B) \rceil$ or 205 initial runs each of size 5 blocks (except the last run, which will have only 4 blocks). Hence, after the sorting phase, 205 sorted runs (or 205 sorted subfiles of the original file) are stored as temporary subfiles on disk.

In the **merging phase**, the sorted runs are merged during one or more **merge passes**. Each merge pass can have one or more merge steps. The **degree of merging (d_M)** is the number of sorted subfiles that can be merged in each merge step. During each merge step, one buffer block is needed to hold one disk block from each of the sorted subfiles being merged, and one additional buffer is needed for containing one disk block of the merge result, which will produce a larger sorted file that is the result of merging several smaller sorted subfiles. Hence, d_M is the smaller of $(n_B - 1)$ and n_R, and the number of merge passes is $\lceil (\log_{dM}(n_R)) \rceil$. In our example, where $n_B = 5$, $d_M = 4$ (four-way merging), so the 205 initial sorted runs would be merged 4 at a time in each step into 52 larger sorted subfiles at the end of the first merge pass. These 52 sorted files are then merged 4 at a time into 13 sorted files, which are then merged into 4 sorted files, and then finally into 1 fully sorted file, which means that *four passes* are needed.

[6] *Internal sorting algorithms* are suitable for sorting data structures, such as tables and lists, that can fit entirely in main memory. These algorithms are described in detail in data structures and algorithms texts, and include techniques such as quick sort, heap sort, bubble sort, and many others. We do not discuss these here. Also, main-memory DBMSs such as HANA employ their own techniques for sorting.

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

Figure 18.2
Outline of the sort-merge algorithm for external sorting.

The performance of the sort-merge algorithm can be measured in terms of the number of disk block reads and writes (between the disk and main memory) before the sorting of the whole file is completed. The following formula approximates this cost:

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

The first term $(2 * b)$ represents the number of block accesses for the sorting phase, since each file block is accessed twice: once for reading into a main memory buffer and once for writing the sorted records back to disk into one of the sorted subfiles. The second term represents the number of block accesses for the merging phase. During each merge pass, a number of disk blocks approximately equal to the original file blocks b is read and written. Since the number of merge passes is $(\log_{dM} n_R)$, we get the total merge cost of $(2 * b * (\log_{dM} n_R))$.

The minimum number of main memory buffers needed is $n_B = 3$, which gives a d_M of 2 and an n_R of $\lceil (b/3) \rceil$. The minimum d_M of 2 gives the worst-case performance of the algorithm, which is:

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

The following sections discuss the various algorithms for the operations of the relational algebra (see Chapter 8).

18.3 Algorithms for SELECT Operation

18.3.1 Implemention Options for the SELECT Operation

There are many algorithms for executing a SELECT operation, which is basically a search operation to locate the records in a disk file that satisfy a certain condition. Some of the search algorithms depend on the file having specific access paths, and they may apply only to certain types of selection conditions. We discuss some of the algorithms for implementing SELECT in this section. We will use the following operations, specified on the relational database in Figure 5.5, to illustrate our discussion:

OP1: $\sigma_{\text{Ssn = '123456789'}}$ (EMPLOYEE)
OP2: $\sigma_{\text{Dnumber > 5}}$ (DEPARTMENT)
OP3: $\sigma_{\text{Dno= 5}}$ (EMPLOYEE)
OP4: $\sigma_{\text{Dno= 5 AND Salary > 30000 AND Sex = 'F'}}$ (EMPLOYEE)
OP5: $\sigma_{\text{Essn = '123456789' AND Pno =10}}$(WORKS_ON)

OP6: An SQL Query:

```
SELECT *
FROM   EMPLOYEE
WHERE  Dno IN (3,27, 49)
```

OP7: An SQL Query (from Section 17.5.3)

```
SELECT First_name, Lname
FROM Employee
WHERE ((Salary*Commission_pct) + Salary ) > 15000;
```

Search Methods for Simple Selection. A number of search algorithms are possible for selecting records from a file. These are also known as **file scans**, because they scan the records of a file to search for and retrieve records that satisfy a selection condition.[7] If the search algorithm involves the use of an index, the index search is called an **index scan**. The following search methods (S1 through S6) are examples of some of the search algorithms that can be used to implement a select operation:

- **S1—Linear search (brute force algorithm).** Retrieve *every record* in the file, and test whether its attribute values satisfy the selection condition. Since the

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

records are grouped into disk blocks, each disk block is read into a main memory buffer, and then a search through the records within the disk block is conducted in main memory.

- **S2—Binary search.** If the selection condition involves an equality comparison on a key attribute on which the file is **ordered**, binary search—which is more efficient than linear search—can be used. An example is OP1 if Ssn is the ordering attribute for the EMPLOYEE file.[8]
- **S3a—Using a primary index.** If the selection condition involves an equality comparison on a **key attribute** with a primary index—for example, Ssn = '123456789' in OP1—use the primary index to retrieve the record. Note that this condition retrieves a single record (at most).
- **S3b—Using a hash key.** If the selection condition involves an equality comparison on a **key attribute** with a hash key—for example, Ssn = '123456789' in OP1—use the hash key to retrieve the record. Note that this condition retrieves a single record (at most).
- **S4—Using a primary index to retrieve multiple records.** If the comparison condition is >, >=, <, or <= on a key field with a primary index—for example, Dnumber > 5 in OP2—use the index to find the record satisfying the corresponding equality condition (Dnumber = 5); then retrieve all subsequent records in the (ordered) file. For the condition Dnumber < 5, retrieve all the preceding records.
- **S5—Using a clustering index to retrieve multiple records.** If the selection condition involves an equality comparison on a **nonkey attribute** with a clustering index—for example, Dno = 5 in OP3—use the index to retrieve all the records satisfying the condition.
- **S6—Using a secondary (B^+-tree) index on an equality comparison.** This search method can be used to retrieve a single record if the indexing field is a **key** (has unique values) or to retrieve multiple records if the indexing field is **not a key**. This can also be used for comparisons involving >, >=, <, or <=. Queries involving a range of values (e.g., 3,000 <= Salary <= 4,000) in their selection are called **range queries**. In case of range queries, the B^+-tree index leaf nodes contain the indexing field value in order—so a sequence of them is used corresponding to the requested range of that field and provide record pointers to the qualifying records.
- **S7a—Using a bitmap index.** (See Section 17.5.2.) If the selection condition involves a set of values for an attribute (e.g., Dnumber in (3,27,49) in OP6), the corresponding bitmaps for each value can be OR-ed to give the set of record ids that qualify. In this example, that amounts to OR-ing three bitmap vectors whose length is the same as the number of employees.

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

- **S7b—Using a functional index.** (See Section 17.5.3.) In OP7, the selection condition involves the expression `((Salary*Commission_pct) + Salary )`. If there is a functional index defined as (as shown in Section 17.5.3):

```
CREATE INDEX income_ix
ON EMPLOYEE (Salary + (Salary*Commission_pct));
```

  then this index can be used to retrieve employee records that qualify. Note that the exact way in which the function is written while creating the index is immaterial.

In the next chapter, we discuss how to develop formulas that estimate the access cost of these search methods in terms of the number of block accesses and access time. Method S1 (**linear search**) applies to any file, but all the other methods depend on having the appropriate access path on the attribute used in the selection condition. Method S2 (**binary search**) requires the file to be sorted on the search attribute. The methods that use an index (S3a, S4, S5, and S6) are generally referred to as **index searches**, and they require the appropriate index to exist on the search attribute. Methods S4 and S6 can be used to retrieve records in a certain *range* in **range queries**. Method S7a (**bitmap index search**) is suitable for retrievals where an attribute must match an enumerated set of values. Method S7b (**functional index search**) is suitable when the match is based on a function of one or more attributes on which a functional index exists.

18.3.2 Search Methods for Conjunctive Selection

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

- **S8—Conjunctive selection using an individual index.** If an attribute involved in any **single simple condition** in the conjunctive select condition has an access path that permits the use of one of the methods S2 to S6, use that condition to retrieve the records and then check whether each retrieved record *satisfies the remaining simple conditions* in the conjunctive select condition.
- **S9—Conjunctive selection using a composite index.** If two or more attributes are involved in equality conditions in the conjunctive select condition and a composite index (or hash structure) exists on the combined fields—for example, if an index has been created on the composite key (Essn, Pno) of the WORKS_ON file for OP5—we can use the index directly.
- **S10—Conjunctive selection by intersection of record pointers.**[9] If secondary indexes (or other access paths) are available on more than one of the fields involved in simple conditions in the conjunctive select condition, and if

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

the indexes include record pointers (rather than block pointers), then each index can be used to retrieve the **set of record pointers** that satisfy the individual condition. The **intersection** of these sets of record pointers gives the record pointers that satisfy the conjunctive select condition, which are then used to retrieve those records directly. If only some of the conditions have secondary indexes, each retrieved record is further tested to determine whether it satisfies the remaining conditions.[10] In general, method S10 assumes that each of the indexes is on a *nonkey field* of the file, because if one of the conditions is an equality condition on a key field, only one record will satisfy the whole condition. The bitmap and functional indexes discussed above in S7 are applicable for conjunctive selection on multiple attributes as well. For conjunctive selection on multiple attributes, the resulting bitmaps are AND-ed to produce the list of record ids; the same can be done when one or more set of record ids comes from a functional index.

Whenever a single condition specifies the selection—such as OP1, OP2, or OP3—the DBMS can only check whether or not an access path exists on the attribute involved in that condition. If an access path (such as index or hash key or bitmap index or sorted file) exists, the method corresponding to that access path is used; otherwise, the brute force, linear search approach of method S1 can be used. Query optimization for a SELECT operation is needed mostly for conjunctive select conditions whenever *more than one* of the attributes involved in the conditions have an access path. The optimizer should choose the access path that *retrieves the fewest records* in the most efficient way by estimating the different costs (see Section 19.3) and choosing the method with the least estimated cost.

18.3.3 Search Methods for Disjunctive Selection

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

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

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

All the methods discussed in S1 through S7 are applicable for each simple condition yielding a possible set of record ids. The query optimizer must choose the appropriate one for executing each SELECT operation in a query. This optimization uses

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

formulas that estimate the costs for each available access method, as we will discuss in Sections 19.4 and 19.5. The optimizer chooses the access method with the lowest estimated cost.

18.3.4 Estimating the Selectivity of a Condition

To minimize the overall cost of query execution in terms of resources used and response time, the query optimizer receives valuable input from the system catalog, which contains crucial statistical information about the database.

Information in the Database Catalog. A typical RDBMS catalog contains the following types of information:

For each relation (table) r with schema R containing r_R tuples:

- The number of rows/records or its cardinality: $|r(R)|$. We will refer to the number of rows simply as r_R.
- The "width" of the relation (i.e., the length of each tuple in the relation) this length of tuple is referred to as R.
- The number of blocks that relation occupies in storage: referred to as b_R.
- The blocking factor bfr, which is the number of tuples per block.

For each attribute A in relation R:

- The number of distinct values of A in R: NDV (A, R).
- The max and min values of attribute A in R: max (A, R) and min (A, R).

Note that many other forms of the statistics are possible and may be kept as needed. If there is a composite index on attributes <A, B>, then the NDV (R, <A, B>) is of significance. An effort is made to keep these statistics as accurate as possible; however, keeping them accurate up-to-the-minute is considered unnecessary since the overhead of doing so in fairly active databases is too high. We will be revisiting many of the above parameters again in Section 19.3.2.

When the optimizer is choosing between multiple simple conditions in a conjunctive select condition, it typically considers the *selectivity* of each condition. The **selectivity (*sl*)** is defined as the ratio of the number of records (tuples) that satisfy the condition to the total number of records (tuples) in the file (relation), and thus it is a number between zero and one. *Zero selectivity* means none of the records in the file satisfies the selection condition, and a selectivity of one means that all the records in the file satisfy the condition. In general, the selectivity will not be either of these two extremes, but will be a fraction that estimates the percentage of file records that will be retrieved.

Although exact selectivities of all conditions may not be available, **estimates of selectivities** are possible from the information kept in the DBMS catalog and are used by the optimizer. For example, for an equality condition on a key attribute of relation $r(R)$, $s = 1/|r(R)|$, where $|r(R)|$ is the number of tuples in relation $r(R)$. For an equality condition on a nonkey attribute with *i distinct values, s* can be estimated by

$(|r(R)|/i)/|r(R)|$ or $1/i$, assuming that the records are evenly or **uniformly distributed** among the distinct values. Under this assumption, $|r(R)|/i$ records will satisfy an equality condition on this attribute. For a range query with the selection condition,

A ≥ v, assuming uniform distribution,
sl = 0 if v > max (A, R)
sl = max (A, R) – v / max (A, R) – min (A, R)

In general, the number of records satisfying a selection condition with selectivity *sl* is estimated to be $|r(R)| * sl$. The smaller this estimate is, the higher the desirability of using that condition first to retrieve records. For a nonkey attribute with NDV (A, R) distinct values, it is often the case that those values are not uniformly distributed.

If the actual distribution of records among the various distinct values of the attribute is kept by the DBMS in the form of a **histogram**, it is possible to get more accurate estimates of the number of records that satisfy a particular condition. We will discuss the catalog information and histograms in more detail in Section 19.3.3.

18.4 Implementing the JOIN Operation

The JOIN operation is one of the most time-consuming operations in query processing. Many of the join operations encountered in queries are of the EQUIJOIN and NATURAL JOIN varieties, so we consider just these two here since we are only giving an overview of query processing and optimization. For the remainder of this chapter, the term **join** refers to an EQUIJOIN (or NATURAL JOIN).

There are many possible ways to implement a **two-way join**, which is a join on two files. Joins involving more than two files are called **multiway joins**. The number of possible ways to execute multiway joins grows rapidly because of the combinatorial explosion of possible join orderings. In this section, we discuss techniques for implementing *only two-way joins*. To illustrate our discussion, we refer to the relational schema shown in Figure 5.5 once more—specifically, to the EMPLOYEE, DEPARTMENT, and PROJECT relations. The algorithms we discuss next are for a join operation of the form:

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

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

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

18.4.1 Methods for Implementing Joins

- **J1—Nested-loop join (or nested-block join).** This is the default (brute force) algorithm because it does not require any special access paths on either file in the

join. For each record t in R (outer loop), retrieve every record s from S (inner loop) and test whether the two records satisfy the join condition $t[A] = s[B]$.[11]

- **J2—Index-based nested-loop join (using an access structure to retrieve the matching records).** If an index (or hash key) exists for one of the two join attributes—say, attribute B of file S—retrieve each record t in R (loop over file R), and then use the access structure (such as an index or a hash key) to retrieve directly all matching records s from S that satisfy $s[B] = t[A]$.
- **J3—Sort-merge join.** If the records of R and S are *physically sorted* (ordered) by value of the join attributes A and B, respectively, we can implement the join in the most efficient way possible. Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for A and B. If the files are not sorted, they may be sorted first by using external sorting (see Section 18.2). In this method, pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file—unless both A and B are nonkey attributes, in which case the method needs to be modified slightly. A sketch of the sort-merge join algorithm is given in Figure 18.3(a). We use $R(i)$ to refer to the ith record in file R. A variation of the sort-merge join can be used when secondary indexes exist on both join attributes. The indexes provide the ability to access (scan) the records in order of the join attributes, but the records themselves are physically scattered all over the file blocks, so this method may be inefficient because every record access may involve accessing a different disk block.
- **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). First, a single pass through the file with fewer records (say, R) hashes its records to the various partitions of R; this is called the **partitioning phase**, since the records of R are partitioned into the hash buckets. In the simplest case, we assume that the smaller file can fit entirely in main memory after it is partitioned, so that the partitioned subfiles of R are all kept in main memory. The collection of records with the same value of $h(A)$ are placed in the same partition, which is a **hash bucket** in a hash table in main memory. In the second phase, called the **probing phase**, a single pass through the other file (S) then hashes each of its records using the same hash function $h(B)$ to *probe* the appropriate bucket, and that record is combined with all matching records from R in that bucket. This simplified description of partition-hash join assumes that the smaller of the two files *fits entirely into memory buckets* after the first phase. We will discuss the general case of partition-hash join below that does not require this assumption. In practice, techniques J1 to J4 are implemented by accessing *whole disk blocks* of a file, rather than individual records. Depending on the available number of buffers in memory, the number of blocks read in from the file can be adjusted.

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

Figure 18.3
Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where R has n tuples and S has m tuples. (a) Implementing the operation $T \leftarrow R \bowtie_{A=B} S$. (b) Implementing the operation $T \leftarrow \pi_{<\text{attribute list}>}(R)$.

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

                    (* output other tuples that match R(i), if any *)
                    set l ← j + 1;
                    while (l ≤ m) and (R(i)[A] = S(l)[B])
                    do {   output the combined tuple <R(i), S(l)> to T;
                           set l ← l + 1
                    }

             (* output other tuples that match S(j), if any *)
             set k ← i + 1;
             while (k ≤ n) and (R(k)[A] = S(j)[B])
             do {   output the combined tuple <R(k), S(j)> to T;
                    set k ← k + 1
             }
             set i ←k, j ← l
           }
    }

(b) create a tuple t[<attribute list>] in T′ for each tuple t in R;
        (* T′ contains the projection results before duplicate elimination *)
    if <attribute list> includes a key of R
        then T ← T′
    else {   sort the tuples in T′;
             set i ← 1, j ← 2;
             while i ≤ n
             do {   output the tuple T′[i] to T;
                    while T′[i] = T′[j] and j ≤ n do j ← j + 1;          (* eliminate duplicates *)
                    i ← j; j ← i + 1
             }
    }
    (*T contains the projection result after duplicate elimination*)
```

Figure 18.3 (continued)
Implementing JOIN, PROJECT, UNION, INTERSECTION, and SET DIFFERENCE by using sort-merge, where R has n tuples and S has m tuples. (c) Implementing the operation $T \leftarrow R \cup S$. (d) Implementing the operation $T \leftarrow R \cap S$. (e) Implementing the operation $T \leftarrow R - S$.

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

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

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

18.4.2 How Buffer Space and Choice of Outer-Loop File Affect Performance of Nested-Loop Join

The buffer space available has an important effect on some of the join algorithms. First, let us consider the nested-loop approach (J1). Looking again at the operation OP6 above, assume that the number of buffers available in main memory for implementing the join is $n_B = 7$ blocks (buffers). Recall that we assume that each memory buffer is the same size as one disk block. For illustration, assume that the DEPARTMENT file consists of $r_D = 50$ records stored in $b_D = 10$ disk blocks and that the EMPLOYEE file consists of $r_E = 6{,}000$ records stored in $b_E = 2{,}000$ disk blocks. It is advantageous to read as many blocks as possible at a time into memory from the file whose records are used for the outer loop. Note that keeping one block for reading from the inner file and one block for writing to the output file, $n_B - 2$ blocks are available to read from the outer relation, The algorithm can then read one block at a time for the inner-loop file and use its records to **probe** (that is, search) the outer-loop blocks that are currently in main memory for matching records. This reduces the total number of block accesses. An extra buffer in main memory is needed to contain the resulting records after they are joined, and the contents of this result buffer can be appended to the **result file**—the disk file that will contain the join result—whenever it is filled. This result buffer block then is reused to hold additional join result records.

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

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

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

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

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

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

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

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

The join algorithm uses a buffer to hold the joined records of the result file. Once the buffer is filled, it is written to disk and its contents are appended to the result file, and then refilled with join result records.[12]

[12]If we reserve two buffers for the result file, double buffering can be used to speed the algorithm (see Section 16.3).

If the result file of the join operation has b_{RES} disk blocks, each block is written once to disk, so an additional b_{RES} block accesses (writes) should be added to the preceding formulas in order to estimate the total cost of the join operation. The same holds for the formulas developed later for other join algorithms. As this example shows, it is advantageous to use the file *with fewer blocks* as the outer-loop file in the nested-loop join.

18.4.3 How the Join Selection Factor Affects Join Performance

Another factor that affects the performance of a join, particularly the single-loop method J2, is the fraction of records in one file that will be joined with records in the other file. We call this the **join selection factor**[13] of a file with respect to an equijoin condition with another file. This factor depends on the particular equijoin condition between the two files. To illustrate this, consider the operation OP7, which joins each DEPARTMENT record with the EMPLOYEE record for the manager of that department. Here, each DEPARTMENT record (there are 50 such records in our example) will be joined with a *single* EMPLOYEE record, but many EMPLOYEE records (the 5,950 of them that do not manage a department) will not be joined with any record from DEPARTMENT.

Suppose that secondary indexes exist on both the attributes Ssn of EMPLOYEE and Mgr_ssn of DEPARTMENT, with the number of index levels $x_{\text{Ssn}} = 4$ and $x_{\text{Mgr\_ssn}} = 2$, respectively. We have two options for implementing method J2. The first retrieves each EMPLOYEE record and then uses the index on Mgr_ssn of DEPARTMENT to find a matching DEPARTMENT record. In this case, no matching record will be found for employees who do not manage a department. The number of block accesses for this case is approximately:

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

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

$$b_D + (r_{\text{D}} * (x_{\text{Ssn}} + 1)) = 10 + (50 * 5) = 260 \text{ block accesses}$$

The second option is more efficient because the join selection factor of DEPARTMENT *with respect to the join condition* Ssn = Mgr_ssn is 1 (every record in DEPARTMENT will be joined), whereas the join selection factor of EMPLOYEE with respect to the same join condition is (50/6,000), or 0.008 (only 0.8% of the records in EMPLOYEE will be joined). For method J2, either the smaller file or the file that has a match for every record (that is, the file with the high join selection factor) should be used in the (single) join loop. It is also possible to create an index specifically for performing the join operation if one does not already exist.

[13]This is different from the *join selectivity*, which we will discuss in Chapter 19.

The sort-merge join J3 is quite efficient if both files are already sorted by their join attribute. Only a single pass is made through each file. Hence, the number of blocks accessed is equal to the sum of the numbers of blocks in both files. For this method, both OP6 and OP7 would need $b_E + b_D = 2{,}000 + 10 = 2{,}010$ block accesses. However, both files are required to be ordered by the join attributes; if one or both are not, a sorted copy of each file must be created specifically for performing the join operation. If we roughly estimate the cost of sorting an external file by $(b \log_2 b)$ block accesses, and if both files need to be sorted, the total cost of a sort-merge join can be estimated by $(b_E + b_D + b_E \log_2 b_E + b_D \log_2 b_D)$.[14]

18.4.4 General Case for Partition-Hash Join

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

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

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

In the **partitioning phase**, R is partitioned into the M partitions $R_1, R_2, \ldots, R_M$, and S into the M partitions $S_1, S_2, \ldots, S_M$. The property of each pair of corresponding partitions R_i, S_i with respect to the join operation is that records in R_i *only need to be joined* with records in S_i, and vice versa. This property is ensured by using the *same hash function* to partition both files on their join attributes—attribute A for R and attribute B for S. The minimum number of in-memory buffers needed for the **partitioning phase** is $M + 1$. Each of the files R and S is partitioned separately. During partitioning of a file, M in-memory buffers are allocated to store the records that hash to each partition, and one additional buffer is needed to hold one block at a time of the input file being partitioned. Whenever the in-memory buffer for a partition gets filled, its contents are appended to a **disk subfile** that stores the partition. The partitioning phase has *two iterations.* After the first iteration, the first file R is partitioned into the subfiles $R_1, R_2, \ldots, R_M$, where all the records that hashed to the same buffer are in the same partition. After the second iteration, the second file S is similarly partitioned.

In the second phase, called the **joining** or **probing phase**, M *iterations* are needed. During iteration i, two corresponding partitions R_i and S_i are joined. The minimum

[14]We can use the more accurate formulas from Section 19.5 if we know the number of available buffers for sorting.

number of buffers needed for iteration i is the number of blocks in the smaller of the two partitions, say R_i, plus two additional buffers. If we use a nested-loop join during iteration i, the records from the smaller of the two partitions R_i are copied into memory buffers; then all blocks from the other partition S_i are read—one at a time—and each record is used to **probe** (that is, search) partition R_i for matching record(s). Any matching records are joined and written into the result file. To improve the efficiency of in-memory probing, it is common to use an *in-memory hash table* for storing the records in partition R_i by using a *different* hash function from the partitioning hash function.[15]

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

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

OP6: EMPLOYEE $\bowtie_{\text{Dno=Dnumber}}$ DEPARTMENT

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

18.4.5 Hybrid Hash-Join

The **hybrid hash-join algorithm** is a variation of partition hash-join, where the *joining* phase for *one of the partitions* is included in the *partitioning* phase. To illustrate this, let us assume that the size of a memory buffer is one disk block; that n_B such buffers are *available*; and that the partitioning hash function used is $h(K) = K \bmod M$, so that M partitions are being created, where $M < n_B$. For illustration, assume we are performing the join operation OP6. In the *first pass* of the partitioning phase, when the hybrid hash-join algorithm is partitioning the smaller of the two files

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

(DEPARTMENT in OP6), the algorithm divides the buffer space among the M partitions such that all the blocks of the *first partition* of DEPARTMENT completely reside in main memory. For each of the other partitions, only a single in-memory buffer—whose size is one disk block—is allocated; the remainder of the partition is written to disk as in the regular partition-hash join. Hence, at the end of the *first pass of the partitioning phase,* the first partition of DEPARTMENT resides wholly in main memory, whereas each of the other partitions of DEPARTMENT resides in a disk subfile.

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

18.5 Algorithms for PROJECT and Set Operations

A PROJECT operation $\pi_{<\text{attribute list}>}(R)$ from relational algebra implies that after projecting R on only the columns in the list of attributes, any duplicates are removed by treating the result strictly as a set of tuples. However, the SQL query:

SELECT Salary
FROM EMPLOYEE

produces a list of salaries of all employees. If there are 10,000 employees and only 80 distinct values for salary, it produces a one column result with 10,000 tuples. This operation is done by simple linear search by making a complete pass through the table.

Getting the true effect of the relational algebra $\pi_{<\text{attribute list}>}(R)$ operator is straightforward to implement if <attribute list> includes a key of relation R, because in this case the result of the operation will have the same number of tuples as R, but with only the values for the attributes in <attribute list> in each tuple. If <attribute list> does not include a key of R, *duplicate tuples must be eliminated.* This can be done by sorting the result of the operation and then eliminating duplicate tuples, which appear consecutively after sorting. A sketch of the algorithm is given in Figure 18.3(b). Hashing can also be used to eliminate duplicates: as each record is hashed and inserted into a bucket of the hash file in memory, it is checked against those records already in the bucket; if it is a duplicate, it is not inserted in the bucket. It is useful to recall here that in SQL queries, the default is not to eliminate duplicates from the query result; duplicates are eliminated from the query result only if the keyword DISTINCT is included.

Set operations—UNION, INTERSECTION, SET DIFFERENCE, and CARTESIAN PRODUCT—are sometimes expensive to implement, since UNION, INTERSECTION, MINUS or SET DIFFERENCE are set operators and must always return distinct results.

In particular, the CARTESIAN PRODUCT operation $R \times S$ is expensive because its result includes a record for each combination of records from R and S. Also, each record in the result includes all attributes of R and S. If R has n records and j attributes, and S has m records and k attributes, the result relation for $R \times S$ will have $n * m$ records and each record will have $j + k$ attributes. Hence, it is important to avoid the CARTESIAN PRODUCT operation and to substitute other operations such as join during query optimization. The other three set operations—UNION, INTERSECTION, and SET DIFFERENCE[16]—apply only to **type-compatible** (or union-compatible) relations, which have the same number of attributes and the same attribute domains. The customary way to implement these operations is to use variations of the **sort-merge technique:** the two relations are sorted on the same attributes, and, after sorting, a single scan through each relation is sufficient to produce the result. For example, we can implement the UNION operation, $R \cup S$, by scanning and merging both sorted files concurrently, and whenever the same tuple exists in both relations, only one is kept in the merged result. For the INTERSECTION operation, $R \cap S$, we keep in the merged result only those tuples that appear in *both sorted relations.* Figure 18.3(c) to (e) sketches the implementation of these operations by sorting and merging. Some of the details are not included in these algorithms.

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

18.5.1 Use of Anti-Join for SET DIFFERENCE (or EXCEPT or MINUS in SQL)

The MINUS operator in SQL is transformed into an anti-join (which we introduced in Section 18.1) as follows. Suppose we want to find out which departments have no employees in the schema of Figure 5.5:

Select Dnumber from DEPARTMENT MINUS Select Dno from EMPLOYEE;

[16]SET DIFFERENCE is called MINUS or EXCEPT in SQL.

can be converted into the following:

SELECT DISTINCT DEPARTMENT.Dnumber
FROM DEPARTMENT, EMPLOYEE
WHERE DEPARTMENT.Dnumber *A* = EMPLOYEE.Dno

We used the nonstandard notation for anti-join, "*A*=", where DEPARTMENT is on the left of anti-join and EMPLOYEE is on the right.

In SQL, there are two variations of these set operations. The operations UNION, INTERSECTION, and EXCEPT or MINUS (the SQL keywords for the SET DIFFERENCE operation) apply to traditional sets, where no duplicate records exist in the result. The operations UNION ALL, INTERSECTION ALL, and EXCEPT ALL apply to multisets (or bags). Thus, going back to the database of Figure 5.5, consider a query that finds all departments that employees are working on where at least one project exists controlled by that department, and this result is written as:

SELECT Dno from EMPLOYEE
INTERSECT ALL
SELECT Dum from PROJECT

This would not eliminate any duplicates of Dno from EMPLOYEE while performing the INTERSECTION. If all 10,000 employees are assigned to departments where some project is present in the PROJECT relation, the result would be the list of all the 10,000 department numbers including duplicates.. This can be accomplished by the semi-join operation we introduced in Section 18.1 as follows:

SELECT DISTINCT EMPLOYEE.Dno
FROM DEPARTMENT, EMPLOYEE
WHERE EMPLOYEE.Dno *S* = DEPARTMENT.Dnumber

If INTERSECTION is used without the ALL, then an additional step of duplicate elimination will be required for the selected department numbers.

18.6 Implementing Aggregate Operations and Different Types of JOINs

18.6.1 Implementing Aggregate Operations

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

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

If an (ascending) B^+-tree index on Salary exists for the EMPLOYEE relation, then the optimizer can decide on using the Salary index to search for the largest Salary value in the index by following the *rightmost* pointer in each index node from the

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

The index could also be used for the AVERAGE and SUM aggregate functions, but only if it is a **dense index**—that is, if there is an index entry for every record in the main file. In this case, the associated computation would be applied to the values in the index. For a **nondense index**, the actual number of records associated with each index value must be used for a correct computation. This can be done if the *number of records associated with each value* in the index is stored in each index entry. For the COUNT aggregate function, the number of values can be also computed from the index in a similar manner. If a COUNT(*) function is applied to a whole relation, the number of records currently in each relation are typically stored in the catalog, and so the result can be retrieved directly from the catalog.

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

```
SELECT    Dno, AVG(Salary)
FROM      EMPLOYEE
GROUP BY  Dno;
```

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

Notice that if a **clustering index** (see Chapter 17) exists on the grouping attribute(s), then the records are *already partitioned* (grouped) into the appropriate subsets. In this case, it is only necessary to apply the computation to each group.

18.6.2 Implementing Different Types of JOINs

In addition to the standard JOIN (also called INNER JOIN in SQL), there are variations of JOIN that are frequently used. Let us briefly consider three of them below: outer joins, semi-joins, and anti-joins.

Outer Joins. In Section 6.4, we discussed the *outer join operation*, with its three variations: left outer join, right outer join, and full outer join. In Chapter 5, we

discussed how these operations can be specified in SQL. The following is an example of a left outer join operation in SQL:

```
SELECT  E.Lname, E.Fname, D.Dname
FROM    (EMPLOYEE E LEFT OUTER JOIN DEPARTMENT D ON E.Dno = D.Dnumber);
```

The result of this query is a table of employee names and their associated departments. The table contains the same results as a regular (inner) join, with the exception that if an EMPLOYEE tuple (a tuple in the *left* relation) *does not have an associated department,* the employee's name will still appear in the resulting table, but the department name would be NULL for such tuples in the query result. Outer join can be looked upon as a combination of inner join and anti-join.

Outer join can be computed by modifying one of the join algorithms, such as nested-loop join or single-loop join. For example, to compute a *left* outer join, we use the left relation as the outer loop or index-based nested loop because every tuple in the left relation must appear in the result. If there are matching tuples in the other relation, the joined tuples are produced and saved in the result. However, if no matching tuple is found, the tuple is still included in the result but is padded with NULL value(s). The sort-merge and hash-join algorithms can also be extended to compute outer joins.

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

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

   $\text{TEMP1} \leftarrow \pi_{\text{Lname, Fname, Dname}} (\text{EMPLOYEE} \bowtie_{\text{Dno=Dnumber}} \text{DEPARTMENT})$

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

   $\text{TEMP2} \leftarrow \pi_{\text{Lname, Fname}} (\text{EMPLOYEE}) - \pi_{\text{Lname, Fname}} (\text{TEMP1})$

   This minus operation can be achieved by performing an anti-join on Lname, Fname between EMPLOYEE and TEMP1, as we discussed above in Section 18.5.2.

3. Pad each tuple in TEMP2 with a NULL Dname field.

   $\text{TEMP2} \leftarrow \text{TEMP2} \times \textbf{NULL}$

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

   $\text{RESULT} \leftarrow \text{TEMP1} \cup \text{TEMP2}$

The cost of the outer join as computed above would be the sum of the costs of the associated steps (inner join, projections, set difference, and union). However, note that step 3 can be done as the temporary relation is being constructed in step 2; that is, we can simply pad each resulting tuple with a NULL. In addition, in step 4, we know that the two operands of the union are disjoint (no common tuples), so there is no need for duplicate elimination. So the preferred method is to use a combination of inner join and anti-join rather than the above steps since the algebraic

approach of projection followed by set difference causes temporary tables to be stored and processed multiple times.

The right outer join can be converted to a left outer join by switching the operands and hence needs no separate discussion. **Full outer join** requires computing the result of inner join and then padding to the result extra tuples arising from unmatched tuples from both the left and right operand relations. Typically, full outer join would be computed by extending sort-merge or hashed join algorithms to account for the unmatched tuples.

Implementing Semi-Join and Anti-Join. In Section 18.1, we introduced these types of joins as possible operations to which some queries with nested subqueries get mapped. The purpose is to be able to perform some variant of join instead of evaluating the subquery multiple times. Use of inner join would be invalid in these cases, since for every tuple of the outer relation, the inner join looks for all possible matches on the inner relation. In semi-join, the search stops as soon as the first match is found and the tuple from outer relation is selected; in anti-join, search stops as soon as the first match is found and the tuple from outer relation is rejected. Both these types of joins can be implemented as an extension of the join algorithms we discussed in Section 18.4.

Implementing Non-Equi-Join Join operation may also be performed when the join condition is one of inequality. In Chapter 6, we referred to this operation as theta-join.This functionality is based on a condition involving any operators, such as $<$, $>$, $\geq$, $\leq$, $\neq$, and so on. All of the join methods discussed are again applicable here with the exception that hash-based algorithms cannot be used.

18.7 Combining Operations Using Pipelining

A query specified in SQL will typically be translated into a relational algebra expression that is *a sequence of relational operations.* If we execute a single operation at a time, we must generate temporary files on disk to hold the results of these temporary operations, creating excessive overhead. Evaluating a query by creating and storing each temporary result and then passing it as an argument for the next operator is called **materialized evaluation**. Each temporary materialized result is then written to disk and adds to the overall cost of query processing.

Generating and storing large temporary files on disk is time-consuming and can be unnecessary in many cases, since these files will immediately be used as input to the next operation. To reduce the number of temporary files, it is common to generate query execution code that corresponds to algorithms for combinations of operations in a query.

For example, rather than being implemented separately, a JOIN can be combined with two SELECT operations on the input files and a final PROJECT operation on the resulting file; all this is implemented by one algorithm with two input files and a single output file. Rather than creating four temporary files, we apply the algorithm directly and get just one result file.

In Section 19.1, we discuss how heuristic relational algebra optimization can group operations together for execution. Combining several operations into one and avoiding the writing of temporary results to disk is called **pipelining** or **stream-based processing**.

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

- Avoiding the additional cost and time delay incurred for writing the intermediate results to disk.
- Being able to start generating results as quickly as possible when the root operator is combined with some of the operators discussed in the following section means that the pipelined evaluation can start generating tuples of the result while rest of the pipelined intermediate tables are undergoing processing.

18.7.1 Iterators for implementing Physical Operations

Various algorithms for algebraic operations involve reading some input in the form of one or more files, processing it, and generating an output file as a relation. If the operation is implemented in such a way that it outputs one tuple at a time, then it can be regarded as an **iterator**. For example, we can devise a tuple-based implementation of the nested-loop join that will generate a tuple at a time as output. Iterators work in contrast with the materialization approach wherein entire relations are produced as temporary results and stored on disk or main memory and are read back again by the next algorithm. The query plan that contains the query tree may be executed by invoking the iterators in a certain order. Many iterators may be active at one time, thereby passing results up the execution tree and avoiding the need for additional storage of temporary results. The iterator interface typically consists of the following methods:

1. Open (): This method initializes the operator by allocating buffers for its input and output and initializing any data structures needed for the operator. It is also used to pass arguments such as selection conditions needed to perform the operation. It in turn calls Open() to get the arguments it needs.
2. Get_Next (): This method calls the Get_next() on each of its input arguments and calls the code specific to the operation being performed on the inputs. The next output tuple generated is returned and the state of the iterator is updated to keep track of the amount of input processed. When no more tuples can be returned, it places some special value in the output buffer.

3. Close(): This method ends the iteration after all tuples that can be generated have been generated, or the required/demanded number of tuples have been returned. It also calls Close() on the arguments of the iterator.

Each iterator may be regarded as a class for its implementation with the above three methods applicable to each instance of that class. If the operator to be implemented allows a tuple to be completely processed when it is received, it may be possible to use the pipelining strategy effectively. However, if the input tuples need to be examined over multiple passes, then the input has to be received as a materialized relation. This becomes tantamount to the Open () method doing most of the work and the benefit of pipelining not being fully achieved. Some physical operators may not lend themselves to the iterator interface concept and hence may not support pipelining.

The iterator concept may also be applied to access methods. Accessing a B^+-tree or a hash-based index may be regarded as a function that can be implemented as an iterator; it produces as output a series of tuples that meet the selection condition passed to the Open() method.

18.8 Parallel Algorithms for Query Processing

In Chapter 2, we mentioned several variations of the client/server architectures, including two-tier and three-tier architectures. There is another type of architecture, called **parallel database architecture**, that is prevalent for data-intensive applications. We will discuss it in further detail in Chapter 23 in conjunction with distributed databases and the big data and NOSQL emerging technologies.

Three main approaches have been proposed for parallel databases. They correspond to three different hardware configurations of processors and secondary storage devices (disks) to support parallelism. In **shared-memory architecture**, multiple processors are attached to an interconnection network and can access a common main memory region. Each processor has access to the entire memory address space from all machines. The memory access to local memory and local cache is faster; memory access to the common memory is slower. This architecture suffers from interference because as more processors are added, there is increasing contention for the common memory. The second type of architecture is known as **shared-disk architecture**. In this architecture, every processor has its own memory, which is not accessible from other processors. However, every machine has access to all disks through the interconnection network. Every processor may not necessarily have a disk of its own. We discussed two forms of enterprise-level secondary storage systems in Section 16.11. Both storage area networks (SANs) and network attached storage (NAS) fall into the shared-disk architecture and lend themselves to parallel processing. They have different units of data transfer; SANs transfer data in units of blocks or pages to and from disks to processors; NAS behaves like a file server that transfers files using some file transfer protocol. In these systems, as more processors are added, there is more contention for the limited network bandwidth.

The above difficulties have led to **shared-nothing architecture** becoming the most commonly used architecture in parallel database systems. In this architecture, each processor accesses its own main memory and disk storage. When a processor A requests data located on the disk D_B attached to processor B, processor A sends the request as a message over a network to processor B, which accesses its own disk D_B and ships the data over the network in a message to processor A. Parallel databases using shared-nothing architecture are relatively inexpensive to build. Today, commodity processors are being connected in this fashion on a rack, and several racks can be connected by an external network. Each processor has its own memory and disk storage.

The shared-nothing architecture affords the possibility of achieving parallelism in query processing at three levels, which we will discuss below: individual operator parallelism, intraquery parallelism, and interquery parallelism. Studies have shown that by allocating more processors and disks, **linear speed-up**—a linear reduction in the time taken for operations—is possible. **Linear scale-up**, on the other hand, refers to being able to give a constant sustained performance by increasing the number of processors and disks proportional to the size of data. Both of these are implicit goals of parallel processing.

18.8.1 Operator-Level Parallelism

In the operations that can be implemented with parallel algorithms, one of the main strategies is to partition data across disks. **Horizontal partitioning** of a relation corresponds to distributing the tuples across disks based on some partitioning method. Given n disks, assigning the ith tuple to disk i mod n is called **round-robin partitioning**. Under **range partitioning**, tuples are equally distributed (as much as possible) by dividing the range of values of some attribute. For example, employee tuples from the EMPLOYEE relation may be assigned to 10 disks by dividing the age range into 10 ranges—say 22–25, 26–28, 29–30, and so on—such that each has roughly one-tenth of the total number of employees. Range partitioning is a challenging operation and requires a good understanding of the distribution of data along the attribute involved in the range clause. The ranges used for partitioning are represented by the **range vector**. With **hash partitioning**, tuple i is assigned to the disk $h(i)$, where h is the hashing function. Next, we briefly discuss how parallel algorithms are designed for various individual operations.

Sorting. If the data has been range partitioned on an attribute—say, age—into n disks on n processors, then to sort the entire relation on age, each partition can be sorted separately in parallel and the results can be concatenated. This potentially causes close to an n-fold reduction in the overall sorting time. If the relation has been partitioned using another scheme, the following approaches are possible:

- Repartition the relation by using range partitioning on the same attribute that is the target for sorting; then sort each partition individually followed by concatenation, as mentioned above.
- Use a parallel version of the external sort-merge algorithm shown in Figure 18.2.

Selection. For a selection based on some condition, if the condition is an equality condition, $<A = v>$ and the same attribute A has been used for range partitioning, the selection can be performed on only that partition to which the value v belongs. In other cases, the selection would be performed in parallel on all the processors and the results merged. If the selection condition is $v1 \leq A \leq v2$ and attribute A is used for range partitioning, then the range of values ($v1$, $v2$) must overlap a certain number of partitions. The selection operation needs to be performed only in those processors in parallel.

Projection and Duplicate Elimination. Projection without duplicate elimination can be achieved by performing the operation in parallel as data is read from each partition. Duplicate elimination can be achieved by sorting the tuples and discarding duplicates. For sorting, any of the techniques mentioned above can be used based on how the data is partitioned.

Join. The basic idea of parallel join is to split the relations to be joined, say R and S, in such a way that the join is divided into multiple n smaller joins, and then perform these smaller joins in parallel on n processors and take a union of the result. Next, we discuss the various techniques involved to achieve this.

a. **Equality-based partitioned join:** If both the relations R and S are partitioned into n partitions on n processors such that partition r_i and partition s_i are both assigned to the same processor P_i, then the join can be computed locally provided the join is an equality join or natural join. Note that the partitions must be non-overlapping on the join key; in that sense, the partitioning is a strict set-theoretic partitioning. Furthermore, the attribute used in the join condition must also satisfy these conditions:
   - It is the same as that used for range partitioning, and the ranges used for each partition are also the same for both R and S. Or,
   - It is the same as that used to partition into n partitions using hash partitioning. The same hash function must be used for R and S. If the distributions of values of the joining attribute are different in R and S, it is difficult to come up with a range vector that will uniformly distribute both R and S into equal partitions. Ideally, the size of $| r_i | + | s_i |$ should be even for all partitions i. Otherwise, if there is too much data skew, then the benefits of parallel processing are not fully achieved. The local join at each processor may be performed using any of the techniques discussed for join: sort merge, nested loop, and hash join.

b. **Inequality join with partitioning and replication:** If the join condition is an inequality condition, involving $<$, $\leq$, $>$, $\geq$, $\neq$, and so on, then it is not possible to partition R and S in such a way that the ith partition of R—namely, r_i— joins the jth partition of S—namely, s_i only. Such a join can be parallelized in two ways:
   - *Asymmetric case*: Partitioning a relation R using one of the partitioning schemes; replicating one of the relations (say S) to all the n partitions; and performing the join between r_i and the entire S at processor P_i. This method is preferred when S is much smaller than R.

- *Symmetric case*: Under this general method, which is applicable to any type of join, both R and S are partitioned. R is partitioned n ways, and S is partitioned m ways. A total of $m * n$ processors are used for the parallel join. These partitions are appropriately replicated so that processors $P_{0,0}$ thru $P_{n-1,m-1}$ (total of $m * n$ processors) can perform the join locally. The processor $P_{i,j}$ performs the join of r_i with s_i using any of the join techniques. The system replicates the partition r_i to processors $P_{i,0}$, $P_{i,1}$ thru $P_{i,m-1}$. Similarly, partition s_i is replicated to processors $P_{0,j}$, $P_{1,j}$, $P_{n-1,j}$. In general, partitioning with replication has a higher cost than just partitioning; thus partitioning with replication costs more in the case of an equijoin.

c. **Parallel partitioned hash join:** The partitioned hash join we described as algorithm J4 in Section 18.4 can be parallelized. The idea is that when R and S are large relations, even if we partition each relation into n partitions equaling the number of processors, the local join at each processor can still be costly. This join proceeds as follows; assume that s is the smaller of r and s:

1. Using a hash function $h1$ on the join attribute, map each tuple of relations r and s to one of the n processors. Let r_i and s_i be the partitions hashed to P_i. First, read the s tuples at each processor on its local disk and map them to the appropriate processor using $h1$.
2. Within each processor P_i, the tuples of S received in step 1 are partitioned using a different hash function $h2$ to, say, k buckets. This step is identical to the partitioning phase of the partitioned hash algorithm we described as J4 in Section 18.4.
3. Read the r tuples from each local disk at each processor and map them to the appropriate processor using hashing function $h1$. As they are received at each processor, the processor partitions them using the same hash function $h2$ used in step 2 for the k buckets; this process is just as in the probing phase of algorithm J4.
4. The processor P_i executes the partitioned hash algorithm locally on the partitions r_i and s_i using the joining phase on the k buckets (as described in algorithm J4) and produces a join result.

The results from all processors P_i are independently computed and unioned to produce the final result.

Aggregation. Aggregate operations with grouping are achieved by partitioning on the grouping attribute and then computing the aggregate function locally at each processor using any of the uni-processor algorithms. Either range partitioning or hash partitioning can be used.

Set Operations. For union, intersection, and set difference operations, if the argument relations R and S are partitioned using the same hash function, they can be done in parallel on each processor. If the partitioning is based on unmatched criteria, R and S may need to be redistributed using an identical hash function.

18.8.2 Intraquery Parallelism

We have discussed how each individual operation may be executed by distributing the data among multiple processors and performing the operation in parallel on those processors. A query execution plan can be modeled as a graph of operations. To achieve a parallel execution of a query, one approach is to use a parallel algorithm for each operation involved in the query, with appropriate partitioning of the data input to that operation. Another opportunity to parallelize comes from the evaluation of an operator tree where some of the operations may be executed in parallel because they do not depend on one another. These operations may be executed on separate processors. If the output of one of the operations can be generated tuple-by-tuple and fed into another operator, the result is **pipelined parallelism**. An operator that does not produce any output until it has consumed all its inputs is said to **block the pipelining**.

18.8.3 Interquery Parallelism

Interquery parallelism refers to the execution of multiple queries in parallel. In shared-nothing or shared-disk architectures, this is difficult to achieve. Activities of locking, logging, and so on among processors (see the chapters in Part 9 on Transaction Processing) must be coordinated, and simultaneous conflicting updates of the same data by multiple processors must be avoided. There must be **cache coherency**, which guarantees that the processor updating a page has the latest version of that page in the buffer. The cache-coherency and concurrency control protocols (see Chapter 21) must work in coordination as well.

The main goal behind interquery parallelism is to scale up (i.e., to increase the overall rate at which queries or transactions can be processed by increasing the number of processors). Because single-processor multiuser systems themselves are designed to support concurrency control among transactions with the goal of increasing transaction throughput (see Chapter 21), database systems using shared memory parallel architecture can achieve this type of parallelism more easily without significant changes.

From the above discussion it is clear that we can speed up the query execution by performing various operations, such as sorting, selection, projection, join, and aggregate operations, individually using their parallel execution. We may achieve further speed-up by executing parts of the query tree that are independent in parallel on different processors. However, it is difficult to achieve interquery parallelism in shared-nothing parallel architectures. One area where the shared-disk architecture has an edge is that it has a more general applicability, since it, unlike the shared-nothing architecture, does not require data to be stored in a partitioned manner. Current SAN- and NAS-based systems afford this advantage. A number of parameters—such as available number of processors and available buffer space—play a role in determining the overall speed-up. A detailed discussion of the effect of these parameters is outside our scope.

18.9 Summary

In this chapter, we gave an overview of the techniques used by DBMSs in processing high-level queries. We first discussed how SQL queries are translated into relational algebra. We introduced the operations of semi-join and anti-join, to which certain nested queries are mapped to avoid doing the regular inner join. We discussed external sorting, which is commonly needed during query processing to order the tuples of a relation while dealing with aggregation, duplicate elimination, and so forth. We considered various cases of selection and discussed the algorithms employed for simple selection based on one attribute and complex selections using conjunctive and disjunctive clauses. Many techniques were discussed for the different selection types, including linear and binary search, use of B^+-tree index, bitmap indexes, clustering index, and functional index. The idea of selectivity of conditions and the typical information placed in a DBMS catalog was discussed. Then we considered the join operation in detail and proposed algorithms called nested-loop join, index-based nested-loop join, sort-merge join, and hash join.

We gave illustrations of how buffer space, join selection factor, and inner–outer relation choice affect the performance of the join algorithms. We also discussed the hybrid hash algorithm, which avoids some of the cost of writing during the joining phase. We discussed algorithms for projection and set operations as well as algorithms for aggregation. Then we discussed the algorithms for different types of joins, including outer joins, semi-join, anti-join, and non-equi-join. We also discussed how operations can be combined during query processing to create pipelined or stream-based execution instead of materialized execution. We introduced how operators may be implemented using the iterator concept. We ended the discussion of query processing strategies with a quick introduction to the three types of parallel database system architectures. Then we briefly summarized how parallelism can be achieved at the individual operations level and discussed intraquery and interquery parallelism as well.

Review Questions

18.1. Discuss the reasons for converting SQL queries into relational algebra queries before optimization is done.

18.2. Discuss semi-join and anti-join as operations to which nested queries may be mapped; provide an example of each.

18.3. How are large tables that do not fit in memory sorted? Give the overall procedure.

18.4. Discuss the different algorithms for implementing each of the following relational operators and the circumstances under which each algorithm can be used: SELECT, JOIN, PROJECT, UNION, INTERSECT, SET DIFFERENCE, CARTESIAN PRODUCT.

18.4. Give examples of a conjunctive selection and a disjunctive selection query and discuss how there may be multiple options for their execution.

18.5. Discuss alternative ways of eliminating duplicates when a "SELECT Distinct <attribute>" query is evaluated.

18.6. How are aggregate operations implemented?

18.7. How are outer join and non–equi-join implemented?

18.8. What is the iterator concept? What methods are part of an iterator?

18.9. What are the three types of parallel architectures applicable to database systems? Which one is most commonly used?

18.10. What are the parallel implementations of join?

18.11. What are intraquery and interquery parallelisms? Which one is harder to achieve in the shared-nothing architecture? Why?

18.12. Under what conditions is pipelined parallel execution of a sequence of operations prevented?

Exercises

18.13. Consider SQL queries Q1, Q8, Q1B, and Q4 in Chapter 6 and Q27 in Chapter 7.

a. Draw at least two query trees that can represent *each* of these queries. Under what circumstances would you use each of your query trees?

b. Draw the initial query tree for each of these queries, and then show how the query tree is optimized by the algorithm outlined in Section 18.7.

c. For each query, compare your own query trees of part (a) and the initial and final query trees of part (b).

18.14. A file of 4,096 blocks is to be sorted with an available buffer space of 64 blocks. How many passes will be needed in the merge phase of the external sort-merge algorithm?

18.15. Can a nondense index be used in the implementation of an aggregate operator? Why or why not? Illustrate with an example.

18.16. Extend the sort-merge join algorithm to implement the LEFT OUTER JOIN operation.

Selected Bibliography

We will give references to the literature for the query processing and optimization area together at the end of Chapter19. Thus the Chapter19 references apply to this chapter and the next chapter. It is difficult to separate the literature that addresses just query processing strategies and algorithms from the literature that discusses the optimization area.

This page intentionally left blank

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.

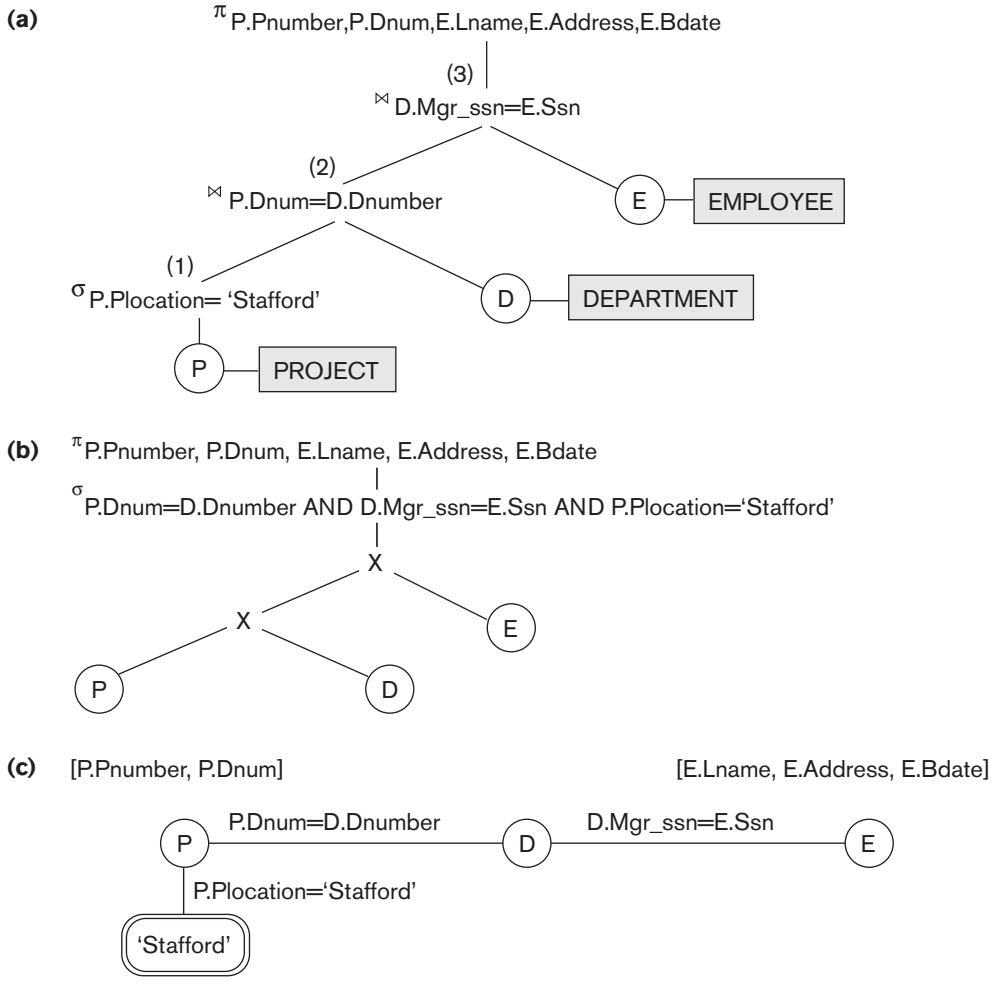


Figure 19.1
Two query trees for the query Q2. (a) Query tree corresponding to the relational algebra expression for Q2. (b) Initial (canonical) query tree for SQL query Q2. (c) Query graph for Q2.

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.

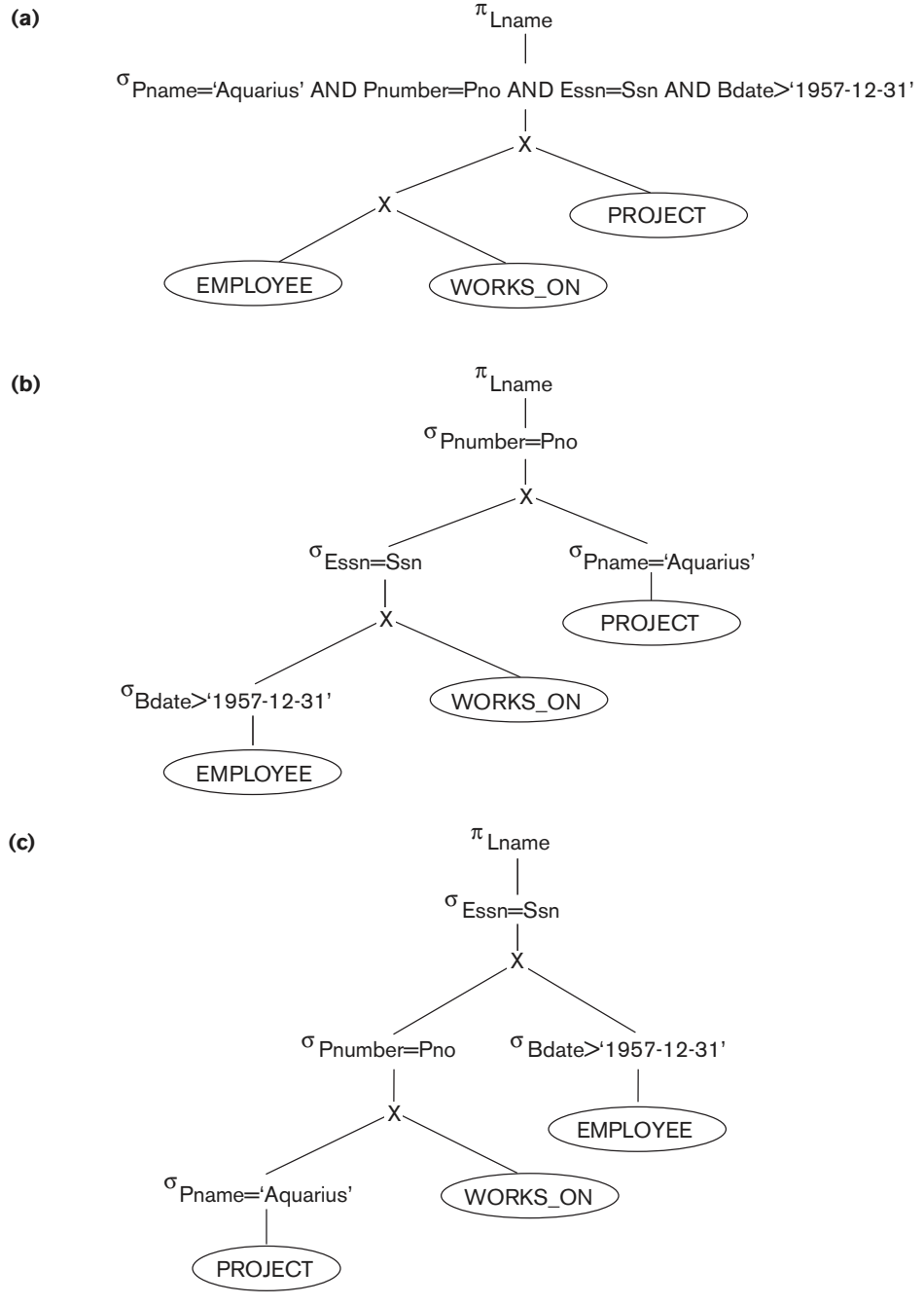
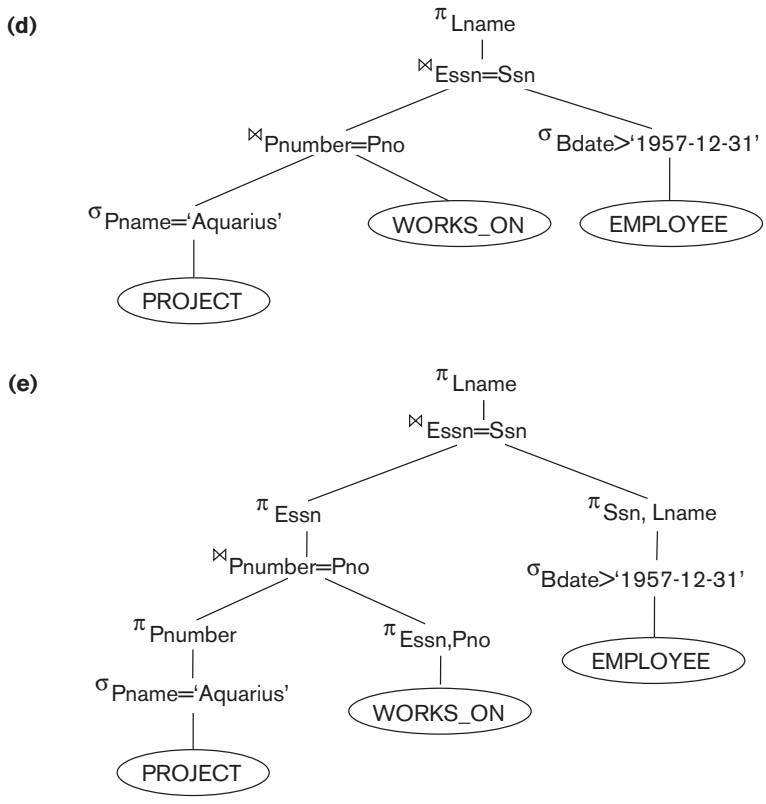


Figure 19.2 (continued)
Steps in converting a query tree during heuristic optimization. (d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations. (e) Moving PROJECT operations down the query tree.



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 $(c_1 \text{ AND } c_2)$, where condition c_1 involves only the attributes of R and condition c_2 involves only the attributes of S, the operations commute as follows:

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

   The same rules apply if the ⋈ 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 $(\sigma, \times)$ sequence into $\bowtie$.** If the condition c of a σ that follows a $\times$ corresponds to a join condition, convert the $(\sigma, \times)$ sequence into a $\bowtie$ as follows:

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

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

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

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

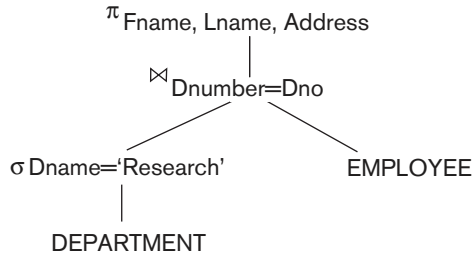


Figure 19.3
A query tree for query Q1.

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 (Ssn, Fn, Ln, Dno)
DEPT (Dno, Dname, Dmgrname, Bldg_id)
BLDG (Bldg_id, 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 $v_{\text{new}} = r \bowtie s \cup r_i \bowtie s$. 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 $V = \sigma_C$ R with condition C for selection, when a set of tuples r_i are inserted into r, the view can be modified as $v_{\text{new}} = v_{old} \cup \sigma_C r_i$. On the other hand, upon deletion of tuples r_d from r, we get $v_{\text{new}} = v_{\text{old}} - \sigma_C r_d$.

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 $\cap$ 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):

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

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 $\pi_L R$ 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 ⋈ 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.

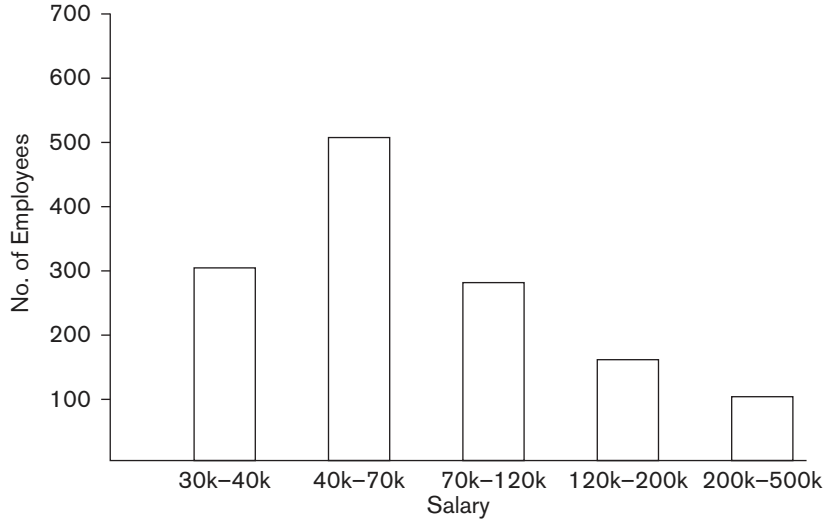
Another important parameter is the **number of distinct values NDV (A, R)** of an attribute in relation *R* and the attribute **selectivity (*sl*)**, which is the fraction of records satisfying an equality condition on the attribute. This allows estimation of the **selection cardinality** (s = ***sl**r***) 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: $\sigma_{\text{Dno}=5\ \text{AND SALARY}>30000\ \text{AND Sex='F'}}$ (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 $C_{S6a} = x_{Ssn} + 1 = 4 + 1 = 5$, 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 $C_{S6b} = x_{Dno} + (b_{I1Dno}/2) + (r_E/2) = 2 + (4/2) + (10{,}000/2) = 5{,}004$), 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 $C_{S6a} = x_{Dno} + s_{Dno} = 2 + 80 = 82$), 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 $C_{S4} = x_{Salary} + (b_E/2) = 3 + (2000/2) = 1003$. Using the condition (Sex = 'F') first gives a cost estimate $C_{S6a} = x_{Sex} + s_{Sex} = 1 + 5000 = 5001$. 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, $s_{Zipcode} = 2{,}000$. 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 $|(R \bowtie_c S)| \leq |R|$, so $js \leq (1/|S|)$.

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 = MIN(1,NDV(Y, T2)/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 $js_{\text{OP6}} = (1/|\text{DEPARTMENT}|) = 1/125$[11] because Dnumber is a key of DEPARTMENT. Also assume that the blocking factor for the resulting join file is $bfr_{\text{ED}} = 4$ 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:

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

4. Using method J2 with DEPARTMENT as outer loop:

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

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 (2*N* − 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

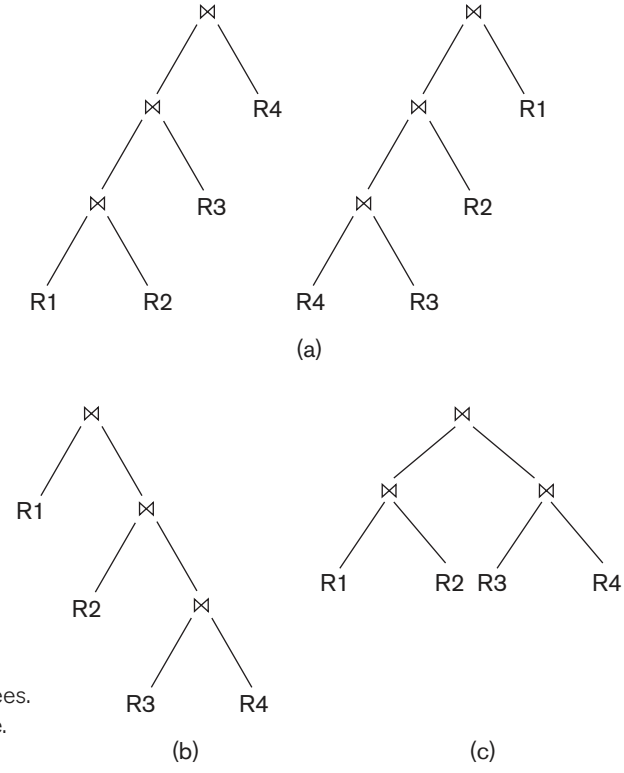


Figure 19.5
(a) Two left-deep join query trees.
(b) A right-deep join query tree.
(c) A bushy query tree.

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.

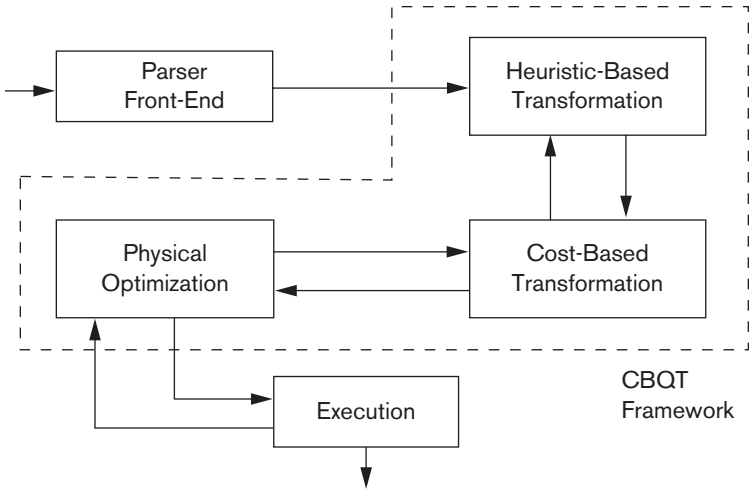


Figure 19.7
Cost-based query transformation framework (based on Ahmed et al., 2006).

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:

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

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

(1984), Shapiro (1986), Kitsuregawa et al. (1989), and Blakeley and Martin (1990), among others. Blakely et al. (1986) discuss maintenance of materialized views. Chaudhari et al. (1995) discuss optimization of queries with materialized views. Approaches to finding a good join order are presented in Ioannidis and Kang (1990) and in Swami and Gupta (1989). A discussion of the implications of left-deep and bushy join trees is presented in Ioannidis and Kang (1991). Kim (1982) discusses transformations of nested SQL queries into canonical representations. Optimization of aggregate functions is discussed in Klug (1982) and Muralikrishna (1992). Query optimization with Group By is presented in Chaudhari and Shim (1994). Yan and Larson (1995) discuss eager and lazy aggregation. Salzberg et al. (1990) describe a fast external sorting algorithm. Estimating the size of temporary relations is crucial for query optimization. Sampling-based estimation schemes are presented in Haas et al. (1995), Haas and Swami (1995), and Lipton et al. (1990). Having the database system store and use more detailed statistics in the form of histograms is the topic of Muralikrishna and DeWitt (1988) and Poosala et al. (1996). Galindo-Legaria and Joshi (2001) discuss nested subquery and aggregation optimization.

O'Neil and Graefe (1995) discuss multi-table joins using bitmap indexes. Kim et al. (1985) discuss advanced topics in query optimization. Semantic query optimization is discussed in King (1981) and Malley and Zdonick (1986). Work on semantic query optimization is reported in Chakravarthy et al. (1990), Shenoy and Ozsoyoglu (1989), and Siegel et al. (1992). Volcano, a query optimizer based on query equivalence rules, was developed by Graefe and Mckenna (1993). Volcano and the follow-on Cascades approach by Graefe (1995) are the basis for Microsoft's SQL Server query optimization. Carey and Kossman (1998) and Bruno et al. (2002) present approaches to query optimization for top-*k* results. Galindo Legaria et al. (2004) discuss processing and optimizing database updates.

Ahmed et al. (2006) discuss cost-based query transformation in Oracle and give a good overview of the global query optimization architecture in Oracle 10g. Ziauddin et al. (2008) discuss the idea of making the optimizer change the execution plan for a query. They discuss Oracle's SQL plan management (SPM) feature, which lends stability to performance. Bellamkonda et al. (2009) provide additional techniques for query optimization. Ahmed et al. (2014) consider the advantages of bushy trees over alternatives for execution. Witkowski et al. (2003) discuss support for *N*-dimensional array-based computation for analytics that has been integrated into the Oracle RDBMS engine.