user space at the level of the DBMS application software, the OS has no knowledge about them. Hence if the OS deactivates a DBMS process holding a lock, other DBMS processes wanting this locked resource get blocked. Such a situation can cause serious performance degradation. OS-level knowledge of semaphores can help eliminate such situations.

- Specialized hardware support for locking can be exploited to reduce associated costs. This can be of great importance, since locking is one of the most common DBMS operations.
- Providing a set of common transaction support operations though the kernel allows application developers to focus on adding new features to their products as opposed to reimplementing the common functionality for each application. For example, if different DDBMSs are to coexist on the same machine and they chose the two-phase commit protocol, then it is more beneficial to have this protocol implemented as part of the kernel so that the DDBMS developers can focus more on adding new features to their products.

23.5 Query Processing and Optimization in Distributed Databases

Now we give an overview of how a DDBMS processes and optimizes a query. First we discuss the steps involved in query processing and then elaborate on the communication costs of processing a distributed query. Then we discuss a special operation, called a *semijoin,* which is used to optimize some types of queries in a DDBMS. A detailed discussion about optimization algorithms is beyond the scope of this text. We attempt to illustrate optimization principles using suitable examples.[3]

23.5.1 Distributed Query Processing

A distributed database query is processed in stages as follows:

1. **Query Mapping.** The input query on distributed data is specified formally using a query language. It is then translated into an algebraic query on global relations. This translation is done by referring to the global conceptual schema and does not take into account the actual distribution and replication of data. Hence, this translation is largely identical to the one performed in a centralized DBMS. It is first normalized, analyzed for semantic errors, simplified, and finally restructured into an algebraic query.
2. **Localization.** In a distributed database, fragmentation results in relations being stored in separate sites, with some fragments possibly being replicated. This stage maps the distributed query on the global schema to separate queries on individual fragments using data distribution and replication information.

[3]For a detailed discussion of optimization algorithms, see Ozsu and Valduriez (1999).

3. **Global Query Optimization.** Optimization consists of selecting a strategy from a list of candidates that is closest to optimal. A list of candidate queries can be obtained by permuting the ordering of operations within a fragment query generated by the previous stage. Time is the preferred unit for measuring cost. The total cost is a weighted combination of costs such as CPU cost, I/O costs, and communication costs. Since DDBs are connected by a network, often the communication costs over the network are the most significant. This is especially true when the sites are connected through a wide area network (WAN).
4. **Local Query Optimization.** This stage is common to all sites in the DDB. The techniques are similar to those used in centralized systems.

The first three stages discussed above are performed at a central control site, whereas the last stage is performed locally.

23.5.2 Data Transfer Costs of Distributed Query Processing

We discussed the issues involved in processing and optimizing a query in a centralized DBMS in Chapter 19. In a distributed system, several additional factors further complicate query processing. The first is the cost of transferring data over the network. This data includes intermediate files that are transferred to other sites for further processing, as well as the final result files that may have to be transferred to the site where the query result is needed. Although these costs may not be very high if the sites are connected via a high-performance local area network, they become significant in other types of networks. Hence, DDBMS query optimization algorithms consider the goal of reducing the *amount of data transfer* as an optimization criterion in choosing a distributed query execution strategy.

We illustrate this with two simple sample queries. Suppose that the EMPLOYEE and DEPARTMENT relations in Figure 3.5 are distributed at two sites as shown in Figure 23.4. We will assume in this example that neither relation is fragmented. According to Figure 23.4, the size of the EMPLOYEE relation is $100 * 10{,}000 = 10^6$ bytes, and the size of the DEPARTMENT relation is $35 * 100 = 3{,}500$ bytes. Consider the query Q: *For each employee, retrieve the employee name and the name of the department for which the employee works.* This can be stated as follows in the relational algebra:

$$Q: \pi_{Fname,Lname,Dname}(EMPLOYEE \bowtie_{Dno=Dnumber} DEPARTMENT)$$

The result of this query will include 10,000 records, assuming that every employee is related to a department. Suppose that each record in the query result is *40 bytes long*. The query is submitted at a distinct site 3, which is called the **result site** because the query result is needed there. Neither the EMPLOYEE nor the DEPARTMENT relations reside at site 3. There are three simple strategies for executing this distributed query:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case, a total of $1{,}000{,}000 + 3{,}500 = 1{,}003{,}500$ bytes must be transferred.

Site 1:

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

10,000 records
each record is 100 bytes long
Ssn field is 9 bytes long
Fname field is 15 bytes long
Dno field is 4 bytes long
Lname field is 15 bytes long

Site 2:

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

100 records
each record is 35 bytes long
Dnumber field is 4 bytes long
Dname field is 10 bytes long
Mgr_ssn field is 9 bytes long

Figure 23.4
Example to illustrate volume of data transferred.

2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 10,000 = 400,000 bytes, so 400,000 + 1,000,000 = 1,400,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case, 400,000 + 3,500 = 403,500 bytes must be transferred.

If minimizing the amount of data transfer is our optimization criterion, we should choose strategy 3. Now consider another query Q′: *For each department, retrieve the department name and the name of the department manager.* This can be stated as follows in the relational algebra:

$$Q': \pi_{Fname,Lname,Dname}(DEPARTMENT \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE)$$

Again, suppose that the query is submitted at site 3. The same three strategies for executing query Q apply to Q′, except that the result of Q′ includes only 100 records, assuming that each department has a manager:

1. Transfer both the EMPLOYEE and the DEPARTMENT relations to the result site, and perform the join at site 3. In this case, a total of 1,000,000 + 3,500 = 1,003,500 bytes must be transferred.
2. Transfer the EMPLOYEE relation to site 2, execute the join at site 2, and send the result to site 3. The size of the query result is 40 * 100 = 4,000 bytes, so 4,000 + 1,000,000 = 1,004,000 bytes must be transferred.
3. Transfer the DEPARTMENT relation to site 1, execute the join at site 1, and send the result to site 3. In this case, 4,000 + 3,500 = 7,500 bytes must be transferred.

Again, we would choose strategy 3—this time by an overwhelming margin over strategies 1 and 2. The preceding three strategies are the most obvious ones for the

case where the result site (site 3) is different from all the sites that contain files involved in the query (sites 1 and 2). However, suppose that the result site is site 2; then we have two simple strategies:

1. Transfer the EMPLOYEE relation to site 2, execute the query, and present the result to the user at site 2. Here, the same number of bytes—1,000,000—must be transferred for both Q and Q′.
2. Transfer the DEPARTMENT relation to site 1, execute the query at site 1, and send the result back to site 2. In this case 400,000 + 3,500 = 403,500 bytes must be transferred for Q and 4,000 + 3,500 = 7,500 bytes for Q′.

A more complex strategy, which sometimes works better than these simple strategies, uses an operation called **semijoin**. We introduce this operation and discuss distributed execution using semijoins next.

23.5.3 Distributed Query Processing Using Semijoin

The idea behind distributed query processing using the *semijoin operation* is to reduce the number of tuples in a relation before transferring it to another site. Intuitively, the idea is to send the *joining column* of one relation R to the site where the other relation S is located; this column is then joined with S. Following that, the join attributes, along with the attributes required in the result, are projected out and shipped back to the original site and joined with R. Hence, only the joining column of R is transferred in one direction, and a subset of S with no extraneous tuples or attributes is transferred in the other direction. If only a small fraction of the tuples in S participate in the join, this can be an efficient solution to minimizing data transfer.

To illustrate this, consider the following strategy for executing Q or Q′:

1. Project the join attributes of DEPARTMENT at site 2, and transfer them to site 1. For Q, we transfer $F = \pi_{\text{Dnumber}}(\text{DEPARTMENT})$, whose size is $4 * 100 = 400$ bytes, whereas for Q′, we transfer $F' = \pi_{\text{Mgr\_ssn}}(\text{DEPARTMENT})$, whose size is $9 * 100 = 900$ bytes.
2. Join the transferred file with the EMPLOYEE relation at site 1, and transfer the required attributes from the resulting file to site 2. For Q, we transfer $R = \pi_{\text{Dno, Fname, Lname}}(F \ltimes_{\text{Dnumber=Dno}} \text{EMPLOYEE})$, whose size is $34 * 10{,}000 = 340{,}000$ bytes, whereas for Q′, we transfer $R' = \pi_{\text{Mgr\_ssn, Fname, Lname}}(F' \ltimes_{\text{Mgr\_ssn=Ssn}} \text{EMPLOYEE})$, whose size is $39 * 100 = 3{,}900$ bytes.
3. Execute the query by joining the transferred file R or R' with DEPARTMENT, and present the result to the user at site 2.

Using this strategy, we transfer 340,400 bytes for Q and 4,800 bytes for Q′. We limited the EMPLOYEE attributes and tuples transmitted to site 2 in step 2 to only those that will *actually be joined* with a DEPARTMENT tuple in step 3. For query Q, this turned out to include all EMPLOYEE tuples, so little improvement was achieved. However, for Q′ only 100 out of the 10,000 EMPLOYEE tuples were needed.

The semijoin operation was devised to formalize this strategy. A **semijoin operation** $R \ltimes_{A=B} S$, where A and B are domain-compatible attributes of R and S, respectively, produces the same result as the relational algebra expression $\pi R(R \ltimes_{A=B} S)$. In a distributed environment where R and S reside at different sites, the semijoin is typically implemented by first transferring $F = \pi_B(S)$ to the site where R resides and then joining F with R, thus leading to the strategy discussed here.

Notice that the semijoin operation is not commutative; that is,

$$R \ltimes S \neq S \ltimes R$$

23.5.4 Query and Update Decomposition

In a DDBMS with *no distribution transparency,* the user phrases a query directly in terms of specific fragments. For example, consider another query Q: *Retrieve the names and hours per week for each employee who works on some project controlled by department 5*, which is specified on the distributed database where the relations at sites 2 and 3 are shown in Figure 23.2, and those at site 1 are shown in Figure 5.6, as in our earlier example. A user who submits such a query must specify whether it references the PROJS_5 and WORKS_ON_5 relations at site 2 (Figure 23.2) or the PROJECT and WORKS_ON relations at site 1 (Figure 5.6). The user must also maintain consistency of replicated data items when updating a DDBMS with *no replication transparency.*

On the other hand, a DDBMS that supports *full distribution, fragmentation,* and *replication transparency* allows the user to specify a query or update request on the schema in Figure 5.5 just as though the DBMS were centralized. For updates, the DDBMS is responsible for maintaining *consistency among replicated items* by using one of the distributed concurrency control algorithms discussed in Section 23.3. For queries, a **query decomposition** module must break up or **decompose** a query into **subqueries** that can be executed at the individual sites. Additionally, a strategy for combining the results of the subqueries to form the query result must be generated. Whenever the DDBMS determines that an item referenced in the query is replicated, it must choose or **materialize** a particular replica during query execution.

To determine which replicas include the data items referenced in a query, the DDBMS refers to the fragmentation, replication, and distribution information stored in the DDBMS catalog. For vertical fragmentation, the attribute list for each fragment is kept in the catalog. For horizontal fragmentation, a condition, sometimes called a **guard**, is kept for each fragment. This is basically a selection condition that specifies which tuples exist in the fragment; it is called a guard because *only tuples that satisfy this condition* are permitted to be stored in the fragment. For mixed fragments, both the attribute list and the guard condition are kept in the catalog.

In our earlier example, the guard conditions for fragments at site 1 (Figure 5.6) are TRUE (all tuples), and the attribute lists are * (all attributes). For the fragments

(a) EMPD5
attribute list: Fname, Minit, Lname, Ssn, Salary, Super_ssn, Dno
guard condition: Dno = 5
DEP5
attribute list: * (all attributes Dname, Dnumber, Mgr_ssn, Mgr_start_date)
guard condition: Dnumber = 5
DEP5_LOCS
attribute list: * (all attributes Dnumber, Location)
guard condition: Dnumber = 5
PROJS5
attribute list: * (all attributes Pname, Pnumber, Plocation, Dnum)
guard condition: Dnum = 5
WORKS_ON5
attribute list: * (all attributes Essn, Pno,Hours)
guard condition: Essn IN (π_{Ssn} (EMPD5)) OR Pno IN ($\pi_{Pnumber}$ (PROJS5))

(b) EMPD4
attribute list: Fname, Minit, Lname, Ssn, Salary, Super_ssn, Dno
guard condition: Dno = 4
DEP4
attribute list: * (all attributes Dname, Dnumber, Mgr_ssn, Mgr_start_date)
guard condition: Dnumber = 4
DEP4_LOCS
attribute list: * (all attributes Dnumber, Location)
guard condition: Dnumber = 4
PROJS4
attribute list: * (all attributes Pname, Pnumber, Plocation, Dnum)
guard condition: Dnum = 4
WORKS_ON4
attribute list: * (all attributes Essn, Pno, Hours)
guard condition: Essn IN (π_{Ssn} (EMPD4))
OR Pno IN ($\pi_{Pnumber}$ (PROJS4))

Figure 23.5
Guard conditions and attributes lists for fragments.
(a) Site 2 fragments.
(b) Site 3 fragments.

shown in Figure 23.2, we have the guard conditions and attribute lists shown in Figure 23.5. When the DDBMS decomposes an update request, it can determine which fragments must be updated by examining their guard conditions. For example, a user request to insert a new EMPLOYEE tuple <'Alex', 'B', 'Coleman', '345671239', '22-APR-64', '3306 Sandstone, Houston, TX', M, 33000, '987654321', 4> would be decomposed by the DDBMS into two insert requests: the first inserts the preceding tuple in the EMPLOYEE fragment at site 1, and the second inserts the projected tuple <'Alex', 'B', 'Coleman', '345671239', 33000, '987654321', 4> in the EMPD4 fragment at site 3.

For query decomposition, the DDBMS can determine which fragments may contain the required tuples by comparing the query condition with the guard conditions. For

example, consider the query Q: *Retrieve the names and hours per week for each employee who works on some project controlled by department 5*. This can be specified in SQL on the schema in Figure 5.5 as follows:

```
Q:  SELECT Fname, Lname, Hours
    FROM   EMPLOYEE, PROJECT, WORKS_ON
    WHERE  Dnum=5 AND Pnumber=Pno AND Essn=Ssn;
```

Suppose that the query is submitted at site 2, which is where the query result will be needed. The DDBMS can determine from the guard condition on PROJS5 and WORKS_ON5 that all tuples satisfying the conditions (Dnum = 5 AND Pnumber = Pno) reside at site 2. Hence, it may decompose the query into the following relational algebra subqueries:

$$T_1 \leftarrow \pi_{\text{Essn}}(\text{PROJS5} \bowtie_{\text{Pnumber=Pno}} \text{WORKS\_ON5})$$
$$T_2 \leftarrow \pi_{\text{Essn, Fname, Lname}}(T_1 \bowtie_{\text{Essn=Ssn}} \text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Hours}}(T_2 * \text{WORKS\_ON5})$$

This decomposition can be used to execute the query by using a semijoin strategy. The DDBMS knows from the guard conditions that PROJS5 contains exactly those tuples satisfying (Dnum = 5) and that WORKS_ON5 contains all tuples to be joined with PROJS5; hence, subquery T_1 can be executed at site 2, and the projected column Essn can be sent to site 1. Subquery T_2 can then be executed at site 1, and the result can be sent back to site 2, where the final query result is calculated and displayed to the user. An alternative strategy would be to send the query Q itself to site 1, which includes all the database tuples, where it would be executed locally and from which the result would be sent back to site 2. The query optimizer would estimate the costs of both strategies and would choose the one with the lower cost estimate.

23.6 Types of Distributed Database Systems

The term *distributed database management system* can describe various systems that differ from one another in many respects. The main thing that all such systems have in common is the fact that data and software are distributed over multiple sites connected by some form of communication network. In this section, we discuss a number of types of DDBMSs and the criteria and factors that make some of these systems different.

The first factor we consider is the **degree of homogeneity** of the DDBMS software. If all servers (or individual local DBMSs) use identical software and all users (clients) use identical software, the DDBMS is called **homogeneous**; otherwise, it is called **heterogeneous**. Another factor related to the degree of homogeneity is the **degree of local autonomy**. If there is no provision for the local site to function as a standalone DBMS, then the system has **no local autonomy**. On the other hand, if *direct access* by local transactions to a server is permitted, the system has some degree of local autonomy.

Figure 23.6 shows classification of DDBMS alternatives along orthogonal axes of distribution, autonomy, and heterogeneity. For a centralized database, there is