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.

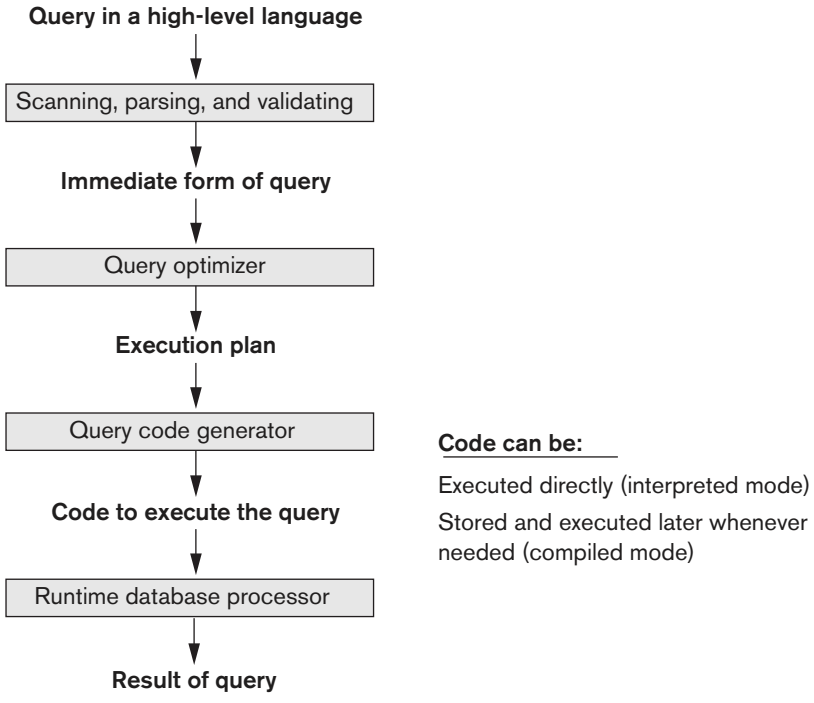

Figure 18.1
Typical steps when processing a high-level query.

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.