Chapter 4

DISTRIBUTED DBMS ARCHITECTURE

The architecture of a system defines its structure. This means that the components of the system are identified, the function of each component is specified, and the interrelationships and interactions among these components are defined. This general framework also holds true for computer systems in general and software systems in particular. The specification of the architecture of a software system requires identification of the various modules, with their interfaces and interrelationships, in terms of the data and control flow through the system. From a software engineering perspective, the task of developing individual modules is called *programming-in-the-small*, whereas the task of integrating them into a complete system is referred to as *programming-in-the-large*.

Since we are treating distributed DBMSs as large-scale software systems, we can define their architecture in a similar manner. In this chapter we develop three "reference" architectures for a distributed DBMS: client/server systems, peer-to-peer distributed DBMS, and multidatabase systems. These are "idealized" views of a DBMS in that many of the commercially available systems may deviate from them; however, the architectures will serve as a reasonable framework within which the issues related to distributed DBMS can be discussed. A reference architecture is commonly created by standards developers since it clearly defines the interfaces that need to be standardized. For example, the ISO/OSI model discussed in Chapter 3 is a reference architecture for wide area computer networks.

We started discussing the architectural features of relational DBMSs in Chapter 2. In this chapter we extend the discussion by studying generic architectures. The primary objective of the architecture definition is to structure the distributed DBMS such that it provides the functionality identified in Section 1.3. In particular, the transparency levels identified in that section are important, because the structure of a system should match the level of transparency one wants to provide. We start by looking at the DBMS standardization efforts (Section 4.1) and present a well-known reference architecture for centralized DBMSs. This is followed (Section 4.2) by a study of the design space for distributed DBMS implementation. We identify the alternatives and give examples. The reference architecture for

distributed DBMSs that is used in the remainder of this book is introduced in Section 4.3. Finally, in Section 4.4, global directory issues are dealt with.

4.1 DBMS STANDARDIZATION

In this section we discuss the standardization efforts related to DBMSs because of the close relationship between the architecture of a system and the reference model of that system, which is developed as a precursor to any standardization activity. For all practical purposes, the reference model can be thought of as an idealized architectural model of the system. It is defined as "a conceptual framework whose purpose is to divide standardization work into manageable pieces, and to show at a general level how these pieces are related with each other" [DAFTG, 1986]. A reference model (and therefore a system architecture) can be described according to three different approaches [Kangassalo, 1983]:

1. Based on *components*. The components of the system are defined together with the interrelationships between components. Thus a DBMS consists of a number of components, each of which provides some functionality. Their orderly and well-defined interaction provides total system functionality. This is a desirable approach if the ultimate objective is to design and implement the system under consideration. On the other hand, it is difficult to determine the functionality of a system by examining its components. The DBMS standard proposals prepared by the Computer Corporation of America for the National Bureau of Standards ([CCA, 1980], [CCA, 1982]) fall within this category.

2. Based on *functions*. The different classes of users are identified and the functions that the system will perform for each class are defined. The system specifications within this category typically specify a hierarchical structure for user classes. This results in a hierarchical system architecture with well-defined interfaces between the functionalities of different layers. The ISO/OSI architecture discussed in Chapter 3 [ISO, 1983] fall in this category. The advantage of the functional approach is the clarity with which the objectives of the system are specified. However, it gives very little insight into how these objectives will be attained or the level of complexity of the system.

3. Based on *data*. The different types of data are identified, and an architectural framework is specified which defines the functional units that will realize or use data according to these different views. Since data is the central resource that a DBMS manages, this approach (also referred as the *datalogical* approach) is claimed to be the preferable choice for standardization activities [DAFTG, 1986]. The advantage of the data approach is the central importance it associates with the data resource. This is significant from the DBMS viewpoint since the fundamental resource that a DBMS manages is data. On the other hand, it is impossible to specify an architectural model fully unless the functional modules are

also described. The ANSI/SPARC architecture [Tsichritzis and Klug, 1978] discussed in the next section belongs in this category.

Even though three distinct approaches are identified, one should never lose sight of the interplay among them. As indicated in a report of the Database Architecture Framework Task Group of ANSI [DAFTG, 1986], all three approaches need to be used together to define an architectural model, with each point of view serving to focus our attention on different aspects of an architectural model.

A more important issue is the orthogonality of the foregoing classification schemes and the DBMS objectives (e.g., functionality, performance, etc.). Regardless of how we choose to view a DBMS, these objectives have to be taken into account. For example, in the functional approach, the objectives have to be addressed within each functional unit (e.g., query processor, transaction manager, etc.). In the remainder of this section we concentrate on a reference architecture that has generated considerable interest and is the basis of our reference model, described in Section 4.3.

In late 1972, the Computer and Information Processing Committee (X3) of the American National Standards Institute (ANSI) established a Study Group on Database Management Systems under the auspices of its Standards Planning and Requirements Committee (SPARC). The mission of the study group was to study the *feasibility* of setting up standards in this area, as well as determining which aspects should be standardized if it was feasible. The study group issued its interim report in 1975 [SPARC, 1975], and its final report in 1977 [Tsichritzis and Klug, 1978]. The architectural framework proposed in these reports came to be known as the "ANSI/SPARC architecture," its full title being "ANSI/X3/SPARC DBMS Framework." The study group proposed that the interfaces be standardized, and defined an architectural framework that contained 43 interfaces, 14 of which would deal with the physical storage subsystem of the computer and therefore not be considered essential parts of the DBMS architecture.

With respect to our earlier discussion on alternative approaches to standardization, the ANSI/SPARC architecture is claimed to be based on the data organization. It recognizes three views of data: the *external view*, which is that of the user, who might be a programmer; the *internal view*, that of the system or machine; and the *conceptual view*, that of the enterprise. For each of these views, an appropriate schema definition is required. Figure 4.1 depicts the ANSI/SPARC architecture from the data organization perspective.

At the lowest level of the architecture is the internal view, which deals with the physical definition and organization of data. The location of data on different storage devices and the access mechanisms used to reach and manipulate data are the issues dealt with at this level. At the other extreme is the external view, which is concerned with how users view the database. An individual user's view represents the portion of the database that will be accessed by that user as well as the relationships that the user would like to see among the data. A view can be shared among a number of users, with the collection of user views making up the external schema. In between these two ends is the conceptual schema, which is an abstract definition of the database. It is the "real world" view of the enterprise being modeled in the database [Yormark, 1977]. As such, it is supposed to represent

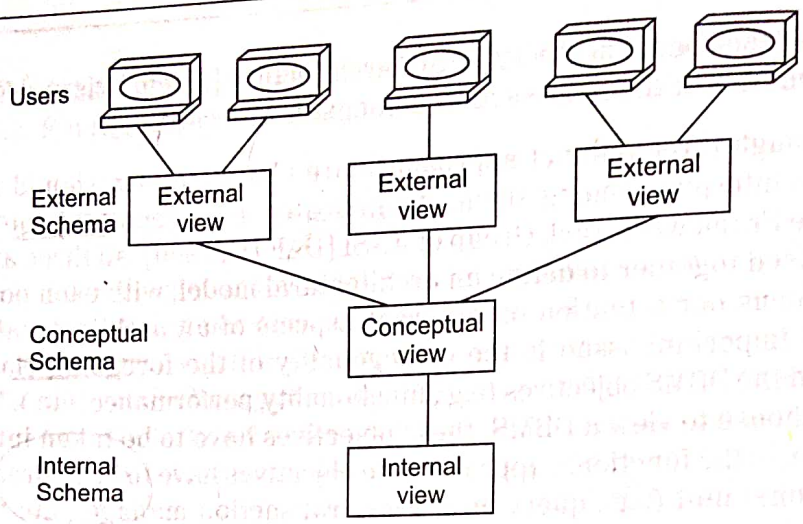

Figure 4.1. The ANSI/SPARC Architecture

the data and the relationships among data without considering the requirements of individual applications or the restrictions of the physical storage media. In reality, however, it is not possible to ignore these requirements completely, due to performance reasons. The transformation between these three levels is accomplished by mappings that specify how a definition at one level can be obtained from a definition at another level.

Example 4.1

Let us consider the engineering database example we have been using and indicate how it can be described using a fictitious DBMS that conforms to the ANSI/SPARC architecture. Remember that we have four relations: EMP, PROJ, ASG, and PAY. The conceptual schema should describe each relation with respect to its attributes and its key. The description might look like the following:[1]

```
RELATION EMP    [
    KEY = {END}
    ATTRIBUTES = {
        END             : CHARACTER(9)
        ENAME           : CHARACTER(15)
        TITLE           : CHARACTER(10)
        }       ]
RELATION PAY  [
    KEY = {TITLE}
    ATTRIBUTES = {
        TITLE           : CHARACTER(10)
        SAL             : NUMERIC(6)
        }       ]
```

[1]Two points to note here. First, we are using the relational representation for the conceptual schema, but by no means do we suggest that the relational model is the only suitable formalism at the conceptual level. Second, the syntax of the description does not conform to any programming language.

```
RELATION PROJ [
    KEY = {PNO}
    ATTRIBUTES = {
        PNO      : CHARACTER(7)
        PNAME    : CHARACTER(20)
        BUDGET   : NUMERIC(7)
        }    ]
RELATION ASG [
    KEY = {END,PNO}
    ATTRIBUTES = {
        ENO      : CHARACTER(9)
        PNO      : CHARACTER(7)
        RESP     : CHARACTER(10)
        DUR      : NUMERIC(3)
        }    ]
```

At the internal level, the storage details of these relations are described. Let us assume that the EMP relation is stored in an indexed file, where the index is defined on the key attribute (i.e., the ENO) called EMINX.[2] Let us also assume that we associate a HEADER field which might, contain flags (delete, update, etc.) and other control information. Then the internal schema definition of the relation may be as follows:

```
INTERNAL_REL EMPL [
    INDEX ON E# CALL EMINX
    FIELD = {
        HEADER    : BYTE(1)
        E#        : BYTE(9)
        E:NAME    : BYTE(15)
        TIT       : BYTE(10)
        }
    ]
```

We have used similar syntaxes for both the conceptual and the internal descriptions. This is done for convenience only and does not imply the true nature of languages for these functions.

Finally, let us consider the external views, which we will describe using SQL notation. We consider two applications: one that calculates the payroll payments for engineers, and a second that produces a report on the budget of each project.[3] Notice that for the first application, we need attributes from both the EMP and the PAY relations. In other words, the view consists of a join, which can be defined as

```
CREATE    VIEW    PAYROLL (ENO, ENAME, SAL)
AS        SELECT  EMP.END,
                  EMP.ENAME,
                  PAY.SAL
```

[2] To keep the presentation simple, we will not concern ourselves with the details of indexing. Consider EMINX to be a primary index.

[3] For simplicity, we will ignore semantic data control aspects of external view generation. These issues are discussed in Chapter 6.

```
FROM    EMP, PAY
WHERE   EMP.TITLE = PAY.TITLE
```

The second application is simply a projection of the PROJ relation, which can be specified as

```
CREATE    VIEW    BUDGET(PNAME, BUD)
AS        SELECT  PNAME, BUDGET
          FROM    PROJ
```

The investigation of the ANSI/SPARC architecture with respect to its functions results in a considerably more complicated view, as depicted in Figure 4.2.[4] The square boxes represent processing functions, whereas the hexagons are administrative roles. The arrows indicate data, command, program, and description flow, whereas the "I"-shaped bars on them represent interfaces.

The major component that permits mapping between different data organizational views is the data dictionary/directory (depicted as a triangle), which is a meta-database. It should at least contain schema and mapping definitions. It may also contain usage statistics, access control information, and the like. It is clearly seen that the data dictionary/directory serves as the central component in both processing different schemas and in providing mappings among them.

We also see in Figure 4.2 a number of administrator roles, which might help to define a functional interpretation of the ANSI/SPARC architecture. The three roles are the database administrator, the enterprise administrator, and the application administrator. The database administrator is responsible for defining the internal schema definition. The enterprise administrator's role is to prepare the conceptual schema definition. The person in this role is the focal point of the use of information within an enterprise. Finally, the application administrator is responsible for preparing the external schema for applications. Note that these are roles that might be fulfilled by one particular person or by several people. Hopefully, the system will provide sufficient support for these roles.

In addition to these three classes of administrative user defined by the roles, there are two more, the application programmer and the system programmer. Two more user classes can be defined, namely casual users and novice end users. Casual users occasionally access the database to retrieve and possibly to update information. Such users are aided by the definition of external schemas and by an easy-to-use query language. Novice users typically have no knowledge of databases and access information by means of predefined menus and transactions (e.g., banking machines).

4.2 ARCHITECTURAL MODELS FOR DISTRIBUTED DBMSs

Let us consider the possible ways in which multiple databases may be put together for sharing by multiple DBMSs. We use a classification (Figure 4.3) that organizes

[4]This is only a part of the system schematic that is. provided in [Tsichritzis and Klug, 1978].

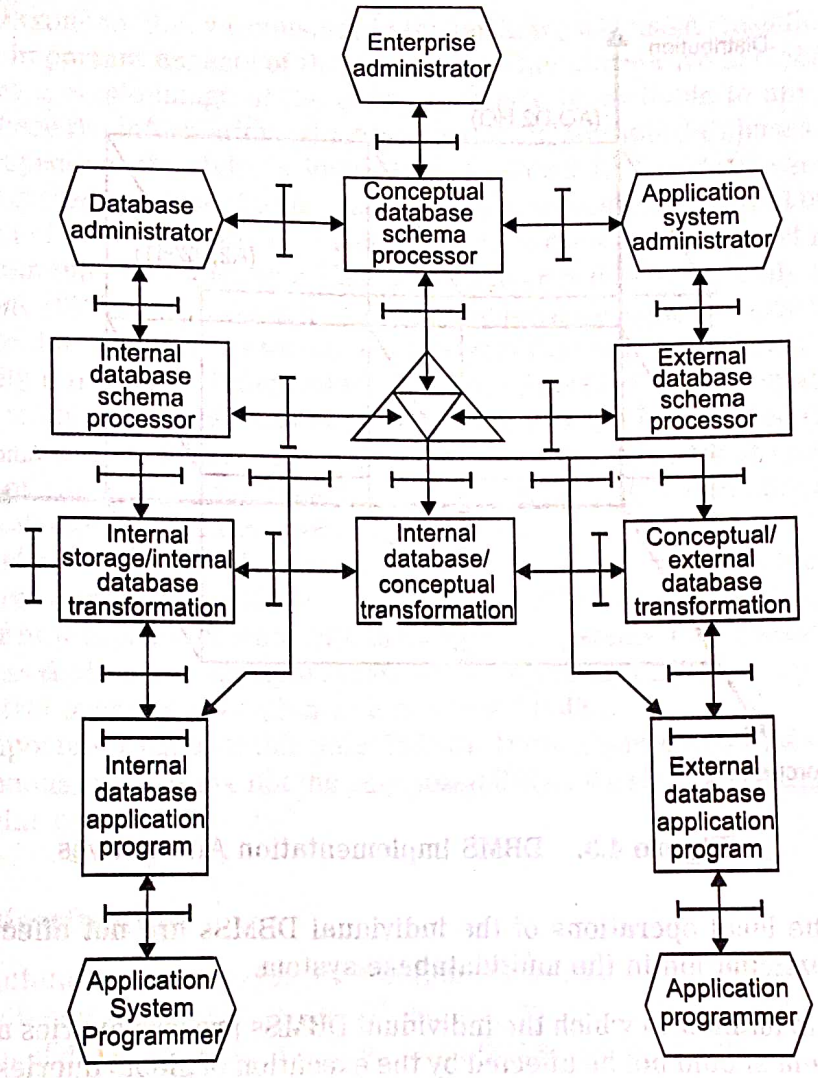

Figure 4.2. Partial Schematic of the ANSI/SPARC Architectural Model (Adapted from [Tsichritzis and Klug, 1978])

the systems as characterized with respect to (1) the autonomy of local systems, (2) their distribution, and (3) their heterogeneity.

4.2.1 Autonomy

Autonomy refers to the distribution of control, not of data. It indicates the degree to which individual DBMSs can operate independently. Autonomy is a function of a number of factors such as whether the component systems exchange information, whether they can independently execute transactions, and whether one is allowed to modify them. Requirements of an autonomous system have been specified in a variety of ways. For example, [Gligor and Popescu–Zeletin, 1986] lists these requirements as follows:

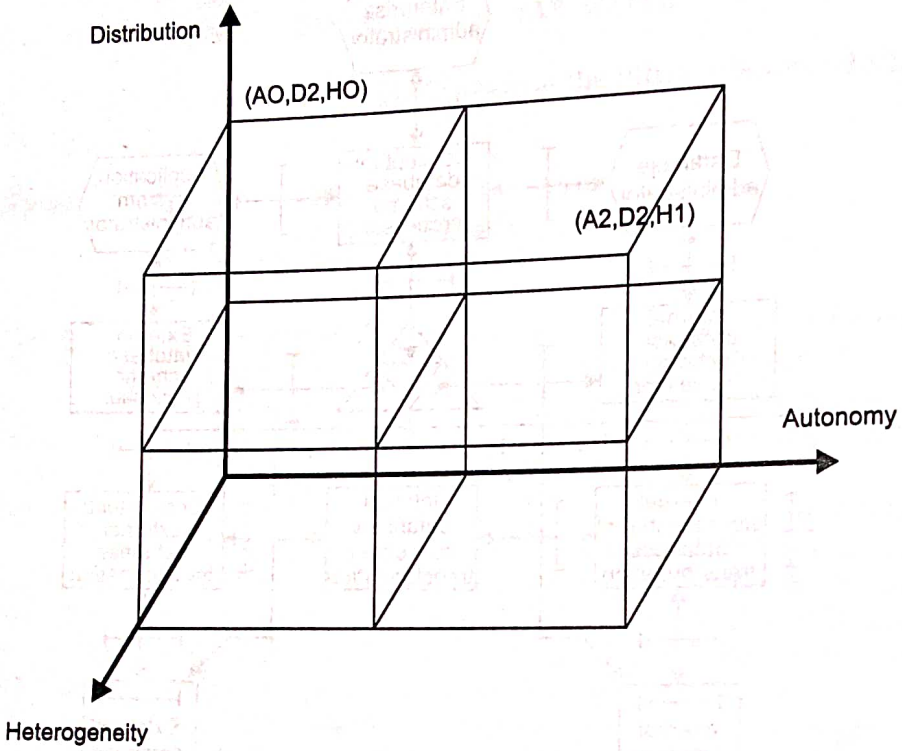

Figure 4.3. DBMS Implementation Alternatives

1. The local operations of the individual DBMSs are not affected by their participation in the multidatabase system.
2. The manner in which the individual DBMSs process queries and optimize them should not be affected by the execution of global queries that access multiple databases.
3. System consistency or operation should not be compromised when individual DBMSs join or leave the multidatabase confederation.

On the other hand, [Du and Elmagarmid, 1989] specifies the dimensions of autonomy as:

1. Design autonomy: Individual DBMSs are free to use the data models and transaction management techniques that they prefer.
2. Communication autonomy: Each of the individual DBMSs is free to make its own decision as to what type of information it wants to provide to the other DBMSs or to the software that controls their global execution.
3. Execution autonomy: Each DBMS can execute the transactions that are submitted to it in any way that it wants to.

In the taxonomy that we consider in the book, we will use a classification that covers the important aspects of these features. One alternative is *tight integration*, where a single-image of the entire database is available to any user who wants to share the information, which may reside in multiple databases. From the users' perspective, the data is logically centralized in one database. In these tightly-integrated systems, the data managers are implemented so that one of them is in control of the processing of each user request even if that request is serviced by more than one data manager. The data managers do not typically operate as independent DBMSs even though they usually have the functionality to do so.

Next we identify *semiautonomous* systems that consist of DBMSs that can (and usually do) operate independently, but have decided to participate in a federation to make their local data sharable. Each of these DBMSs determine what parts of their own database they will make accessible to users of other DBMSs. They are not fully autonomous systems because they need to be modified to enable them to exchange information with one another.

The last alternative that we consider is *total isolation*, where the individual systems are stand-alone DBMSs, which know neither of the existence of other DBMSs nor how to communicate with them. In such systems, the processing of user transactions that access multiple databases is especially difficult since there is no global control over the execution of individual DBMSs.

It is important to note at this point that the three alternatives that we consider for autonomous systems are not the only possibilities. We simply highlight the three most popular ones.

4.2.2 Distribution

Whereas autonomy refers to the distribution of control, the distribution dimension of the taxonomy deals with data. Of course, we are considering the physical distribution of data over multiple sites; as we discussed in Chapter 1, the user sees the data as one logical pool. There are a number of ways DBMSs have been distributed. We abstract these alternatives into two classes: *client/server* distribution and *peer-to-peer* distribution (or *full* distribution). Together with the non-distributed option, the taxonomy identifies three alternative architectures.

The client/server distribution, which has become quite popular in the last number of years, concentrates data management duties at servers while the clients focus on providing the application environment including the user interface. The communication duties are shared between the client machines and servers. Client/server DBMSs represent the first attempt at distributing functionality. There are a variety of ways of structuring them, each providing a different level of distribution. With respect to the framework, we abstract these differences and leave that discussion to Section 4.1, which we devote to client/server DBMS architectures. What is important at this point is that the sites on a network are distinguished as "clients" and "servers" and their functionality is different.

In *peer-to-peer systems*, there is no distinction of client machines versus servers. Each machine has full DBMS functionality and can communicate with other machines to execute queries and transactions. These systems, which are also

called *fully distributed*, are our main focus in this book, even though many of the techniques carry over to client/server systems as well.

4.2.3 Heterogeneity

Heterogeneity may occur in various forms in distributed systems, ranging from hardware heterogeneity and differences in networking protocols to variations in data managers. The important ones from the perspective of this book relate to data models, query languages, and transaction management protocols. Representing data with different modeling tools creates heterogeneity because of the inherent expressive powers and limitations of individual data models. Heterogeneity in query languages not only involves the use of completely different data access paradigms in different data models (set-at-a-time access in relational systems versus record-at-a-time access in network and hierarchical systems), but also covers differences in languages even when the individual systems use the same data model. Different query languages that use the same data model often select very different methods for expressing identical requests (e.g., DB2 uses SQL, while INGRES uses QUEL).[5]

4.2.4 Architectural Alternatives

Let us consider the architectural alternatives starting at the origin in Figure 4.3 and moving along the autonomy dimension. For identification, we use a notation based on the alternatives along the three dimensions. The dimensions are identified as A (autonomy), D (distribution) and H (heterogeneity). The alternatives along each dimension are identified by numbers as 0, 1 or 2. These numbers, of course, have different meanings along each of the dimensions. Along the autonomy dimension, 0 represents tight integration, 1 represents semiautonomous systems and 2 represents total isolation. Along distribution, 0 is for no distribution, 1 is for client/server systems, and 2 is for peer-to-peer distribution. Finally, along the heterogeneity dimension, 0 identifies homogeneous systems while 1 stands for heterogeneous systems. In Figure 4.3, we have identified two alternative architectures that are the focus of this book: (A0, D2, H0) which is a (peer-to-peer) distributed homogeneous DBMS and (A2, D2, H1) which represents a (peer-to-peer) distributed, heterogeneous multidatabase system. We should note that not all the architectural alternatives that are identified by this design space are meaningful. We will, nevertheless, discuss each and indicate the unrealistic architectures where appropriate.

(A0, D0, H0): The first class of systems are those which are logically integrated. Such systems can be given the generic name *composite systems* [Heimbigner and McLeod, 1985]. If there is no distribution or heterogeneity, the system is a set of multiple DBMSs that are logically integrated. There are not many examples of such systems, but they may be suitable for shared-everything multiprocessor systems.

[8]For completeness, note that INGRES also supports SQL.

(A0, D0, H1): If heterogeneity is introduced, one has multiple data managers that are heterogeneous but provide an integrated view to the user. In the past, some work was done in this class where systems were designed to provide integrated access to network, hierarchical, and relational databases residing on a single machine (see, e.g., [Dogac and Ozkarahan, 1980]).

(A0, D1, H0): The more interesting case is where the database is distributed even though an integrated view of the data is provided to users. This alternative represents client/server distribution that we mentioned earlier and will discuss further in Section 4.3.1.

(A0, D2, H0): This point in the design space represents a scenario where the same type of transparency is provided to the user in a fully distributed environment. There is no distinction among clients and servers, each site providing identical functionality. We discuss this design point in more detail in Section 4.3.2 and this alternative remains our main focus in this book.

Recall from Chapter 1 that the last two cases exactly match the definition of what we have called a *distributed DBMS* but without any heterogneity. The following cases will introduce various degrees of heterogeneity to the system.

(A1, D0, H0): The next point in the autonomy dimension are semiautonomous systems, which are commonly termed *federated DBMS* [Heimbigner and McLeod, 1985]. As specified before, the component systems in a federated environment have significant autonomy in their execution, but their participation in a federation indicate that they are willing to cooperate with others in executing user requests that access multiple databases. At this point of the design space, the component systems do not have to be distributed or heterogeneous. An example may be multiple installations (on the same machine) of an "open" DBMS. Here open means that the DBMS knows how to participate in a federation. In such a set-up, each DBMS is devoted to a particular function yet a layer of software on top of them provides the user the capability to access all of them in an integrated manner. This is not a very realistic design alternative, but it establishes the framework for the next two architectures.

(A1, D0, H1): These are systems that introduce heterogeneity as well as autonomy, what we might call a *heterogeneous federated DBMS*. Examples of these are easy to find in everyday use. Assume, for example, that there is a relational DBMS that manages structured data, an image DBMS that handles still images and a video server.[6] If we wish to provide an integrated view to the users, then it is necessary to "hide" the autonomy and heterogeneity of the component systems and establish a common interface. Systems of this type are the focus of Chapter 15.

[6] We are assuming, at the moment, that these systems have sacrificed some of their autonomy by agreeing to join a federation. What this sacrifice means is not, as we will discuss in Chapter 15.

(A1, D1, H1): Systems of this type introduce distribution by placing component systems on different machines. They may be referred to as *distributed, heterogeneous federated DBMS*. It is fair to state that the distribution aspects of these systems are less important than their autonomy and heterogeneity. Distribution introduces some new problems, but generally the techniques developed for homogeneous and non-autonomous distributed DBMSs (i.e., alternatives (A0, D1, H0) and (A0, D2, H0)) can be applied to deal with those issues.

(A2, D0, H0): If we move to full autonomy, we get what we call the class of *multidatabase system* (MDBS) architectures. The identifying characteristic of these systems is that the components have no concept of cooperation and they do not even know how to "talk to each other". Without heterogeneity or distribution, an MDBS is an interconnected collection of autonomous databases. A multidatabase management system (multi-DBMS) is the software that provides for the management of this collection of autonomous databases and transparent access to it. This is not a very realistic alternative either since it assumes no heterogeneity among component systems. This can happen in only two cases: either we have multiple installations of the same DBMS or we have a set of DBMSs with identical functionality and interface. Neither of these circumstances are likely to occur.

(A2, D0, H1): This case is realistic, maybe even more so than (A1, D0, H1), in that we always want to build applications which access data from multiple storage systems with different characteristics. Some of these storage systems may not even be DBMSs and they certainly have not been designed and developed with a view to interoperating with any other software. The example that we give for (A1, D0, H1) applies to this case as well if we assume that the component systems have no concept of entering a federation.

(A2, D1, H1) and (A2, D2, H1): We consider these two cases together simply because of the similarity of the problems. They both represent the case where component databases that make up the MDBS are distributed over a number of sites—we call this the *distributed MDBS*. As indicated above, the solutions to distribution issues for the two cases are similar and the general approach to dealing with interoperability does not differ too much. Perhaps the major difference is that in the case of client/server distribution (A2, D1, H1), most of the interoperability concerns are delegated to *middleware* systems resulting in a *three-layer architecture*.

The organization of a distributed MDBS as well as its management is quite different from that of a distributed DBMS. We discuss this issue in more detail in the upcoming sections. At this point it suffices to point out that the fundamental difference is one of the level of autonomy of the local data managers. Centralized or distributed multidatabase systems can be homogeneous or heterogeneous.

The fundamental point of the foregoing discussion is that the distribution of databases, their possible heterogeneity, and their autonomy are orthogonal issues. Since our concern in this book is on distributed systems, it is more important to note the orthogonality between autonomy and heterogeneity. Thus it is possible to have autonomous distributed databases that are not heterogeneous. In that sense, the more important issue is the autonomy of the databases rather than their heterogeneity. In other words, if the issues related to the design of a distributed multidatabase are resolved, introducing heterogeneity may not involve significant additional difficulty. This, of course, is true only from the perspective of database management; there may still be significant heterogeneity problems from the perspective of the operating system and the underlying hardware.

It is fair to claim that the fundamental issues related to multidatabase systems can be investigated without reference to their distribution. The additional considerations that distribution brings, in this case, are no different from those of logically integrated distributed database systems. Therefore, in this chapter we consider architectural models of logically integrated distributed DBMSs and multidatabase systems.

4.3 DISTRIBUTED DBMS ARCHITECTURE

In this section we consider, in detail, three of the system architectures from among the thirty that we identified in the previous section. The three are client/server systems (where we discount the heterogeneity and autonomy issues—i.e., (Ax, D1, Hy)), distributed databases, corresponding to (A0, D2, H0), and multidatabase systems, corresponding to (A2, Dx, Hy). These represent extremes that help focus the discussion on the most important issues.

4.3.1 Client/Server Systems

Client/server DBMSs entered the computing scene at the beginning of 1990's and have made a significant impact on both the DBMS technology and the way we do computing. The general idea is very simple and elegant: distinguish the functionality that needs to be provided and divide these functions into two classes: server functions and client functions. This provides a *two-level architecture* which makes it easier to manage the complexity of modern DBMSs and the complexity of distribution.

As with any highly popular term, client/server has been much abused and has come to mean different things. If one takes a process-centric view, then any process that requests the services of another process is its client and vice versa. In this sense, client/server computing is not new. However, it is important to note that "client/server computing" and "client/server DBMS," as it is used in its more modern context, do not refer to processes, but to actual machines. Thus, we focus on what software should run on the client machines and what software should run on the server machine.

Put this way, the issue is clearer and we can begin to study the differences in client and server functionality. The first thing to note is that the server does most of the data management work. This means that all of query processing and optimization, transaction management and storage management is done at the server. The client, in addition to the application and the user interface, has a *DBMS client* module that is responsible for managing the data that is cached to the client and (sometimes) managing the transaction locks that may have been cached as well. It is also possible to place consistency checking of user queries at the client side, but this is not common since it requires the replication of the system catalog at the client machines. Of course, there is operating system and communication software that runs on both the client and the server, but we only focus on the DBMS related functionality. This architecture, depicted in Figure 4.4, is quite common in relational systems where the communication between the clients and the server(s) is at the level of SQL statements. In other words, the client passes SQL queries to the server without trying to understand or optimize them. The server does most of the work and returns the result relation to the client.

There are a number of different types of client/server architecture. The simplest is the case where there is only one server which is accessed by multiple clients. We call this *multiple client-single server*. From a data management

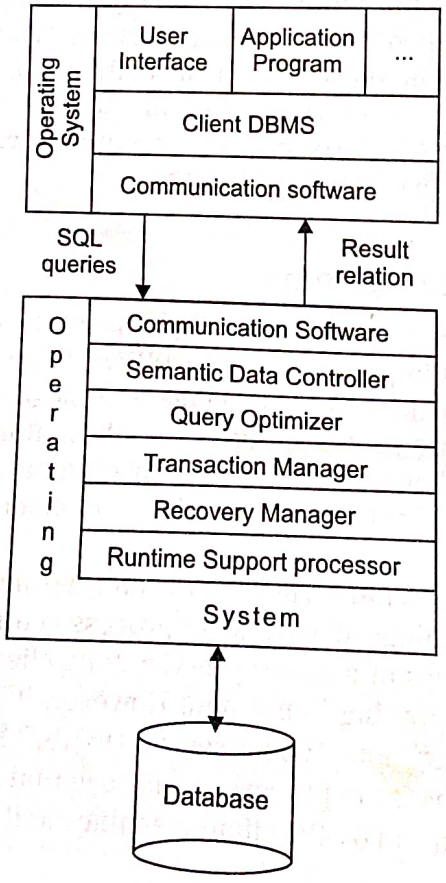

Figure 4.4. Client/Server Reference Architecture

perspective, this is not much different from centralized databases since the database is stored on only one machine (the server) which also hosts the software to manage it. However, there are some (important) differences from centralized systems in the way transactions are executed and caches are managed. We do not consider such issues at this point. A more sophisticated client/server architecture is one where there are multiple servers in the system (the so-called *multiple client-multiple server* approach). In this case, two alternative management strategies are possible: either each client manages its own connection to the appropriate server or each client knows of only its "home server" which then communicates with other servers as required. The former approach simplifies server code, but loads the client machines with additional responsibilities. This leads to what has been called "heavy client" systems. The latter approach, on the other hand, concentrates the data management functionality at the servers. Thus, the transparency of data access is provided at the server interface, leading to "light clients."

From a datalogical perspective, client/server DBMSs provide the same view of data as do peer-to-peer systems that we discuss next. That is, they give the user the appearance of a logically single database, while at the physical level data may be distributed. Thus the primary distinction between client/server systems and peer-to-peer ones is not in the level of transparency that is provided to the users and applications, but in the architectural paradigm that is used to realize this level of transparency.

4.3.2 Peer-to-Peer Distributed Systems

Let us start the description of the architecture by looking at the data organizational view. We first note that the physical data organization on each machine may be, and probably is, different. This means that there needs to be an individual internal schema definition at each site, which we call the *local internal schema* (LIS). The enterprise view of the data is described by the *global conceptual schema* (GCS), which is global because it describes the logical structure of the data at all the sites.

As we discussed briefly in Chapter 1, data in a distributed database is usually fragmented and replicated. To handle this phenomenon of fragmentation and replication, the logical organization of data at each site needs to be described. Therefore, there needs to be a third layer in the architecture, the *local conceptual schema* (LCS). In the architectural model we have chosen, then, the global conceptual schema is the union of the local conceptual schemas. Finally, user applications and user access to the database is supported by *external schemas* (ESs), defined as being above the global conceptual schema.

This architecture model, depicted in Figure 4.5, provides the levels of transparency discussed in Section 4.1. Data independence is supported since the model is an extension of ANSI/SPARC, which provides such independence naturally. Location and replication transparencies are supported by the definition of the local and global conceptual schemas and the mapping in between. Network transparency, on the other hand, is supported by the definition of the global conceptual schema. The user queries data irrespective of its location or of which local component of the distributed database system will service it. As mentioned before, the

distributed DBMS translates global queries into a group of local queries, which are executed by distributed DBMS components at different sites that communicate with one another.

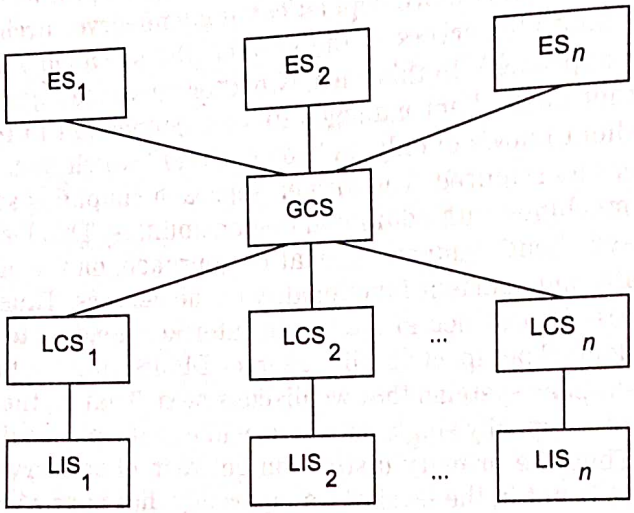

Figure 4.5. Distributed Database Reference Architecture

In terms of the detailed functional description of our model, the ANSI/SPARC model is extended by the addition of a *global directory/dictionary* (GD/D)[7] that permits the required global mappings. The local mappings are still performed by a *local directory/dictionary* (LD/D). Thus the local database management components are integrated by means of global DBMS functions (Figure 4.6).

As we can see in Figure 4.6, the local conceptual schemas are mappings of the global schema onto each site. Furthermore, such databases are typically designed in a top-down fashion, and therefore, all external view definitions are made globally. We have also depicted, in Figure 4.6, a local database administrator at each site. The existence of such a role may be controversial. However, remember that one of the primary motivations of distributed processing is the desire to have local control over the administration of data.

The detailed components of a distributed DBMS are shown in Figure 4.7. One component handles the interaction with users, and another deals with the storage. The first major component, which we call the *user processor*, consists of four elements:

1. The *user interface handler* is responsible for interpreting user commands as they come in, and formatting the result data as it is sent to the user.
2. The *semantic data controller* uses the integrity constraints and authorizations that are defined as part of the global conceptual schema to check

[7] In the remainder, we will simply refer to this as the *global directory*.

if the user query can be processed. This component, which is studied in detail in Chapter 6, is also responsible for authorization and other functions.

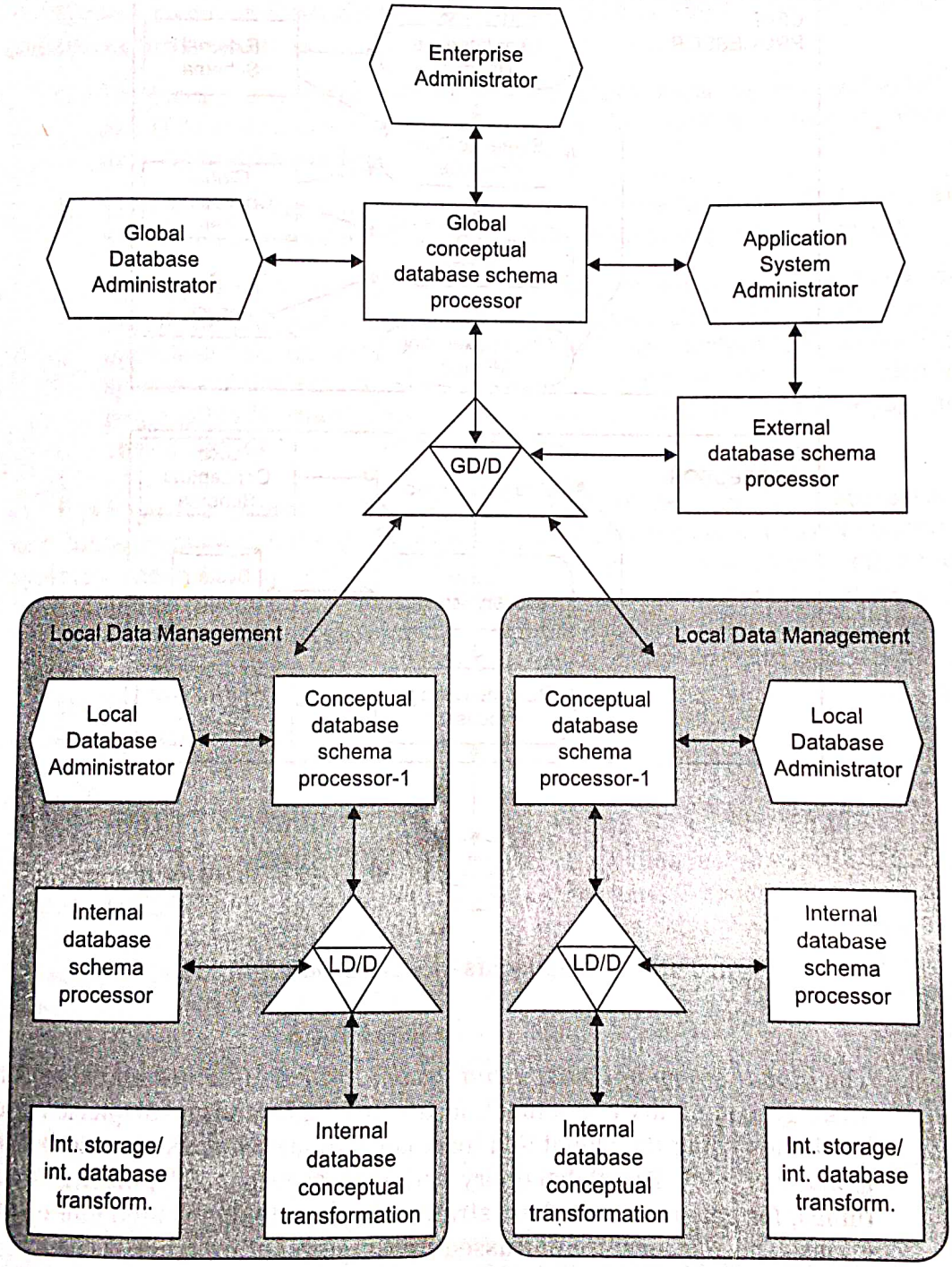

Figure 4.6. Functional Schematic of an Integrated Distributed DBMS

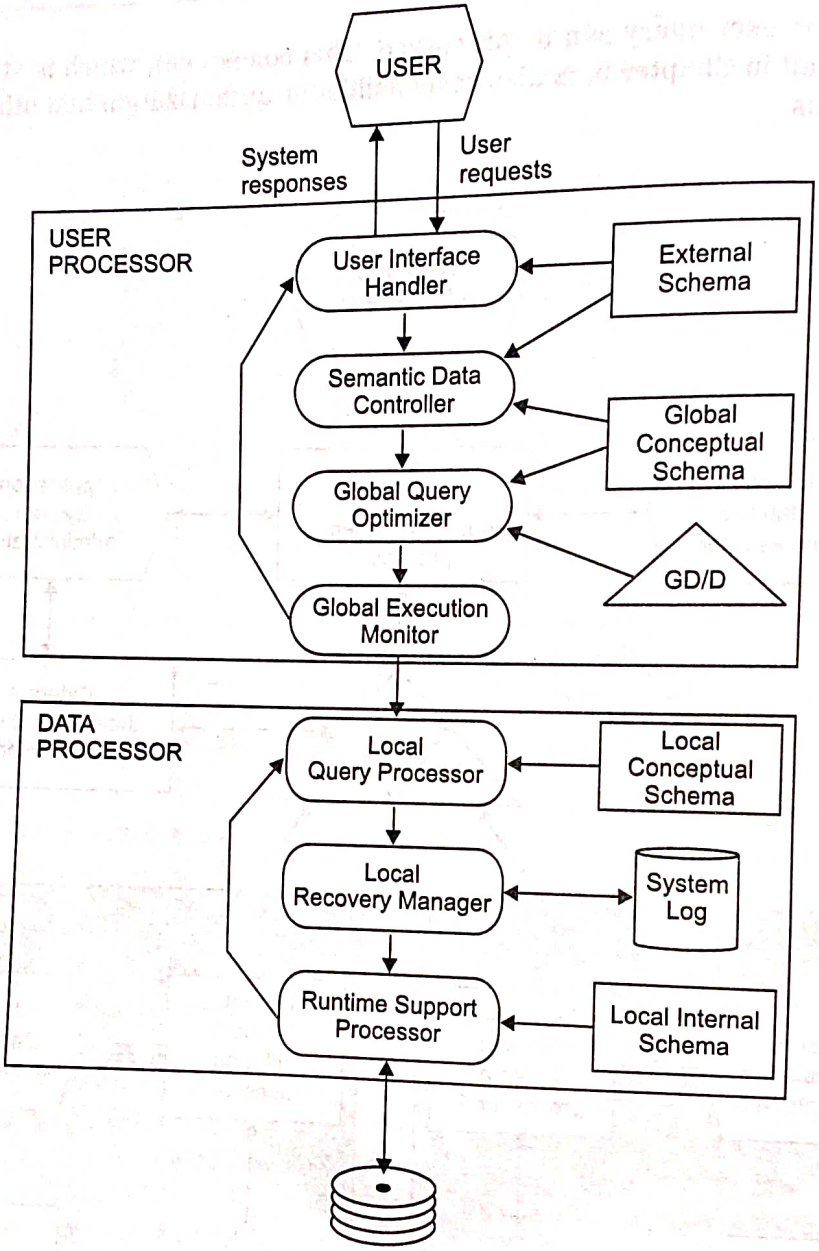

Figure 4.7. Components of a Distributed DBMS

3. The *global query optimizer and decomposer* determines an execution strategy to minimize a cost function, and translates the global queries into local ones using the global and local conceptual schemas as well as the global directory. The global query optimizer is responsible, among other things, for generating the best strategy to execute distributed join operations. These issues are discussed in Chapters 7 through 9.

4. The *distributed execution monitor* coordinates the distributed execution of the user request. The execution monitor is also called the *distributed transaction manager*. In executing queries in a distributed fashion, the execution monitors at various sites may, and usually do, communicate with one another.

The second major component of a distributed DBMS is the *data processor* and consists of three elements:

1. The *local query optimizer*, which actually acts as the *access path selector*, is responsible for choosing the best access path[8] to access any data item (touched upon briefly in Chapter 9).
2. The *local recovery manager* is responsible for making sure that the local database remains consistent even when failures occur (Chapter 12).
3. The *run-time support processor* physically accesses the database according to the physical commands in the schedule generated by the query optimizer. The run-time support processor is the interface to the operating system and contains the *database buffer* (or *cache*) *manager*, which is responsible for maintaining the main memory buffers and managing the data accesses.

It is important to note, at this point, that our use of the terms "user processor" and "data processor" does not imply a functional division similar to client/server systems. These divisions are merely organizational and there is no suggestion that they should be placed on different machines. In peer-to-peer systems, one expects to find both the user processor modules and the data processor modules on each machine. However, there have been suggestions to separate "query-only sites" in a system from full-functionality ones. In this case, the former sites would only need to have the user processor.

4.3.3 MDBS Architecture

The differences in the level of autonomy between the distributed multi-DBMSs and distributed DBMSs are also reflected in their architectural models. The fundamental difference relates to the definition of the global conceptual schema. In the case of logically integrated distributed DBMSs, the global conceptual schema defines the conceptual view of the *entire* database, while in the case of distributed multi-DBMSs, it represents only the collection of *some* of the local databases that each local DBMS wants to share. Thus the definition of a *global database* is different in MDBSs than in distributed DBMSs. In the latter, the global database is equal to the union of local databases, whereas in the former it is only a subset of the same union. There are even arguments as to whether the global conceptual schema

[8] The term *access path* refers to the data structures and the algorithms that are used to access the data. A typical access path, for example, is an index on one or more attributes of a relation.

should even exist in multidatabase systems. This question forms the basis of our architectural discussions in this section.

Models Using a Global Conceptual Schema

In an MDBS, the GCS is defined by integrating either the external schemas of local autonomous databases or parts of their local conceptual schemas (Figure 4.8). Furthermore, users of a local DBMS define their own views on the local database and do not need to change their applications if they do not want to access data from another database. This is again an issue of autonomy.

Designing the global conceptual schema in multidatabase systems involves the integration of either the local conceptual schemas or the local external schemas. A major difference between the design of the GCS in multi-DBMSs and in logically integrated distributed DBMSs is that in the former the mapping is from local conceptual schemas to a global schema. In the latter, however, mapping is in the

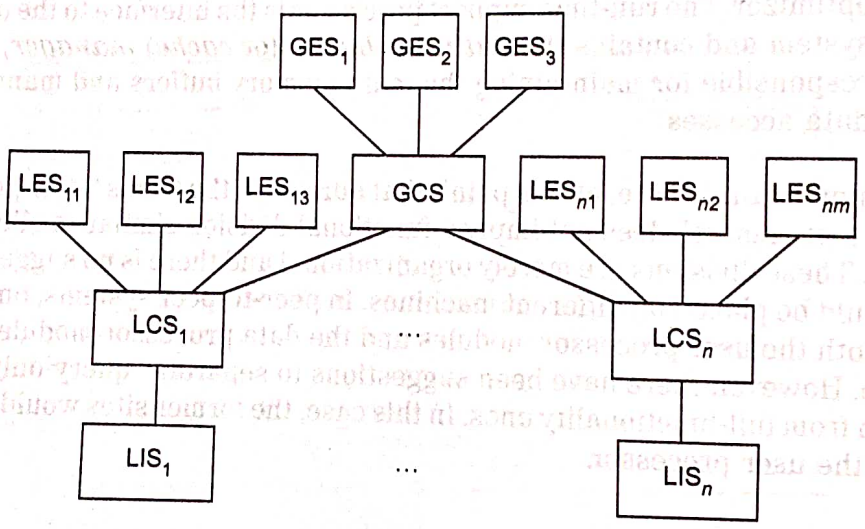

Figure 4.8. MDBS Architecture with a GCS

reverse direction. As we discuss in Chapter 5, this is because the design in the former is usually a bottom-up process, whereas in the latter it is usually a top-down procedure. Furthermore, if heterogeneity exists in the multidatabase system, a canonical data model has to be found to define the GCS.

Once the GCS has been designed, views over the global schema can be defined for users who require global access. It is not necessary for the GES and GCS to be defined using the same data model and language; whether they do or not determines whether the system is homogeneous or heterogeneous.

If heterogeneity exists in the system, then two implementation alternatives exist: unilingual and multilingual. A *unilingual* multi-DBMS requires the users to utilize possibly different data models and languages when both a local database and the global database are accessed. The identifying characteristic of unilingual

systems is that any application that accesses data from multiple databases must do so by means of an external view that is defined on the global conceptual schema. This means that the user of the global database is effectively a different user than those who access only a local database, utilizing a different data model and a different data language. Thus, one application may have a *local external schema* (LES) defined on the local conceptual schema as well as a *global external schema* (GES) defined on the global conceptual schema. The different external view definitions may require the use of different access languages. Figure 4.8 actually depicts the datalogical model of a unilingual database system that integrates the local conceptual schemas (or parts of them) into a global conceptual schema. Examples of such an architecture are the MULTIBASE system ([Landers and Rosenberg, 1982], [Smith et al., 1981]) Mermaid [Templeton et al., 1987] and DDTS [Dwyer et al., 1986].

An alternative is *multilingual* architecture, where the basic philosophy is to permit each user to access the global database (i.e., data from other databases) by means of an external schema, defined using the language of the user's local DBMS. The GCS definition is quite similar in the multilingual architecture and the unilingual approach, the major difference being the definition of the external schemas, which are described in the language of the external schemas of the local database. Assuming that the definition is purely local, a query issued according to a particular schema is handled exactly as any query in the centralized DBMSs. Queries against the global database are made using the language of the local DBMS, but they generally require some processing to be mapped to the global conceptual schema.

The multilingual approach obviously makes querying the databases easier from the user's perspective. However, it is more complicated because we must deal with translation of queries at run time. The multilingual approach is used in Sirius–Delta [Ferrier and Stangret, 1982] and in the HD–DBMS project [Cardenas, 1987].

Models Without a Global Conceptual Schema

The existence of a global conceptual schema in a multidatabase system is a controversial issue. There are researchers who even define a multidatabase management system as one that manages "several databases without a global schema" [Litwin, 1988]. It is argued that the absence of a GCS is a significant advantage of multidatabase systems over distributed database systems. One prototype system that has used this architectural model is the MRDSM project ([Litwin and Abdellatif, 1987], [Litwin and Abdellatif, 1986]). The architecture depicted in Figure 4.9, identifies two layers: the local system layer and the multidatabase layer on top of it. The local system layer consists of a number of DBMSs, which present to the multidatabase layer the part of their local database they are willing to share with users of other databases. This shared data is presented either as the actual local conceptual schema or as a local external schema definition. (Figure 4.9 shows this layer as a collection of local conceptual schemas.) If heterogeneity is involved, each of these schemas, LCS_i, may use a different data model.

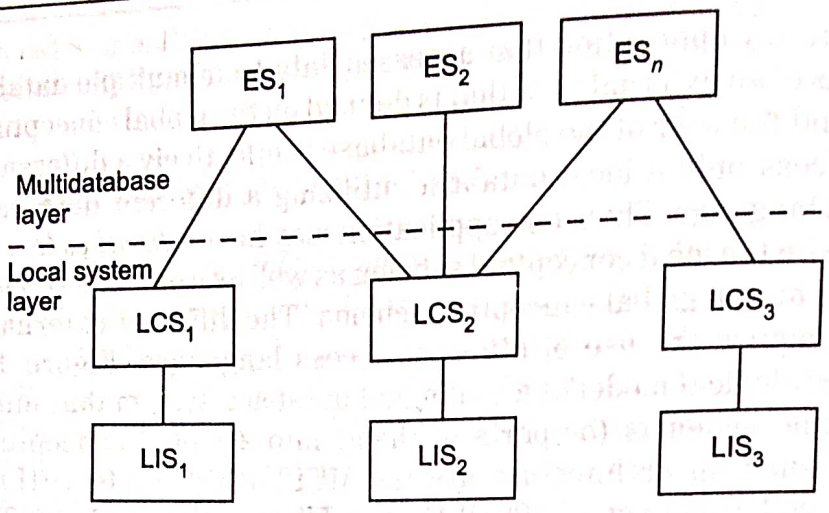

Figure 4.9. MDBS Architecture Without a GCS (From [Litwin, 1988])

Above this layer, external views are constructed where each view may be defined on one local conceptual schema or on multiple conceptual schemas. Thus the responsibility of providing access to multiple (and may be heterogeneous) databases is delegated to the mapping between the external schemas and the local conceptual schemas. This is fundamentally different from architectural models that use a global conceptual schema, where this responsibility is taken over by the mapping between the global conceptual schema and the local ones. This shift in responsibility has a practical consequence. Access to multiple databases is provided by means of a powerful language in which user applications are written [Siegel, 1987].

Federated database architectures, which we discussed briefly, do not use a global conceptual schema either. In the specific system described in [Heimbigner and McLeod, 1985], each local DBMS defines an *export schema,* which describes the data it is willing to share with others. In the terminology that we have been using, the global database is the union of all the export schemas. Each, application that accesses, the global database does so by the definition of an *import schema,* which is simply a global external view.

The component-based architectural model of a multi-DBMS is significantly different from a distributed DBMS. The fundamental difference is the existence of full-fledged DBMSs, each of which manages a different database. The MDBS provides a layer of software that runs on top of these individual DBMSs and provides users with the facilities of accessing various databases (Figure 4.10). Depending on the existence (or lack) of the global conceptual schema or the existence of heterogeneity (or lack of it), the contents of this layer of software would change significantly. Note that Figure 4.10 represents a nondistributed multi-DBMS. If the system is distributed, we would need to replicate the multidatabase layer to each site where there is a local DBMS that participates in the system. Also note that as far as the individual DBMSs are concerned, the MDBS layer is simply another application that submits requests and receives answers.

The domain of federated database and multidatabase systems is complicated by the proliferation of terminology and different architectural models. We bring

some order to the field in this section, but the architectural approaches that we summarize are not unique. In Chapter 15, we discuss another architectural specification that is widely cited, due to [Sheth and Larson, 1990].

REVIEW QUESTIONS

4.1 What do you mean by DBMS standardization? Give an example to explain it.
4.2 Give a brief account of architectural models for distributed DBMS.
4.3 Explain client/server reference architecture.
4.4 Explain functional schematic of an integrated distributed DBMS.
4.5 What are the components of a distributed DBMS?
4.6 Explain MDBS architecture without a GCS.