Chapter 1

INTRODUCTION

Distributed database system (DDBS) technology is the union of what appear to be two diametrically opposed approaches to data processing: *database system* and *computer network* technologies. Database systems have taken us from a paradigm of data processing in which each application defined and maintained its own data (Figure 1.1) to one in which the data is defined and administered centrally (Figure 1.2). This new orientation results in *data independence*, whereby the application programs are immune to changes in the logical or physical organization of the data, and vice versa.

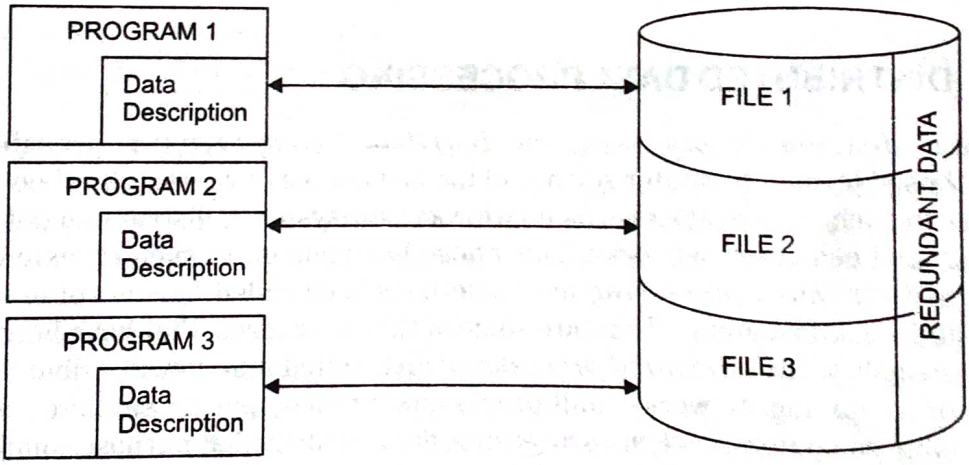

Figure 1.1. Traditional File Processing

One of the major motivations behind the use of database systems is the desire to integrate the operational data of an enterprise and to provide centralized, thus controlled access to that data. The technology of computer networks, on the other hand, promotes a mode of work that goes against all centralization efforts. At first glance it might be difficult to understand how these two contrasting approaches can possibly be synthesized to produce a technology that is more powerful and more promising than either one alone. The key to this understanding is the realization that the most important objective of the database technology is *integration*, not *centralization*. It is important to realize that either one of these terms does not

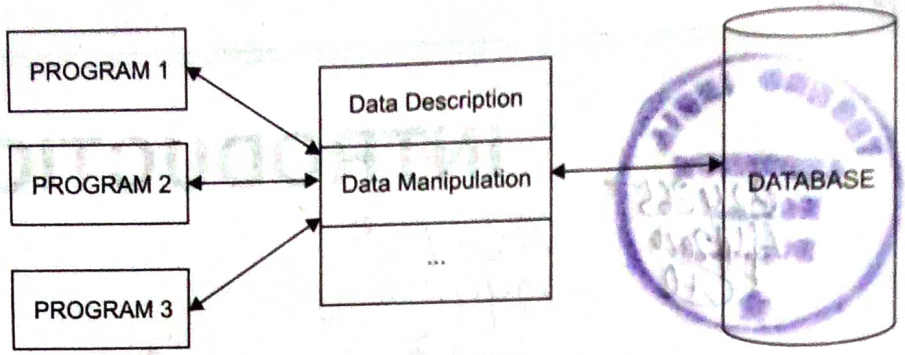

Figure 1.2. Database Processing

necessarily imply the other. It is possible to achieve integration without centralization, and that is exactly what the distributed database technology attempts to achieve.

In this chapter we define the fundamental concepts and set the framework for discussing distributed databases. We start by examining distributed systems in general in order to clarify the role of database technology within distributed data processing, and then move on to topics that are more directly related to DDBS.

1.1 DISTRIBUTED DATA PROCESSING

The term *distributed processing* (or *distributed computing*) is probably the most abused term in computer science of the last couple of years. It has been used to refer to such diverse systems as multiprocessor systems, distributed data processing, and computer networks. This abuse has gone on to such an extent that the term *distributed processing* has sometimes been called "a concept in search of a definition and a name." Here are some of the other terms that have been used synonymously with *distributed processing*: distributed function, distributed computers or computing, networks, multiprocessors/multicomputers, satellite processing/satellite computers, backend processing, dedicated/special-purpose computers, time-shared systems, and functionally modular systems.

Obviously, some degree of distributed processing goes on in any computer system, even on single-processor computers. Starting with the second-generation computers, the central processing unit (CPU) and input/output (I/O) functions have been separated and overlapped. This separation and overlap can be considered as one form of distributed processing. However, it should be quite clear that what we would like to refer to as distributed processing, or distributed computing, has nothing to do with this form of distribution of functions in a single-processor computer system.

A term that has caused so much confusion is obviously quite difficult to define precisely. There have been numerous attempts to define what distributed processing is, and almost every researcher has come up with a definition. In this book we define distributed processing in such a way that it leads to a definition of what

a distributed database system is. The working definition we use for a *distributed computing system* states that it is a number of autonomous processing elements (not necessarily homogeneous) that are interconnected by a computer network and that cooperate in performing their assigned tasks. The "processing element" referred to in this definition is a computing device that can execute a program on its own.

One fundamental question that needs to be asked is: What is being distributed? One of the things that might be distributed is the *processing logic.* In fact, the definition of a distributed computing system given above implicitly assumes that the processing logic or processing elements are distributed. Another possible distribution is according to *function.* Various functions of a computer system could be delegated to various pieces of hardware or software. A third possible mode of distribution is according to *data.* Data used by a number of applications may be distributed to a number of processing sites. Finally, *control* can be distributed. The control of the execution of various tasks might be distributed instead of being performed by one computer system. From the viewpoint of distributed database systems, these modes of distribution are all necessary and important. In the following sections we talk about these in more detail.

1.2 WHAT IS A DISTRIBUTED DATABASE SYSTEM?

We can define a *distributed database* as *a collection of multiple, logically interrelated databases distributed over a computer network.* A *distributed database management system* (distributed DBMS) is then defined as *the software system that permits the management of the DDBS and makes the distribution transparent to the users.* The two important terms in these definitions are *"logically interrelated"* and *"distributed over a computer network."* They help eliminate certain cases that have sometimes been accepted to represent a DDBS.

A DDBS is not a "collection of files" that can be individually stored at each node of a computer network. To form a DDBS, files should not only be logically related, but there should be structure among the files, and access should be via a common interface. We should note that there has been much recent activity in providing DBMS functionality over semi-structured data that are stored in files on the Internet (such as Web pages). In light of this activity, the above requirement may seem unnecessarily strict. However, providing "DBMS-like" access to data is different than a DDBS; in fact, we deal with issues such as these in Chapter 16, where we address Web and database issues.

It has sometimes been assumed that the physical distribution of data is not the most significant issue. The proponents of this view would therefore feel comfortable in labeling as a distributed database two (related) databases that reside in the same computer system. However, the physical distribution of data is very important. It creates problems that are not encountered when the databases reside in the same computer. These difficulties are discussed in Section 1.4. Note that physical distribution does not necessarily imply that the computer systems be

geographically far apart; they could actually be in the same room. It simply implies that the communication between them is done over a network instead of through shared memory, with the network as the only shared resource.

This brings us to another point. The definition above also rules out multiprocessor systems as DDBSs. A multiprocessor system is generally considered to be a system where two or more processors share some form of memory, either primary memory, in which case the multiprocessor is called *shared memory* (also called *tightly coupled*) (Figure 1.3), or secondary memory, when it is called *shared disk* (also called *loosely coupled*) (Figure 1.4)[1].

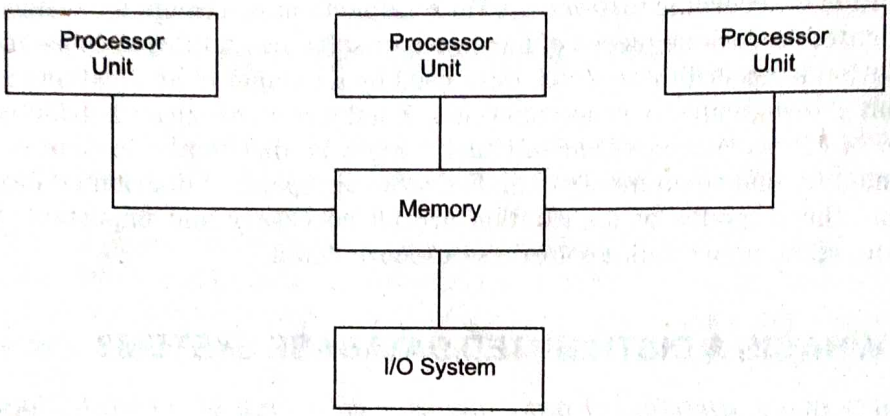

Figure 1.3. Shared Memory Multiprocessor

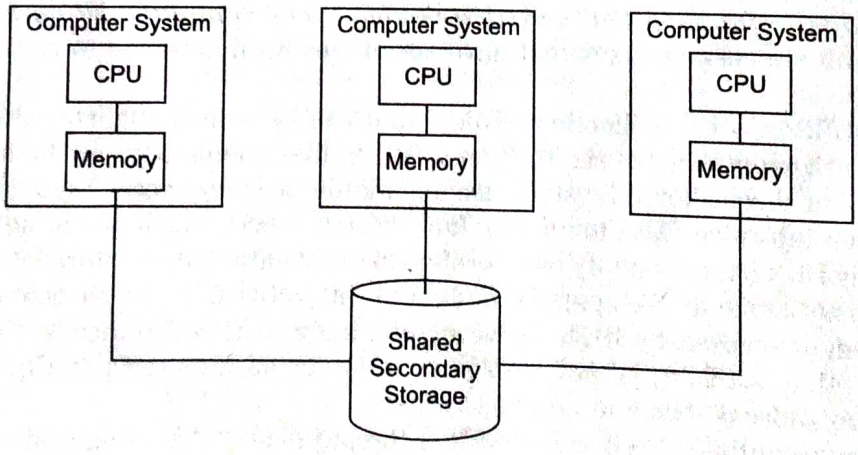

Figure 1.4. Shared Disk Multiprocessor

[1]Note at this point that our definition of coupling modes is different from that of Bochmann discussed in the preceding section. We refer only to coupling in multiprocessors, not to distributed processing in general.

Another distinction that is commonly made in this context is between *shared-everything* and *shared-nothing* architectures. The former architectural model permits each processor to access everything (primary and secondary memories, and peripherals) in the system and covers the two models that we described above. Sharing memory enables the processors to communicate without exchanging messages. The shared-nothing architecture (Figure 1.5) is one where each processor has its own primary and secondary memories as well as peripherals, and communicates with other processors over a very high speed interconnect (e.g., bus or a switch). In this sense the shared-nothing multiprocessors are quite similar to the distributed environment that we consider in this book. However, there are differences between the interactions in multiprocessor architectures and the rather loose interaction that is common in distributed computing environments. The fundamental difference is the mode of operation. A multiprocessor system design is rather symmetrical, consisting of a number of identical processor and memory

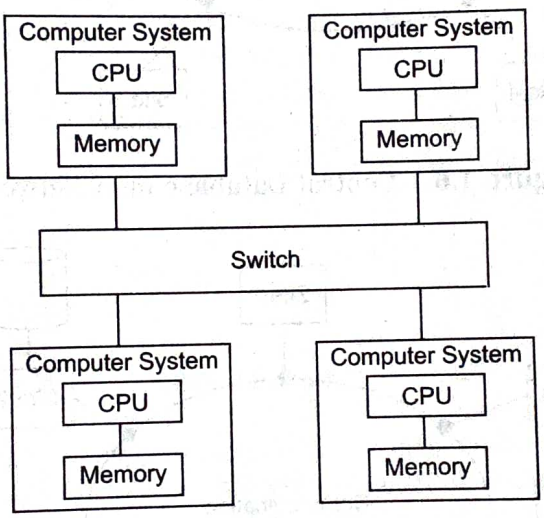

Figure 1.5. Shared Nothing Multiprocessor System

components, and controlled by one or more copies of the same operating system, which is responsible for a strict control of the task assignment to each processor. This is not true in distributed computing systems, where heterogeneity of the operating system as well as the hardware is quite common. We discuss these issues in much more detail in Chapter 13.

In addition, a DDBS is not a system where, despite the existence of a network, the database resides at only one node of the network (Figure 1.6). In this case, the problems of database management are no different from the problems encountered in a centralized database environment[2]. The database is centrally managed by one computer system (site 2 in Figure 1.6) and all the requests are routed to

[2]In Chapter 4, we will discuss client/server systems which relax this requirement to a certain extent.

that site. The only additional consideration has to do with transmission delays. It is obvious that the existence of a computer network or a collection of "files" is not sufficient to form a distributed database system. What we are interested in is an environment where data is distributed among a number of sites (Figure 1.7).

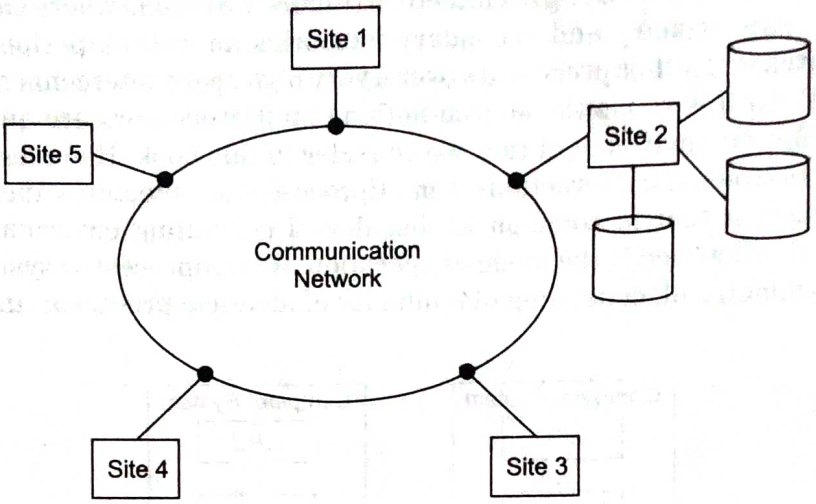

Figure 1.6. Central Database on a Network

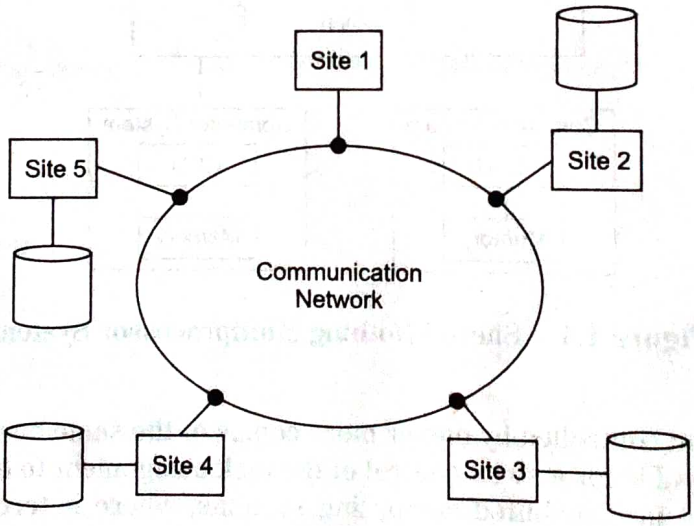

Figure 1.7. DDBS Environment

1.3 PROMISES OF DDBSs

Many advantages of DDBSs have been cited in literature, ranging from sociological reasons for decentralization [D'Oliviera, 1977] to better economics. All of these can be distilled to four fundamentals which may also be viewed as promises of DDBS technology. In this section we discuss these promises and, in the process, introduce many of the concepts that we will study in subsequent chapters.

1.3.1 Transparent Management of Distributed and Replicated Data

Transparency refers to separation of the higher-level semantics of a system from lower-level implementation issues. In other words, a transparent system "hides" the implementation details from users. The advantage of a fully transparent DBMS is the high level of support that it provides for the development of complex applications. It is obvious that we would like to make all DBMSs (centralized or distributed) fully transparent.

Let us start our discussion with an example. Consider an engineering firm that has offices in Boston, Edmonton, Paris and San Francisco. They run projects at each of these sites and would like to maintain a database of their employees, the projects and other related data. Assuming that the database is relational, we can store this information in two relations: EMP(**ENO**, ENAME, TITLE)[3] and PROJ(**PNO**, PNAME, BUDGET). We also introduce a third relation to store salary information: PAY(**TITLE**, SAL) and a fourth relation ASG which indicates which employees have been assigned to which projects for what duration with what responsibility: ASG(**ENO**, **PNO**, DUR, RESP). If all of this data was stored in a centralized DBMS, and we wanted to find out the names and employees who worked on a project for more than 12 months, we would specify this using the following SQL query:

```
SELECT   ENAME, SAL
FROM     EMP, ASG, PAY
WHERE    ASG.DOR > 12
AND      EMP.ENO = ASG.ENO
AND      PAY.TITLE = EMP.TITLE
```

However, given the distributed nature of this firm's business, it is preferable, under these circumstances, to localize each data such that data about the employees in Edmonton office are stored in Edmonton, those in the Boston office are stored in Boston, and so forth. The same applies to the project and salary information. Thus, what we are engaged in is a process where we partition each of the relations and store each partition at a different site. This is known as *fragmentation* and we discuss it further below and in detail in Chapter 5.

Furthermore, it may be preferable to duplicate some of this data at other sites for performance and reliability reasons. The result is a distributed database which is fragmented and replicated (Figure 1.8). Fully transparent access means that the users can still pose the query as specified above, without paying any attention to the fragmentation, location, or replication of data, and let the system worry about resolving these issues.

[3]We discuss relational systems in Chapter 2 where we develop this example further. For the time being, it is sufficient to note that this nomenclature indicates that we have just defined a relation with three attributes: ENO (which is the key), ENAME and TITLE.

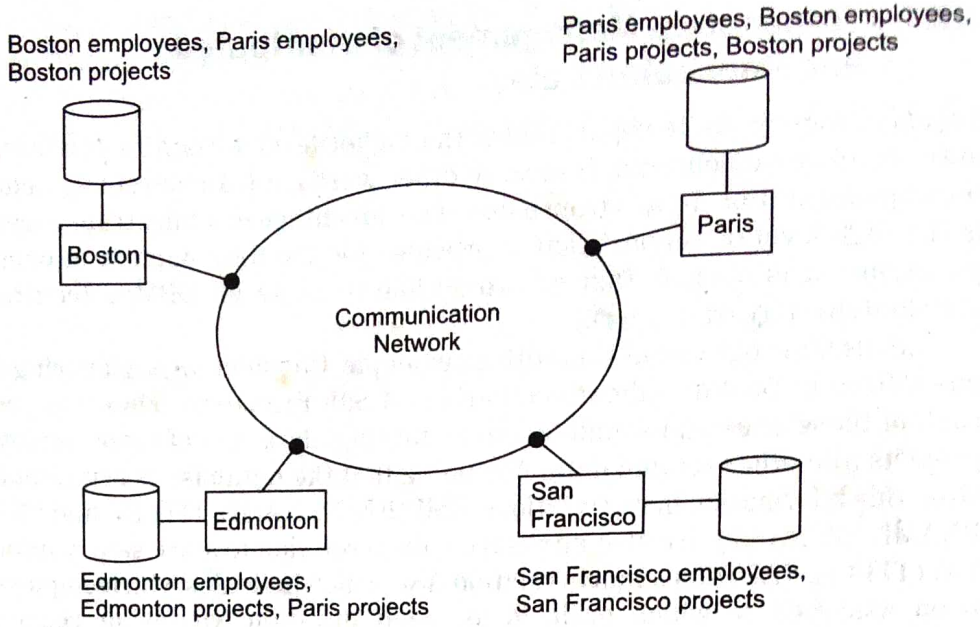

Figure 1.8. A Distributed Application

Data Independence

Data independence is a fundamental form of transparency that we look for within a DBMS. It is also the only type that is important within the context of a centralized DBMS. It refers to the immunity of user applications to changes in the definition and organization of data, and vice versa.

As we will see later in Section 4.1, data definition can occur at two levels. At one level the logical structure of the data is specified, and at the other level the physical structure of the data is defined. The former is commonly known as the *schema definition*; whereas the latter is referred to as the *physical data description*. We can therefore talk about two types of data independence: logical data independence and physical data independence. *Logical data independence* refers to the immunity of user applications to changes in the logical structure of the database. In general, if a user application operates on a subset of the attributes of a relation, it should not be affected later when new attributes are added to the same relation. For example, let us consider the EMP relation discussed above. If a user application deals with only the address fields of this relation (it might be a simple mailing program), the later additions to the relation of say, skill, would not and should not affect the mailing application.

Physical data independence deals with hiding the details of the storage structure from user applications. When a user application is written, it should not be concerned with the details of physical data organization. The data might be organized on different disk types, parts of it might be organized differently (e.g., random versus indexed-sequential access) or might even be distributed across different storage hierarchies (e.g., disk storage and tape storage). The application

should not be involved with these issues since, conceptually, there is no difference in the operations carried out against the data. Therefore, the user application should not need to be modified when data organizational changes occur with respect to these issues. Nevertheless, it is common knowledge that these changes may be necessary for performance considerations.

Network Transparency

In centralized database systems, the only available resource that needs to be shielded from the user is the data (i.e., the storage system). In a distributed database management environment, however, there is a second resource that needs to be managed in much the same manner: the network. Preferably, the user should be protected from the operational details of the network. Furthermore, it is desirable to hide even the existence of the network, if possible. Then there would be no difference between database applications that would run on a centralized database and those that would run on a distributed database. This type of transparency is referred to as *network transparency* or *distribution transparency*.

One can consider network transparency from the viewpoint of either the services provided or the data. From the former perspective, it is desirable to have uniform means by which services are accessed. From a DBMS perspective, distribution transparency requires that users do not have to specify where data is located.

Some have separated distribution transparency into two: location transparency and naming transparency. *Location transparency refers* to the fact that the command used to perform a task is independent of both the location of the data and the system on which an operation is carried out. *Naming transparency* means that a unique name is provided for each object in the database. In the absence of naming transparency, users are required to embed the location name (or an identifier) as part of the object name.

Replication Transparency

The issue of replicating data within a distributed database is discussed in quite some detail in Chapter 5. At this point, let us just mention that for performance, reliability, and availability reasons, it is usually desirable to be able to distribute data in a replicated fashion across the machines on a network. Such replication helps performance since diverse and conflicting user requirements can be more easily accommodated. For example, data that is commonly accessed by one user can be placed on that user's local machine as well as on the machine of another user with the same access requirements. This increases the locality of reference. Furthermore, if one of the machines fails, a copy of the data is still available on another machine on the network. Of course, this is a very simple-minded description of the situation. In fact, the decision as to whether to replicate or not, and how many copies of any database object to have, depends to a considerable degree on user applications. Note that replication causes problems in updating databases.

Therefore, if the user applications are predominantly update-oriented, it may not be a good idea to have too many copies of the data. As this discussion is the subject matter of Chapter 5, we will not dwell further here on the pros and cons of replication.

Assuming that data is replicated, the issue related to transparency that needs to be addressed is whether the users should be aware of the existence of copies or whether the system should handle the management of copies and the user should act as if there is a single copy of the data (note that we are not referring to the placement of copies, only their existence). From a user's perspective the answer is obvious. It is preferable not to be involved with handling copies and having to specify the fact that a certain action can and/or should be taken on multiple copies. From a systems point of view, however, the answer is not that simple. Remember that replication transparency refers only to the existence of replicas, not to their actual location. Note also that distributing these replicas across the network in a transparent manner is the domain of network transparency.

Fragmentation Transparency

The final form of transparency that needs to be addressed within the context of a distributed database system is that of fragmentation transparency. In Chapter 5 we discuss and justify the fact that it is commonly desirable to divide each database relation into smaller fragments and treat each fragment as a separate database object (i.e., another relation). This is commonly done for reasons of performance, availability, and reliability. Furthermore, fragmentation can reduce the negative effects of replication. Each replica is not the full relation but only a subset of it; thus less space is required and fewer data items need be managed.

There are two general types of fragmentation alternatives. In one case, called *horizontal fragmentation*, a relation is partitioned into a set of sub-relations each of which have a subset of the tuples (rows) of the original relation. The second alternative is *vertical fragmentation* where each sub-relation is defined on a subset of the attributes (columns) of the original relation.

When database objects are fragmented, we have to deal with the problem of handling user queries that were specified on entire relations but now have to be performed on subrelations. In other words, the issue is one of finding a query processing strategy based on the fragments rather than the relations, even though the queries are specified on the latter. Typically, this requires a translation from what is called a *global query* to several *fragment queries*. Since the fundamental issue of dealing with fragmentation transparency is one of query processing, we defer the discussion of techniques by which this translation can be performed until Chapter 8.

Who Should Provide Transparency?

In previous sections we discussed various possible forms of transparency within a distributed computing environment. Obviously, to provide easy and efficient

access by novice users to the services of the DBMS, one would want to have full transparency, involving all the various types that we discussed. Nevertheless, the level of transparency is inevitably a compromise between ease of use and the difficulty and overhead cost of providing high levels of transparency.

What has not yet been discussed is who is responsible for providing these services. It is possible to identify three distinct layers at which the services of transparency can be provided. It is quite common to treat these as mutually exclusive means of providing the service, although it is more appropriate to view them as complementary.

We could leave the responsibility of providing transparent access to data resources to the access layer. The transparency features can be built into the user language, which then translates the requested services into required operations. In other words, the compiler or the interpreter takes over the task and no transparent service is provided to the implementer of the compiler or the interpreter.

The second layer at which transparency can be provided is the operating system level. State-of-the-art operating systems provide some level of transparency to system users. For example, the device drivers within the operating system handle the minute details of getting each piece of peripheral equipment to do what is requested. The typical computer user, or even an application programmer, does not normally write device drivers to interact with individual peripheral equipment; that operation is transparent to the user.

Providing transparent access to resources at the operating system level can obviously be extended to the distributed environment, where the management of the network resource is taken over by the distributed operating system. This is a good level at which to provide network transparency if it can be accomplished. The unfortunate aspect is that not all commercially available distributed operating systems provide a reasonable level of transparency in network management.

The third layer at which transparency can be supported is within the DBMS. The transparency and support for database functions provided to the DBMS designers by an underlying operating system is generally minimal and typically limited to very fundamental operations for performing certain tasks. It is the responsibility of the DBMS to make all the necessary translations from the operating system to the higher-level user interface. This mode of operation is the most common method today. There are, however, various problems associated with leaving the task of providing full transparency to the DBMS. These have to do with the interaction of the operating system with the distributed DBMS and are discussed throughout this book.

It is therefore quite important to realize that reasonable levels of transparency depend on different components within the data management environment. Network transparency can easily be handled by the distributed operating system as part of its responsibilities for providing replication and fragmentation transparencies (especially those aspects dealing with transaction management and recovery). The DBMS should be responsible for providing a high level of data independence together with replication and fragmentation transparencies. Finally, the user interface can support a higher level of transparency not only in terms of a uniform access method to the data resources from within a language, but also in terms of

structure constructs that permit the user to deal with objects in his or her environment rather than focusing on the details of database description. Specifically, it should be noted that the interface to a distributed DBMS does not need to be a programming language but can be a graphical user interface, a natural language interface, and even a voice system.

A hierarchy of these transparencies is shown in Figure 1.9. It is not always easy to delineate clearly the levels of transparency, but such a figure serves an important instructional purpose even if it is not fully correct. To complete the picture we have added a "language transparency" layer, although it is not discussed in this chapter. With this generic layer, users have high-level access to the data (e.g., fourth-generation languages, graphical user interfaces, natural language access).

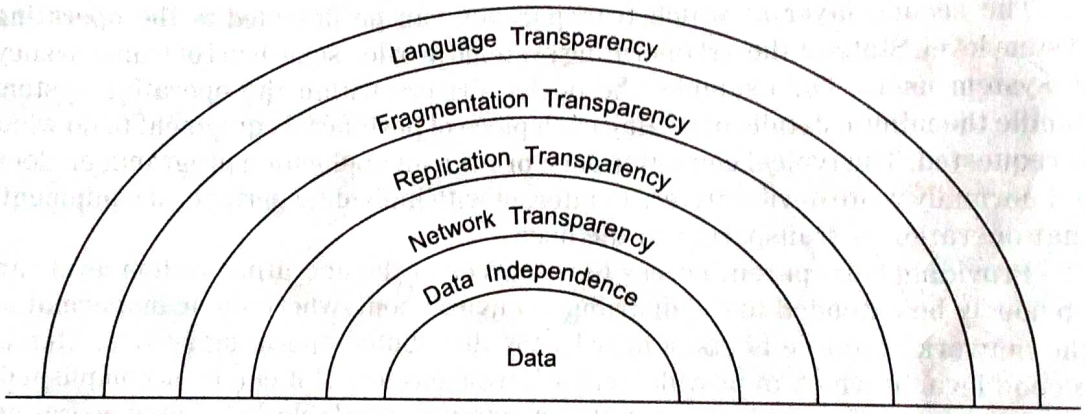

Figure 1.9. Layers of Transparency

How Do Existing Systems Fare?

Most of the commercial distributed DBMSs today have started to provide some level of transparency support. Typically the systems provide distribution transparency, support for horizontal fragmentation and some form of replication transparency.

This level of support is quite recent. Until recently, most commercial distributed DBMSs did not provide a sufficient level of transparency. Some (e.g., R* [Williams et al., 1982]) required users to embed the location names within the name of each database object. Furthermore, they required the user to specify the full name for access to the object. Obviously, one can set up aliases for these long names if the operating system provides such a facility. However, user-defined aliases are not real solutions to the problem in as much as they are attempts to avoid addressing them within the distributed DBMS. The system, not the user, should be responsible for assigning unique names to objects and for translating user-known names to these unique internal object names.

Besides these semantic considerations, there is also a very pragmatic problem associated with embedding location names within object names. Such an approach makes it very difficult to move objects across machines for performance optimi-

zation or other purposes. Every such move will require users to change their access names for the affected objects, which is clearly undesirable.

Other systems did not provide any support for the management of replicated data across multiple logical databases. Even those that did required that the user be physically "logged on" to one database at a given time (e.g., Oracle versions prior to V7).

At this point it is important to point out that full transparency is not a universally accepted objective. Gray argues that full transparency makes the management of distributed data very difficult and claims that "applications coded with transparent access to geographically distributed databases have: poor manageability, poor modularity, and poor message performance" [Gray, 1989]. He proposes a remote procedure call mechanism between the requestor users and the server DBMSs whereby the users would direct their queries to a specific DBMS. It is indeed true that the management of distributed data is more difficult if transparent access is provided to users, and that the client/server architecture (which we discuss in Chapter 4) with a remote procedure call-based communication between the clients and the servers is the right architectural approach. In fact, some commercial distributed DBMSs are organized in this fashion. However, the goal of fully transparent access to distributed and replicated data is an important one and it is up to the system vendors to resolve the system issues.

1.3.2 Reliability Through Distributed Transactions

Distributed DBMSs are intended to improve reliability since they have replicated components and, thereby eliminate single points of failure. The failure of a single site, or the failure of a communication link which makes one or more sites unreachable, is not sufficient to bring down the entire system. In the case of a distributed database, this means that some of the data may be unreachable, but with proper care, users may be permitted to access other parts of the distributed database. The "proper care" comes in the form of support for distributed transactions and application protocols.

We discuss transactions and transaction processing in detail in Chapters 10–12. A *transaction* is a basic unit of consistent and reliable computing, consisting of a sequence of database operations executed as an atomic action. It transforms a consistent database state to another consistent database state even when a number of such transactions are executed concurrently (sometimes called *concurrency transparency*), and even when failures occur (also called *failure atomicity*). Therefore, a DBMS that provides full transaction support guarantees that concurrent execution of user transactions will not violate database consistency in the face of system failures as long as each transaction is correct, i.e., obeys the integrity rules specified on the database.

Let us give an example of a transaction based on the engineering firm example that we discussed above. Assume that there is an application that updates the salaries of all the employees by 10%. It is desirable to encapsulate the query (or the program code) that accomplishes this task within transaction boundaries. For example, if a system failure occurs half-way through the execution of this program,

we would like the DBMS to be able to determine, upon recovery, where it left off and continue with its operation (or start all over again). This is the topic of failure atomicity. Alternatively, if some other user runs a query calculating the average salaries of the employees in this firm while the original update action is going on, the calculated result will be in error. Therefore we would like the system to be able to synchronize the *concurrent* execution of these two programs. To encapsulate a query (or a program code) within transactional boundaries, it is sufficient to declare the begin of the transaction and its end:

```
Begin_transaction SALARY_UPDATE
begin
        EXEC SQL UPDATE PAY
         SET     SAL = SAL*1.1
end.
```

Distributed transactions execute at a number of sites at which they access the local database. The above transaction, for example, will execute in Boston, Edmonton, Paris and San Francisco. With full support for distributed transactions, user applications can access a single logical image of the database and rely on the distributed DBMS to ensure that their requests will be executed correctly no matter what happens in the system. "Correctly" means that user applications do not need to be concerned with coordinating their accesses to individual local databases nor do they need to worry about the possibility of site or communication link failures during the execution of their transactions. This illustrates the link between distributed transactions and transparency, since both involve issues related to distributed naming and directory management, among other things.

Providing transaction support requires the implementation of distributed concurrency control (Chapter 11) and distributed reliability (Chapter 12) protocols—in particular, two-phase commit (2PC) and distributed recovery protocols—which are significantly more complicated than their centralized counterparts. Supporting replicas require the implementation of replica control protocols that enforce a specified semantics of accessing them.

Commercial systems provide varying degrees of distributed transaction support. Some (e.g., Oracle V7 and V8) provide support for distributed transactions while earlier versions of Oracle required users to have one database open at a given time, thereby eliminating the need for distributed transactions, while others (e.g., Sybase) implement the basic primitives that are necessary for the 2PC protocol, but require the user applications to handle the coordination of the commit actions. In other words, the distributed DBMS does not enforce atomicity of distributed transactions, but provide the basic primitives by which user applications can enforce it.

1.3.3 Improved Performance

The case for the improved performance of distributed DBMSs is typically made based on two points:

1. A distributed DBMS fragments the conceptual database, enabling data to be stored in close proximity to its points of use (also called *data localization*). This has two potential advantages:

   - Since each site handles only a portion of the database, contention for CPU and I/O services is not as severe as for centralized databases, and
   - Localization reduces remote access delays that are usually involved in wide area networks (for example, the minimum round-trip message propagation delay in satellite-based systems is about 1 second).

   Most distributed DBMSs are structured to gain maximum benefit from data localization. Full benefits of reduced contention and reduced communication overhead can be obtained only by a proper fragmentation and distribution of the database.

2. The inherent parallelism of distributed systems may be exploited for inter-query and intra-query parallelism. Inter-query parallelism results from the ability to execute multiple queries at the same time while intra-query parallelism is achieved by breaking up a single query into a number of subqueries each of which is executed at a different site, accessing a different part of the distributed database.

The first point relates to the overhead of distributed computing if the data have to reside at remote sites and one has to access it by teleprocessing. The argument is that it is better, in these circumstances, to distribute the data management functionality to where the data is located rather than moving large amounts of data. This has lately become a topic of contention. Some argue that with the widespread use of high-speed, high-capacity networks, distributing data and data management functions no longer make sense and it may be much simpler to store data at a central site and access it (by downloading) over high-speed networks. This argument, while appealing, misses the point of distributed databases. First of all, in most of today's applications, data are distributed; what may be open for debate is how and where we process it. Second, and more important, is that this argument does not distinguish between bandwidth (the capacity of the computer links) and latency (how long it takes for data to be transmitted). Latency is inherent in the distributed environments and there are physical limits to how fast we can send data over computer networks. As indicated above, for example, satellite links take about half-a-second to transmit data between two ground stations. This is a function of the distance of the satellites from the earth and there is nothing that we can do to improve that performance. For some applications, this might constitute an unacceptable delay.

The parallelism argument is also important. If the user access to the distrib uted database consisted only of querying (i.e., read-only access), then provision

of inter-query and intra-query parallelism would imply that as much of the database as possible should be replicated. However, since most database accesses are not read-only, the mixing of read and update operations requires the implementation of elaborate concurrency control and commit protocols.

In addition to optimizing the systems to deal with this issue, some existing commercial systems take a rather interesting approach to deal with the conflict between read-only performance and update performance. They multiplex the database by maintaining two copies. One copy is for ad hoc querying (called the *query database*[4]) and the other for updates by application programs (called the *production database*). At regular intervals, the production database is copied to the query database. This does not eliminate the need to implement concurrency control and reliability protocols for the production database since these are necessary to synchronize the write operations on the same data; however, it improves the performance of the queries since they can be executed without the overhead of transaction manipulation.

In addition to these, there is an administrative measure that some take to deal with the overhead of transaction management. Some installations open their databases only for queries (i.e., read-only access) during the regular operating hours while the updates are batched. The database is then closed to query activity during off-hours when the batched updates are run sequentially. This is time multiplexing between read activity and update activity.

In general, the performance characteristics of distributed database systems are not very well understood. There are not a sufficient number of true distributed database applications to provide a sound base to make practical judgments. In addition, the performance models of distributed database systems are not sufficiently developed. The database community has developed a number of benchmarks to test the performance of transaction processing applications, but it is not clear whether they can be used to measure the performance of distributed transaction management. The performance of the commercial DBMS products, even with respect to these benchmarks, are generally not openly published. NonStop SQL is one product for which performance figures, as well as the experimental setup that is used in obtaining them, has been published [Tandem, 1988].

1.4 COMPLICATING FACTORS

The problems encountered in database systems take on additional complexity in a distributed environment, even though the basic underlying principles are the same. Furthermore, this additional complexity gives rise to new problems influenced mainly by three factors.

First, data may be replicated in a distributed environment. A distributed database can be designed so that the entire database, or portions of it, reside at different sites of a computer network. It is not essential that every site on the network contain the database; it is only essential that there be more than one site

[4]Data warehouses that have attracted much recent attention are of this type.

where the database resides. The possible duplication of data items is mainly due to reliability and efficiency considerations. Consequently, the distributed database system is responsible for (1) choosing one of the stored copies of the requested data for access in case of retrievals, and (2) making sure that the effect of an update is reflected on each and every copy of that data item.

Second, if some sites fail (e.g., by either hardware or software malfunction), or if some communication links fail (making some of the sites unreachable) while an update is being executed, the system must make sure that the effects will be reflected on the data residing at the failing or unreachable sites as soon as the system can recover from the failure.

The third point is that since each site cannot have instantaneous information on the actions currently being carried out at the other sites, the synchronization of transactions on multiple sites is considerably harder than for a centralized system.

These difficulties point to a number of potential problems with distributed DBMSs. These are discussed in the following.

Complexity. DDBS problems are inherently more complex than centralized database management ones, as they include not only the problems found in a centralized environment, but also a new set of unresolved problems. We discuss these new issues shortly.

Cost. Distributed systems require additional hardware (communication mechanisms, etc.), thus have increased hardware costs. However, the trend toward decreasing hardware costs does not make this a significant factor. A more important fraction of the cost lies in the fact that additional and more complex software and communication may be necessary to solve some of the technical problems. The development of software engineering techniques (distributed debuggers and the like) should help in this respect.

Perhaps the most important cost component is due to the replication of effort (manpower). When computer facilities are set up at different sites, it becomes necessary to employ people to maintain these facilities. This usually results in an increase in the personnel in the data processing operations. Therefore, the trade-off between increased profitability due to more efficient and timely use of information and the increased personnel costs has to be analyzed carefully.

Distribution of Control. This point was stated previously as an advantage of DDBSs. Unfortunately, distribution creates problems of synchronization and coordination (the reasons for this added complexity are studied in the next section). Distributed control can therefore easily become a liability if care is not taken to adopt adequate policies to deal with these issues.

Security. One of the major benefits of centralized databases has been the control it provides over the access to data. Security can easily be controlled in one central location, with the DBMS enforcing the rules. However, in a distributed database system, a network is involved which is a medium that has its own security requirements. It is well known that there are serious problems in maintaining adequate security over computer networks. Thus the security problems in distributed database systems are by nature more complicated than in centralized ones.

1.5 PROBLEM AREAS

In Section 1.3, we discussed a number of technical problems that need to be resolved to realize the full potential of distributed DBMSs. In this section we discuss them in a more organized fashion as a prelude to our in-depth studies in the rest of the book.

1.5.1 Distributed Database Design

The question that is being addressed is how the database and the applications that run against it should be placed across the sites. There are two basic alternatives to placing data: *partitioned* (or *nonreplicated*) and *replicated*. In the partitioned scheme the database is divided into a number of disjoint partitions each of which is placed at a different site. Replicated designs can be either *fully replicated* (also called *fully duplicated*) where the entire database is stored at each site, or *partially replicated* (or *partially duplicated*) where each partition of the database is stored at more than one site, but not at all the sites. The two fundamental design issues are *fragmentation*, the separation of the database into partitions called *fragments*, and *distribution*, the optimum distribution of fragments.

The research in this area mostly involves mathematical programming in order to minimize the combined cost of storing the database, processing transactions against it, and communication. The general problem is NP-hard. Therefore, the proposed solutions are based on heuristics.

1.5.2 Distributed Query Processing

Query processing deals with designing algorithms that analyze queries and convert them into a series of data manipulation operations. The problem is how to decide on a strategy for executing each query over the network in the most cost-effective way, however cost is defined. The factors to be considered are the distribution of data, communication costs, and lack of sufficient locally-available information. The objective is to optimize where the inherent parallelism is used to improve the performance of executing the transaction, subject to the above-mentioned constraints. The problem is NP-hard in nature, and the approaches are usually heuristic.

1.5.3 Distributed Directory Management

A directory contains information (such as descriptions and locations) about data items in the database. Problems related to directory management are similar in nature to the database placement problem discussed in the preceding section. A directory may be global to the entire DDBS or local to each site; it can be centralized at one site or distributed over several sites; there can be a single copy or multiple copies.

1.5.4 Distributed Concurrency Control

Concurrency control involves the synchronization of accesses to the distributed database, such that the integrity of the database is maintained. It is, without any doubt, one of the most extensively studied problems in the DDBS field. The concurrency control problem in a distributed context is somewhat different than in a centralized framework. One not only has to worry about the integrity of a single database, but also about the consistency of multiple copies of the database. The condition that requires all the values of multiple copies of every data item to converge to the same value is called *mutual consistency.*

The alternative solutions are too numerous to discuss here, so we examine them in detail in Chapter 11. Let us only mention that the two general classes are *pessimistic*, synchronizing the execution of user requests before the execution starts, and *optimistic*, executing the requests and then checking if the execution compromised the consistency of the database. Two fundamental primitives that can be used with both approaches are *locking*, which is based on the mutual exclusion of accesses to data items, and *timestamping*, where the transactions are executed in some order. There are variations of these schemes as well as hybrid algorithms that attempt to combine the two basic mechanisms.

1.5.5 Distributed Deadlock Management

The deadlock problem in DDBSs is similar in nature to that encountered in operating systems. The competition among users for access to a set of resources (data, in this case) can result in a deadlock if the synchronization mechanism is based on locking. The well-known alternatives of prevention, avoidance, and detection/recovery also apply to DDBSs.

1.5.6 Reliability of Distributed DBMS

We mentioned earlier that one of the potential advantages of distributed systems is improved reliability and availability. This, however, is not a feature that comes automatically. It is important that mechanisms be provided to ensure the consistency of the database as well as to detect failures and recover from them. The implication for DDBBs is that when a failure occurs and various sites become either inoperable or inaccessible, the databases at the operational sites remain consistent and up to date. Furthermore, when the computer system or network recovers from the failure, the DDBSs should be able to recover and bring the databases at the failed sites up-to-date. This may be especially difficult in the case of network partitioning, where the sites are divided into two or more groups with no communication among them.

1.5.7 Operating System Support

The current implementation of distributed database systems on top of (or under) the conventional operating systems suffers from the performance bottleneck. The

support provided by operating systems for database operations does not correspond properly to the requirements of the database management software. The major operating system-related problems in single-processor systems are memory management, file system and access methods, crash recovery, and process management. In distributed environments there is the additional problem of having to deal with multiple layers of network software. The work in this area is on finding solutions to the dichotomy of providing adequate and simple support for distributed database operations, as well as providing general operating system support for other applications.

1.5.8 Heterogeneous Databases

When there is no homogeneity among the databases at various sites either in terms of the way data is logically structured (data model) or in terms of the mechanisms provided for accessing it (data language), it becomes necessary to provide a translation mechanism between database systems. This translation mechanism usually involves a canonical form to facilitate data translation, as well as program templates for translating data manipulation instructions.

It turns out that heterogeneity is typically introduced if one is constructing a distributed DBMS from a number of autonomous, centralized DBMSs. In this setting the problems are more general than heterogeneity. In fact, such systems, which we call *multidatabase systems*, should be considered complementary to the distributed DBMSs as defined in this chapter. Thus, all the problems that we have discussed in the preceding sections have complementary specifications for multidatabase systems. We discuss these systems in Chapter 15.

1.5.9 Relationship among Problems

We should mention at this point that these problems are not isolated from one another. The reasons for studying them in isolation are that (1) problems are difficult enough to study by themselves, and would probably be impossible to present all together, and that (2) it might be possible to characterize the effect of one problem on another one, through the use of parameters and constraints. In fact, each problem is affected by the solutions found for the others, and in turn affects the set of feasible solutions for them. In this section we discuss how they are related.

The relationship among the components is shown in Figure 1.10. The design of distributed databases affects many areas. It affects directory management, because the definition of fragments and their placement determine the contents of the directory (or directories) as well as the strategies that may be employed to manage them. The same information (i.e., fragment structure and placement) is used by the query processor to determine the query evaluation strategy. On the other hand, the access and usage patterns that are determined by the query processor are used as inputs to the data distribution and fragmentation algorithms. Similarly, directory placement and contents influence the processing of queries.

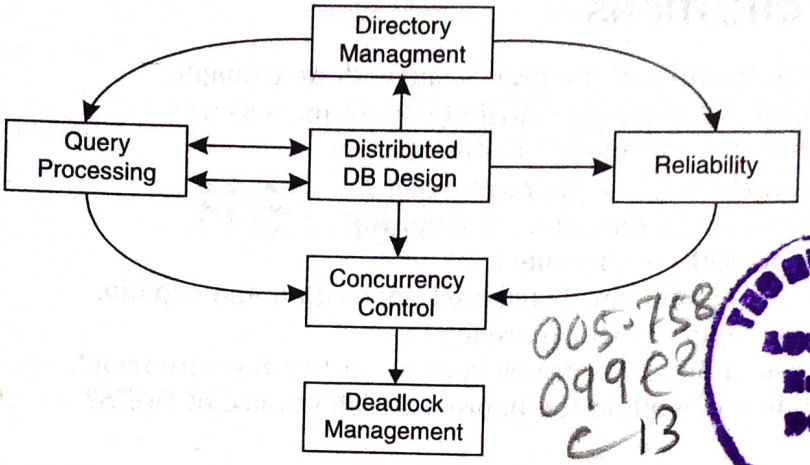

Figure 1.10. Relationship Among Research Issues

The replication of fragments when they are distributed affects the concurrency control strategies that might be employed. As we will study in Chapter 11, some concurrency control algorithms cannot be easily used with replicated databases. Similarly, usage and access patterns to the database will influence the concurrency control algorithms. If the environment is update intensive, the necessary precautions are quite different from those in a query-only environment.

There is a strong relationship among the concurrency control problem, the deadlock management problem, and reliability issues. This is to be expected, since together they are usually called the *transaction management* problem. The concurrency control algorithm that is employed will determine whether or not a separate deadlock management facility is required. If a locking-based algorithm is used, deadlocks will occur, whereas they will not if timestamping is the chosen alternative.

Reliability mechanisms are implemented on top of a concurrency control algorithm. Therefore, the relationship among them is self-explanatory. It should also be mentioned that the reliability mechanisms being considered have an effect on the choice of the concurrency control algorithm. Techniques to provide reliability also make use of data placement information since the existence of duplicate copies of the data serve as a safeguard to maintain reliable operation.

Two of the problems we discussed in the preceding sections—operating system issues and heterogeneous databases—are not illustrated in Figure 1.10. This is obviously not because they have no bearing on other issues; in fact, exactly the opposite is true. The type of operating system used and the features supported by that operating system greatly influence what solution strategies can be applied in any of the other problem areas. Similarly, the nature of all these problems change considerably when the environment is heterogeneous. The same issues have to be dealt with differently when the machine architecture, the operating systems, and the local database management software vary from site to site.

REVIEW QUESTIONS

1.1 Explain traditional file processing with an example.
1.2 What do you mean by distributed data processing?
1.3 Explain shared memory multiprocessor.
1.4 Explain shared disk multiprocessor.
1.5 Explain central database on a network.
1.6 Explain DDBS environment.
1.7 Give an example for distributed application and explain.
1.8 Explain layers of transparency.
1.9 Discuss in detail the problem areas in DDBS environment.
1.10 How do you explain the improved performance of DDBS?