1

INTRODUCTION

> **Distributed: (adjective)** of, relating to, or being a computer network in which at least some of the processing is done by the individual workstations and information is shared by and often stored at the workstations.
>
> —Merriam-Webster's 11th Collegiate Dictionary

> **Database (noun)** a [sic] usually large collection of data organized especially for rapid search and retrieval (as by a computer).
>
> —Merriam-Webster's 11th Collegiate Dictionary

Informally speaking, a **database (DB)** is simply a collection of data stored on a computer, and the term **distributed** simply means that more than one computer might cooperate in order to perform some task. Most people working with distributed databases would accept both of the preceding definitions without any reservations or complaints. Unfortunately, achieving this same level of consensus is not as easy for any of the other concepts involved with **distributed databases (DDBs)**. A DDB is not simply "more than one computer cooperating to store a collection of data"—this definition would include situations that are not really distributed databases, such as any machine that contains a DB and also mounts a remote file system from another machine. Similarly, this would be a bad definition because it would not apply to any scenario where we deploy a DDB on a single computer. Even when a DDB is deployed using only one computer, it remains a DDB because it is still **possible to deploy it across multiple computers**. Often, in order to discuss a particular approach for implementing a DB, we need to use more restrictive and specific definitions. This means that the same terms might have conflicting definitions when we consider more than one DB implementation alternative. This can be very confusing when researching DBs in general and especially confusing when focusing on DDBs. Therefore, in this chapter, we

Distributed Database Management Systems by Saeed K. Rahimi and Frank S. Haug

will present some definitions and archetypical examples along with a new taxonomy. We hope that these will help to minimize the confusion and make it easier to discuss multiple implementation alternatives throughout the rest of the book.

1.1 DATABASE CONCEPTS

Whenever we use the term "DB" in this book, we are always contemplating a collection of **persistent** data. This means that we "save" the data to some form of **secondary storage** (the data usually written to some form of hard disk). As long as we shut things down in an orderly fashion (following the correct procedures as opposed to experiencing a power failure or hardware failure), all the data written to secondary storage should still exist when the system comes back online. We can usually think of the data in a DB as being stored in one or more files, possibly spanning several partitions, or even several hard disk drives—even if the data is actually being stored in something more sophisticated than a simple file.

1.1.1 Data Models

Every DB captures data in two interdependent respects; it captures both the **data structure** and the **data content**. The term "data content" refers to the values actually stored in the DB, and usually this is what we are referring to when we simply say "data." The term "data structure" refers to all the necessary details that describe how the data is stored. This includes things like the format, length, location details for the data, and further details that identify how the data's internal parts and pieces are interconnected. When we want to talk about the structure of data, we usually refer to it as the **data model (DM)** (also called the **DB's schema**, or simply the **schema**). Often, we will use a special language or programmatic facility to create and modify the DM. When describing this language or facility, authors sometimes refer to the facility or language as "the data model" as well, but if we want to be more precise, this is actually the **data modeling language (ML)**—even when there is no textual language. The DM captures many details about the data being stored, but the DM does not include the actual data content. We call all of these details in the DM **metadata**, which is informally defined as "data about data" or "everything about data except the content itself." We will revisit data models and data modeling languages in Chapters 10 and 11.

1.1.2 Database Operations

Usually, we want to perform several different kinds of operations on DBs. Every DB must at least support the ability to "create" new data content (store new data values in the DB) and the ability to retrieve existing data content. After all, if we could not create new data, then the DB would always be empty! Similarly, if we could not retrieve the data, then the data would serve no purpose. However, these operations do not need to support the same kind of interface; for example, perhaps the data creation facility runs as a batch process but the retrieval facility might support interactive requests from a program or user. We usually expect newer DB software to support much more sophisticated operations than minimum requirements dictate. In particular, we usually

want the ability to update and delete existing data content. We call this set of operations **CRUD** (which stands for "create, retrieve, update, and delete"). Most modern DBs also support similar operations involving the data structures and their constituent parts. Even when the DBs support these additional "schema CRUD" operations, complicated restrictions that are dependent on the ML and sometimes dependent on very idiosyncratic deployment details can prevent some schema operations from succeeding.

Some DBs support operations that are even more powerful than schema and data CRUD operations. For example, many DBs support the concept of a **query**, which we will define as "a request to retrieve a collection of data that can potentially use complex criteria to broaden or limit the collection of data involved." Likewise, many DBs support the concept of a **command**, which we will define as "a request to create new data, to update existing data, or to delete existing data—potentially using complex criteria similar to a query." Most modern DBs that support both queries and commands even allow us to use separate queries (called **subqueries**) to specify the complex criteria for these operations.

Any DB that supports CRUD operations must consider **concurrent access** and **conflicting operations**. Anytime two or more requests (any combination of queries and commands) attempt to access overlapping collections of data, we have concurrent access. If all of the operations are only retrieving data (no creation, update, or deletion), then the DB can implement the correct behavior without needing any sophisticated logic. If any one of the operations needs to perform a write (create, update, or delete), then we have conflicting operations on overlapping data. Whenever this happens, there are potential problems—if the DB allows all of the operations to execute, then the execution order might potentially change the results seen by the programs or users making the requests. In Chapters 5, 6, and 8, we will discuss the techniques that a DB might use to control these situations.

1.1.3 Database Management

When DBs are used to capture large amounts of data content, or complex data structures, the potential for errors becomes an important concern—especially when the size and complexity make it difficult for human verification. In order to address these potential errors and other issues (like the conflicting operation scenario that we mentioned earlier), we need to use some specialized software. The DB vendor can deploy this specialized software as a library, as a separate program, or as a collection of separate programs and libraries. Regardless of the deployment, we call this specialized software a **database management system (DBMS)**. Vendors usually deploy a DBMS as a collection of separate programs and libraries.

1.1.4 DB Clients, Servers, and Environments

There is no real standard definition for a DBMS, but when a DBMS is deployed using one or more programs, this collection of programs is usually referred to as the **DB-Server**. Any application program that needs to connect to a DB is usually referred to as the **DB-Client**. Some authors consider the DB-Server and the DBMS to be equivalent—if there is no DB-Server, then there is no DBMS; so the terms **DBMS-Server** and **DBMS-Client** are also very common. However, even when there is no DB-Server, the application using the DB is still usually called the DB-Client. Different

DBMS implementations have different restrictions. For example, some DBMSs can manage more than one DB, while other implementations require a separate DBMS for each DB.

Because of these differences (and many more that we will not discuss here), it is sometimes difficult to compare different implementations and deployments. Simply using the term "DBMS" can suggest certain features or restrictions in the mind of the reader that the author did not intend. For example, we expect most modern DBMSs to provide certain facilities, such as some mechanism for defining and enforcing integrity constraints—but these facilities are not necessarily required for all situations. If we were to use the term "DBMS" in one of these situations where these "expected" facilities were not required, the reader might incorrectly assume that the "extra" facilities (or restrictions) were a required part of the discussion. Therefore, we introduce a new term, **database environment (DBE)**, which simply means one or more DBs along with any software providing at least the minimum set of required data operations and management facilities. In other words, a DBE focuses on the DB and the desired functionality—it can include a DBMS if that is part of the deployment, but does not have to include a DBMS as long as the necessary functionality is present. Similarly, the term DBE can be applied to DBs deployed on a single host, as well as DBs deployed over a distributed set of machines. By using this new term, we can ignore the architectural and deployment details when they are not relevant. While this might seem unnecessary, it will prevent the awkward phrasing we would have to use otherwise. (If you prefer, you can substitute a phrase like "one or more DB instances with the optional DBMS applications, libraries, or services needed to implement the expected data operations and management facilities required for this context" whenever you see the term "DBE.") There are times when we will explicitly use the term DBMS; in those instances, we are emphasizing the use of a traditional DBMS rather than some other facility with more or less capabilities or limitations. For example, we would use the term DBMS when we want to imply that an actual DBMS product such as Oracle, DB2, and so on is being used. If we use the term DBE, we could still be referring to one of these products, but we could also be referring to any other combination of software with greater, lesser, or equal levels of functionality. As we shall see, the term "DBE" can even refer to a larger system containing several other DBs, DBMSs, and DBEs within it!

1.2 DBE ARCHITECTURAL CONCEPTS

When considering the architecture of a complicated system, such as a DBE, there are several different ways we can view the details. In this section, we will provide a very brief and high-level view useful for discussing the archetypical DBE architectures used later in the chapter and later in the book. We will revisit architectural concerns in Chapters 12 to 15. For our purposes here, we will merely consider **services, components, subsystems**, and **sites**.

1.2.1 Services

Regardless of the deployment details, we can create logical collections of related functionality called **services**. For example, we mentioned earlier that many DBs support

queries; we can call the logical grouping of the software that implements this functionality the **query service**. We can define services like this for both publicly visible functions (such as query) and internal functions (such as query optimization). Services are merely logical collections, which means that they do not necessarily have corresponding structure duplicated in the actual implementation or deployment details. We call any piece of software that uses a service a **service consumer**, while any piece of software implementing the service is called a **service provider**. Implicitly, each service has at least one **interface** (similar to a contractual agreement that defines the inputs, outputs, and protocols used by the service consumers and providers). These interfaces can be very abstract (merely specifying an order of steps to be taken) or they can describe very tangible details such as data types or even textual syntax. Most of these interface details are usually only present in lower-level design diagrams—not the high-level architectural or deployment diagrams.

The same piece of software can be both a service consumer and a service provider and can even consume or provide several different services using many different interfaces—but it is usually better to limit the number of services involved for an individual piece of code. Although we can talk about the services as part of the overall architecture or deployment (like interfaces), we usually do not see them directly represented in architectural or deployment diagrams. Instead, we usually see the **components** and **subsystems** (which we will discuss further in the next section) implementing the services in these diagrams. We will discuss services further in Chapter 14.

1.2.2 Components and Subsystems

For our purposes, a **component** is simply a deployable bundle that provides a reasonably cohesive set of functionality, and a **subsystem** is a collection of one or more components that work together toward a common goal. Whenever we want to use the two terms interchangeably, we will use the term **COS (component or subsystem)**. Unlike a service, which is merely a logical grouping, a COS is a physical grouping, which means that it does have a corresponding structure in the implementation. Frequently, we name these COSs after the primary service that they **provide**. There can be multiple instances of the same COS deployed within the system. These instances are often referred to as **servers**, although we can also use other terminology. For example, we might see a Query COS defined in the architecture, and several Query Servers (instances of the Query COS) deployed within the environment. Alternatively, we might refer to these COSs or their deployed instances as Query Managers, Query Processors, Query Controllers, or even some other name. Different instances of the same COS can have different implementation and configuration details, as long as the instances still provide all the necessary services using the correct protocols and interfaces. We usually represent a COS in an architectural diagram as a box or oval with its name written inside. Deployment diagrams show each COS instance similarly, but usually the instance name includes some additional hint (such as a number, abbreviation, or other deployment detail) to help differentiate the instances from each other.

1.2.3 Sites

The term **site** represents a logical location in an architectural diagram or a deployment diagram—typically, this is a real, physical machine, but that is not necessarily true.

For example, we might have a deployment using two sites (named Site-A and Site-B). This means that those two sites could be any two machines as long as all of the necessary requirements are satisfied for the machines and their network connections. In certain circumstances, we could also deploy all of the subsystems located at Site-A and Site-B on the same machine. Remember, a DDB deployed on a single machine is still a DDB. In other words, as long as the deployment plan does not explicitly forbid deploying all the COS instances for that DDB on a single machine, we can deploy them this way and still consider it a DDB. Architectural and deployment diagrams depict sites as container objects (usually with the site name and the deployed COS instances included inside them) when there is more than one site involved; otherwise it is assumed that everything in the diagram is located in a single site, which may or may not be named in the diagram.

1.3 ARCHETYPICAL DBE ARCHITECTURES

As we have already discussed, when considering a DBE, there are some bare minimum requirements that need to be present—namely, the ability to add new data content and retrieve existing content. Most real-world DBEs provide more than just this minimal level of functionality. In particular, the update and delete operations for data are usually provided. We might see some more sophisticated facilities such as the query service and other services supporting schema operations. In this section, we will briefly consider the typical services we would expect to see in a DBE. Then, we will consider some archetypical DBE architectures.

1.3.1 Required Services

Figure 1.1 shows a simplistic architectural diagram for a minimal DBE. The architecture shown is somewhat unrealistic. In this architecture, there is a separate subsystem for each service discussed in this section, and each subsystem is named the same as the service that it provides. These subsystems are contained within a larger subsystem,

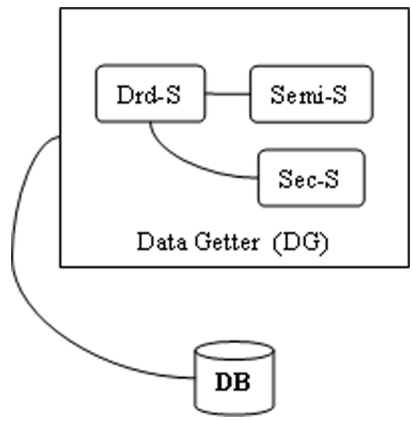

Figure 1.1 DBE architectural diagram emphasizing the minimum required services.

which we call the **Data Getter (DG)**. This DBE provides (at least) three services—they are named Drd-S, Sec-S, and Semi-S. When reading this diagram, we should recall that it is a DBE and, therefore, the services shown should be considered vitally important, or at least expected—but there might be additional services provided by the environment that are not shown here. For example, this diagram does not show any service providers for query or command operations but that does **not necessarily** mean we cannot use this diagram to discuss a DBE with those facilities—instead, it merely means that any DBE without those unmentioned facilities is still **potentially** represented by this diagram.

Whenever we use an architectural or deployment diagram for a DBE, we are usually highlighting some requirement or feature of the environment within a specific context; in this case, we are merely showing what a "bare minimum" DBE must provide, and the four services shown here satisfy those minimum requirements. Every DBE must include a service providing the ability to retrieve data from the DB. We will call this the **Data Read Service (Drd-S)**. Since most DBEs also have at least a basic level of privacy or confidentiality, there should always be some form of **Security Service (Sec-S)**. In an effort to be inclusive, we can consider DBEs with "no security" to be implementing a Sec-S that "permits everyone to do everything." Any real-world DBE should have a Sec-S providing both authentication and authorization, which we will discuss further in Chapter 9. The Drd-S uses the Sec-S to ensure that private data remains unseen by those without proper permissions. There is usually another service providing at least some minimal level of integrity checking or enforcement. This other service is responsible for preventing semantic errors (e.g., data content representing a salary must be greater than zero, otherwise it is a semantic error). Similar to the Sec-S, this service can be less than perfect, or perhaps even implemented using an "allow all modifications" policy if explicit constraints are not defined. We call this service the **Semantic Integrity Service (Semi-S)**, and we will discuss it further in Chapter 3. This service can be used by several services, including the Drd-S, which can use it to provide default values for the content it retrieves (among other possibilities).

1.3.2 Basic Services

In Section 1.1.2, we said that every DB must provide **some mechanism** for populating data (otherwise the DB would always be empty), but we also said that each DB might support different interfaces for the mechanisms they use. Therefore, the ability to write data is **not always** implemented as a service, or in the very least, it is not always implemented in a way that we can incorporate into our DBE architecture. For example, if the only data population mechanism provided by a particular DBE was a program that only supported interactive user input (text screens or graphical dialog boxes) then we could not use that program as a service provider in any meaningful sense. However, if the DBE does provide a "program-friendly" mechanism to write data, we can call this the **Data Write Service (Dwr-S)**. Although it is not a required service, and it is not present in all DBEs, it is typically present in most traditional DBMS products, and in many other DBEs we will consider. If there is a Dwr-S, then it uses the Sec-S (to prevent unauthorized data content additions, modifications, and removals) and the Semi-S (to prevent causing any semantic errors when it adds, modifies, or removes any data content). Once again, unless we specify the requirements more explicitly, it is possible for the Sec-S and Semi-S in a particular DBE to provide varying degrees of

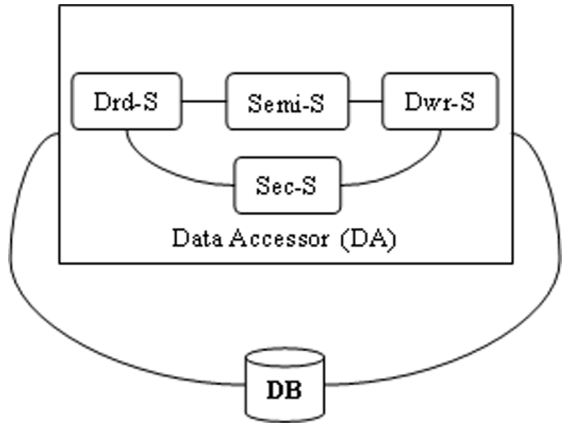

Figure 1.2 DBE architectural diagram emphasizing the basic services.

functionality for these services. Figure 1.2 shows the architectural diagram for a DBE providing the basic services we just discussed. Here, we show all the services used to access the data for read or write operations as a single subsystem called the **Data Accessor (DA)**. We could also have shown the Data Getter subsystem in place of the Drd-S. However, we did not include it here because the Semi-S and Sec-S are used by both the read and the write operations. Similarly, we could consider the combination of the Dwr-S, Semi-S, and Sec-S to be a "Data Setter" subsystem, but these details do not usually add much value to our diagram. In other words, the DA **always** implicitly includes a DG as part of it and the services shown in Figure 1.2.

1.3.3 Expected Services

Every DBE must supply the functionality contained in the DG, and many DBEs will provide the functionality contained in the DA, but often we expect a DBE to be more powerful than these minimal or basic scenarios. In particular, we mentioned the query service earlier, and here we will call it the **Query Request Service (Qreq-S)**. Most modern DBMSs should provide this as well as some form of **Query Optimization Service (Qopt-S)**, but neither of these services is a requirement for all DBEs. Typically, the Qreq-S forms a plan for a query and then passes the plan on to the Qopt-S. The Qopt-S optimizes the plan and then uses the Drd-S to retrieve the data matching the query criteria. We will discuss the Qreq-S and Qopt-S further in Chapter 4. We also mentioned that some DBEs have the ability to execute commands (create, update, and delete operations with potentially complex criteria). Therefore, in most DBEs providing DA operations, we would also expect to see an **Execution Service (Exec-S)** and **Execution Optimization Service (Eopt-S)** to encapsulate these command operations. Again, these are present in most DBMSs, but not necessarily present in all DBEs. Chapter 3 will explore these services further. Often, there is a "nonprogrammatic" interface provided to users. In particular, many relational DBMSs support a special language (called the SQL) and provide batch and/or interactive facilities that users can employ to pass queries and commands to the DB. We will call the service providing this function the **User Interface Service (UI-S)**. This service is not always present in

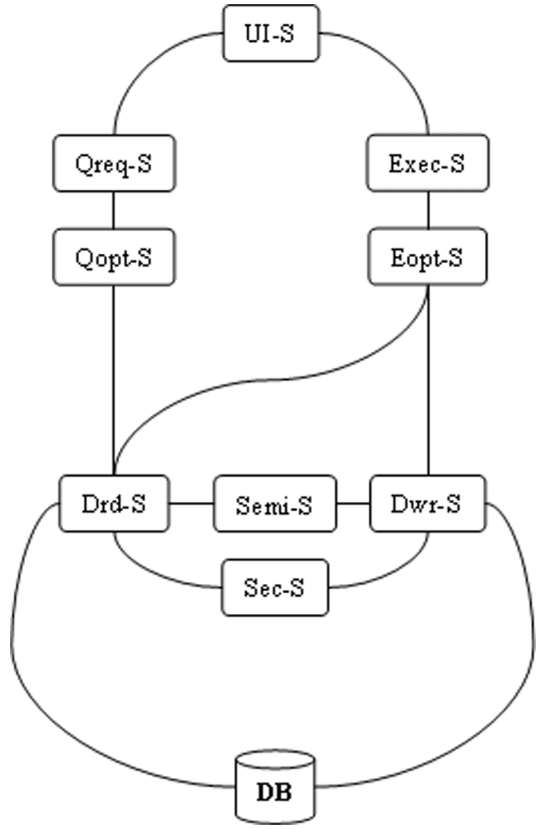

Figure 1.3 DBE architectural diagram emphasizing the expected services.

a DBE and is usually implemented differently, including different syntax, features, and restrictions for the queries and commands. However, we would expect most modern DBMSs (including nonrelational ones) to provide some sort of UI-S. Figure 1.3 shows an example of a typical DBE providing these expected services.

1.3.4 Expected Subsystems

Figure 1.4 shows a reasonably realistic DBE set of subsystems for the architecture we looked at in Figure 1.3; it contains all the same services, but we have bundled the services into four subsystems: the **application processor (AP)**, the **query processor (QP)**, the **command processor (CP)**, and the **data accessor (DA)**. Two of the subsystems (QP and CP) are contained within one of the others (AP), while the other two subsystems (AP and DA) are shown as independent packages. Each component has been allocated to one of the subsystems, and the communication links shown only connect subsystems rather than the components inside them. Although the communication links are not quite as detailed, there is no real loss of information when we do this in a diagram. We have placed the Qreq-S and Qopt-S inside the QP. Similarly, we have placed the Exec-S and Eopt-S inside the CP. The AP subsystem contains the combination of

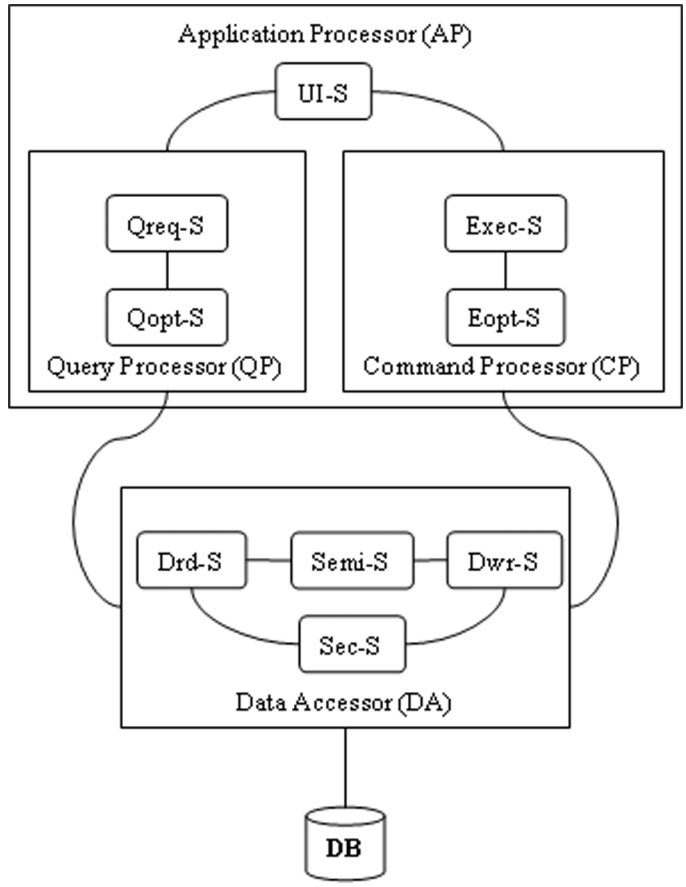

Figure 1.4 Typical subsystems of simple DBE with the expected services.

the UI-S, QP, and CP subsystems. All the remaining service components have been allocated to the DA, which we discussed in Section 1.3.2.

1.3.5 Typical DBMS Services

There can be many other services and subsystems in a DBE, but often these additional services and subsystems are highly dependent on other details, specific to the particular DBE being considered. This is especially true when the particular DBE being focused upon is a DBMS. For example, in a DBE with a Dwr-S, we might include one or more services to handle conflicting operations. Such a DBE might use one or more of the following: a **Transaction Management Service (Tran-S)**, a **Locking Service (Lock-S)**, a **Timestamping Service (Time-S)**, or a **Deadlock Handling Service (Dead-S)**—all of which will be discussed in Chapters 5 and 6. Similarly, most modern relational DBMSs have a **Fallback and Recovery Service (Rec-S)**, which we will discuss in Chapter 8. The architectural diagram for a DBE like this is shown in Figure 1.5; notice that the services shown are implemented as components inside a single subsystem, called "DBMS" in this diagram. If the DBE is for a DDB, we might

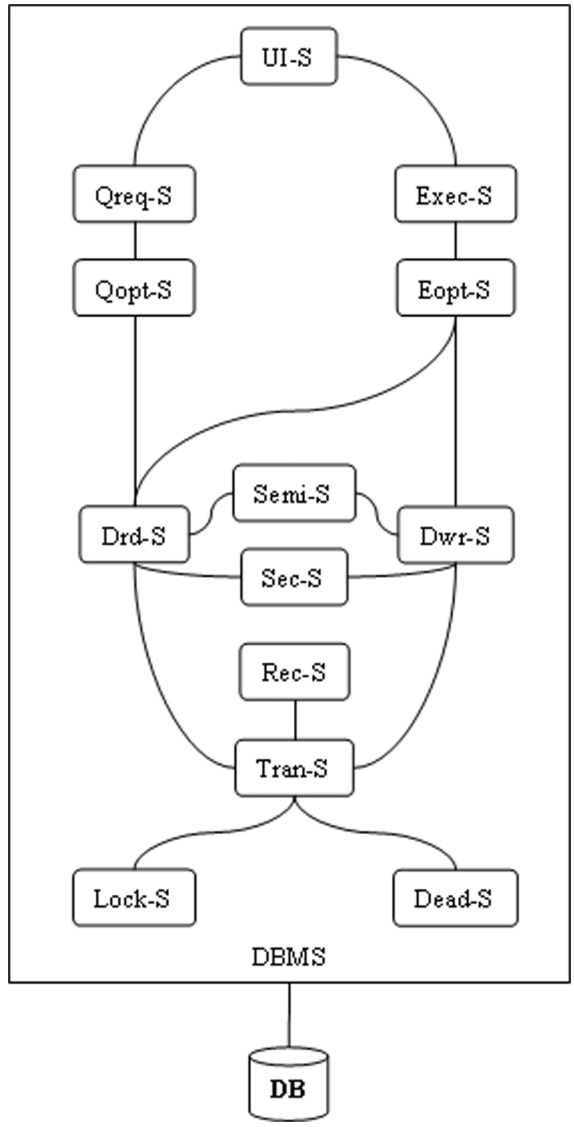

Figure 1.5 DBE architectural diagram for a typical DBMS.

even have a **Replication Service (Repl-S)**, which is not shown in Figure 1.5 (because that architecture is not for a distributed DB). We will discuss the Repl-S in Chapter 7.

1.3.6 Summary Level Diagrams

If we wanted to show the high-level architectural details for an environment containing all of the DBEs just discussed (from Fig. 1.1, 1.2, 1.4, and 1.5) in the same diagram, only the "most visible" subsystem packages would probably be shown (we would not see the components or smaller contained subsystems in this diagram). We can always

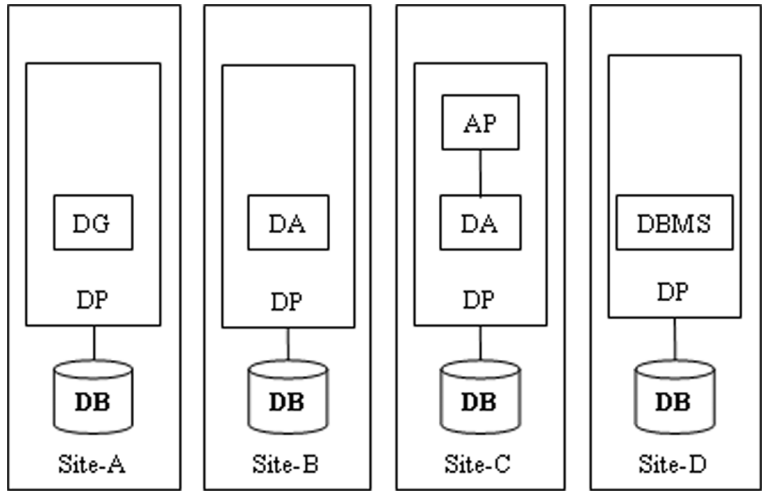

Figure 1.6 Summary level DBE diagrams.

create separate, more detailed diagrams for each subsystem as needed (similar to those shown previously), but at this level of consideration those details would probably not add much value. Figure 1.6 is a more reasonable diagram to use when considering the system at this (or a higher) level. Notice that the site on the left of the diagram (Site-A) contains the DBE from Figure 1.1. Site-B contains the DBE from Figure 1.2, Site-C contains the DBE from Figure 1.4, and Site-D contains the DBE from Figure 1.5. Each DBE consists of a single subsystem, called the DP (which we will discuss further in Section 1.6.2). The DP contains all the necessary subsystems for the centralized DBE at each site. If these sites were participating in a distributed DBE, we might only show the DP box with no internal subsystems displayed.

Although these pictures are useful, they are not a substitute for the actual design and deployment details (most of which are not shown). The diagrams cannot convey many subtle details. For example, suppose the DBE at Site-C did not have a QP; this diagram would still look the same, and the detailed diagram would still look similar to Figure 1.4, except that the QP would be missing. In this scenario, all data retrieval would need to use the DA directly. In other words, the DA's client would need to iterate over all the data and use program logic in the client to discard unwanted data values. Similarly, if the CP at Site-C in this scenario did not support subqueries for its criteria, then the DB-Client might need to iterate over the data values using the DA and use either the CP or the DA to create, modify, or delete data values in the DB. The functionality inside the service components (such as the Semi-S or Sec-S) can also vary greatly between different DBEs: even when the diagram shows these components, we cannot determine how similar or different the implementation details really are by merely looking at the pictures. In fact, since most DBMSs have all of the functionality required by the AP, DA, and DG subsystems, it is also possible for every DBE in the diagram (including those at Site-A, Site-B, and Site-C) to be a DBMS—if this were the case, then we would be providing much more than the minimum requirements for Site-A, Site-B, and Site-C. The opposite is not true, however: Site D must contain a DBMS, with the expected level of functionality required, and not merely a DG, DA, or

AP/DA combination. Once again, all of the specific requirements are not shown in these diagrams and must therefore be found within other diagrams and design documentation.

All of the figures we have looked at so far are architectural diagrams, not deployment diagrams. This means that the components and subsystems listed in the diagram could be bundled into several different possible deployment packages. For example, all of these services could be deployed as a single program, with a single instance installed on a single machine, which we could then call a DBE or DBMS. An alternate deployment of this same architecture could have several instances of this single DBE/DBMS program installed on one or more machines. Even though we have only shown a single DB in the diagram, there might be several DBs deployed—this diagram does not tell us if we can use the same DBE/DBMS for all of these DBs or if there must be a separate instance for each one.

1.4 A NEW TAXONOMY

We mentioned earlier that "DBE" is a new term representing several different possible implementations for similar functionality. Because a DBE considers the system at such an abstract level, it can be used to refer to a wide variety of possible architectures and deployments. In this section, we will present a new taxonomy to be used for classifying all the possible DBE alternatives that we might consider when exploring DB and DDB issues. For our purposes, we will consider a new taxonomy with four levels, presented in order from most abstract to most specific. We hope that this arrangement will reduce the complexity for each successive level and simplify the taxonomic groups (the taxa) containing the environments that we ultimately want to discuss. Like most taxonomies, the extreme cases are most useful for understanding the categorical differences between the taxa, but most real-world DBEs will most likely fall somewhere between the extremes.

The four levels of categorizations (from most abstract to most specific) are:

- COS distribution and deployment (COS-DAD)
- COS closedness or openness (COS-COO)
- Schema and data visibility (SAD-VIS)
- Schema and data control (SAD-CON)

1.4.1 COS Distribution and Deployment

The first level in our taxonomy is the **COS distribution and deployment (COS-DAD)** level. This is perhaps the easiest level to understand, and usually this is the first classification that we want to make when evaluating a particular DBE. The two extreme cases that define this level are the completely **centralized DBE (CDBE)** and the fully **distributed DBE (DDBE)**.

In a completely CDBE, we must deploy all the DBs, and COS instances on a single machine. In other words, placing any of the COS instances or DB instances on a second, separate machine is strictly forbidden. This case includes the initial releases of most traditional DBMSs (such as Oracle, Sybase, Informix, etc.) and many other early DBs that did not have a DBMS-Server (such as dBase, Paradox, Clipper, FoxPro, Microsoft Access, etc.). Most modern releases of DBMSs and DBs have moved away

from this extreme scenario slightly, since we can often deploy the DB-Clients on a separate machine for many of these systems. Similarly, some modern DBMSs have some ability to distribute their DBs (using techniques such as mirroring, etc.), but they are still essentially a CDBE since the "almost distributed DBs" are not really a "true DB" in the same sense as the original DB that they attempt to duplicate.

If each COS instance and each DB instance is deployed on a separate machine, then we have the other extreme (the fully DDBE). Of course, in the real world, we would probably not go to this extreme—in the very least, it is usually not necessary to separate components within the same subsystem from each other, and also not necessary to separate the lowest-level subsystems (such as the DAs) from the DBs (i.e., the files containing the data structure and content) that they access. Typically, the DDBE will consist of one or more "coordinating" server instances (providing services across the entire DDBE) and a set of DBEs that combine to make the DDBE work. We call each of these DBEs a **Sub-DBE (S-DBE)**, because they combine to form the DDBE in a way that is similar to how subsystems combine to form a system. Each S-DBE is a DBE in its own right, which means that each of them is also subject to categorization too using this taxonomy—in particular, each S-DBE can be either a centralized DBE or another distributed DBE. Most S-DBEs are centralized environments, especially for traditional architectures such as the ones we will discuss in Chapter 12. In this book, we are mostly interested in DDBEs and really only consider the CDBEs when we want to evaluate some type of DDBE or examine the S-DBE inside a larger DDBE.

1.4.2 COS Closedness or Openness

The second level in our taxonomy is the **COS closedness or openness (COS-COO)** level. This level considers the software implementation and integration issues for the major subsystems, components, and the DB storage itself. Although we will introduce the two extreme cases for this level (**completely open** and **completely closed**), the COSs in most DBEs will occupy a strata (a series of gradations) within this level rather than either extreme scenario.

There is no such thing as a commercial off-the-shelf (COTS) DDBE that we can simply buy and install. Even if there were such a product, we would still probably want to integrate some of our existing COS instances into the DDBE architecture. Conversely, if our goal was to create our own DDBE "from the ground up," it is doubtful that we would write absolutely everything (including the operating systems, file systems, etc.) completely from scratch. Therefore, every DDBE needs to integrate multiple COS instances, some of which we did not write ourselves, regardless of which architectural alternative we choose to implement. When attempting to integrate all of these COS instances, we need to consider the public interface exposed by each COS. Since most of these COSs were not designed with a DDBE specifically in mind, it is quite possible that the interfaces they expose will not provide everything we need. Simply put, there are several DDBE algorithms and services that can only be implemented when we have complete access to the underlying implementation for the COS instances involved—this is especially true if we want to perform our own DDBE research. For example, suppose we wanted to develop a new mechanism for distributed deadlock handling based on a new variation of locking. Obviously, we could only do this if we were able to see (and perhaps even modify) the underlying locking implementation details for each Sub-DBE in the environment.

Subsystems that provide either "unrestrained access" or at least "sufficient access" to the underlying state and implementation details are **open** to us, while systems that do not are **closed** to us. While most real subsystems provide some level of access, determining the degree to which a particular subsystem is open or closed depends on the actual interfaces it exposes and the type of functionality we are trying to implement. If all the COS instances are open to us, then we have the first extreme case (completely open). If all the COS instances are closed to us, then we have the other extreme case (completely closed).

A completely open DDBE can occur in any of these three scenarios:

- If we write all of the components or service instances ourselves (from scratch)
- If we use free and open source software (FOSS) for all the COS instances that we do not write ourselves
- If we obtain some sort of agreement or arrangement with the COS vendors (for all the COS instances that we do not write ourselves) allowing us the open access we need

The first two scenarios are possible, but in order to satisfy either one, we must completely avoid any non-FOSS, COTS products. Many organizations will be reluctant or unable to do this. The third scenario is also somewhat unusual, since it would most likely involve legal contracts and perhaps additional licensing fees. This means that a completely open DDBE is a very rare situation. A completely closed DDBE can only occur if we do not have sufficient access to the implementation details for any of the COS instances in the DDBE: in other words, only if each COS instance in the DDBE were a black box whose inner mechanisms completely prevented our attempts to understand or integrate. This is a very common situation for the components inside a COTS centralized DBE, where we have no access to the inner workings of the algorithms (such as locking). However, since we cannot buy a COTS distributed DBE, this extreme scenario is impossible for a DDBE.

Each DDBE can be a slightly different situation, but typically, the COS instances become "more open" the further removed they are from the DB. This is because there are many COTS products (such as DBMSs) that we might choose to use in the lower levels of the architecture (closer to the DB), but the higher-level COS software does not exist, which means that we need to write the higher-level COS software ourselves. Remember that any software we write ourselves is always open to us (regardless of whether we give this access to others).

1.4.3 Schema and Data Visibility

The third level in our taxonomy is the schema and data visibility (SAD-VIS) level. In this level, we are considering the DB schema and all the data content stored in each DB in the environment. For a CDBE, this is not very interesting because, typically, there is only a single DB to consider or all the DBs use the same ML. However, a DDBE can be a much more complicated consideration. In a DDBE, each DB is actually under S-DBE, which is a DBE in its own right (usually a CDBE). Different DBEs can potentially have different MLs, which means that the ML for the DDBE and the ML for each S-DBE it contains could theoretically be different. In reality, there are not that many different MLs, so it is not likely that there would be that many different

ones used in the same environment; but the simple fact that there might be different MLs within the same DDBE is important to consider. We will discuss this further in Chapters 2 and 12.

Assuming that there is an appropriate way to combine the different MLs into a single (perhaps different) ML, we can now consider the "combined schema" for all the DBs in the DDBE. If the "most powerful user in the DDBE," which we will call the **DDB administrator (DDBA)**, can see all of the data structure across all the DBs, then we have **total schema visibility (TSV)**. In other words, TSV means that there are no hidden data structures and there is no structure missing from the combined schema but present in one of the S-DBE schemas. Similarly, we can consider the data content for the combined schema and compare it to the data content in each S-DBE. If every data value present in the S-DBE DBs is visible to the DDBA in the DDBE, then we have **total data visibility (TDV)**. In other words, TDV means that there is no hidden data content; there is no data value missing from the DDBE but present in one of the CDBE DBs. When we have both total schema visibility and total data visibility, we have the first extreme case for this level, namely, **total visibility (TV)**. TV can happen in the real world, and in fact, it is a requirement for some particular DDBE architectures, some of which we will discuss in Chapter 12.

It should be obvious that the other extreme case is not possible—if the combined schema was empty and all the data content were hidden, then the DDBE would be completely worthless! If we have some hidden schema, then we have **partial schema visibility (PSV)**. If we have some hidden data, then we have **partial data visibility (PDV)**. Having either PSV, PDV, or both PSV and PDV is referred to as **partial visibility (PV)**. PV is a common occurrence and even a requirement for some particular DDBE architectures. We will also discuss some of these architectures in Chapter 12.

For the sake of completeness, we will mention that there is one more reason why this level is not interesting when we are looking at a single CDBE (as opposed to a DDBE or a Sub-DBE): the TV scenario is really the only possible situation for most CDBEs.

1.4.4 Schema and Data Control

The fourth level in our taxonomy is the schema and data control (SAD-CON) level. In this level, we are considering the set of all operations that we are allowed to perform using only the visible schema and the visible data content (we ignore any operation that would attempt to use hidden schema or content, since it would obviously fail). Once again, our primary focus is on the DDBEs. If the DDBA can perform any and every valid schema operation on the visible combined schema structures, then we have **total schema control (TSC)**. Similarly, if the DDBA can perform any and every possible data operation on the visible data content for the DDBE, then we have **total data control (TDC)**. When we have both TSC and TDC, we have the first extreme scenario, which we call **total control (TC)**. If there is at least one valid schema operation that the DDBA does not have permission to perform for some part of the visible combined schema, then we only have **partial schema control (PSC)**. If there is at least one data operation that the DDBA does not have permission to perform on some subset of the visible data content, then we have **partial data control (PDC)**. Having either PSC, PDC, or both PSC and PDC is referred to as **partial control (PC)**.

Again, it should be obvious that it is impossible for the other extreme scenario to exist in either a CDBE or a DDBE. In other words, if the DDBA cannot perform any valid schema operation on any piece of visible schema, and also cannot perform any valid data operation on any subset of the visible data, then the DBE is unusable for any purpose. Like the previous level, the situation for a CDBE is not very interesting, because we would always expect TC. Although TC is possible for many DDBE architectures (and even required for some of them), PC is also very common for many DDBEs. For example, many DDBE architectures do not provide schema operations, and many real-world implementations are fundamentally read-only with respect to the data content. We will discuss some of these architectures in Chapter 12.

1.5 AN EXAMPLE DDBE

Suppose that our organization has three centralized DBEs deployed as depicted in Figure 1.7. We have one instance of IBM's DB2 DBMS running on our IBM mainframe, which contains mission critical information about the entire organization and supports our payroll, human resources, and customer fulfillment applications. We also have two different departmental groups within our organization—one group is using Oracle's DBMS to maintain project information and the other group is using Microsoft's SQL Server DBMS to keep track of production information.

In this example, even though project information in the Oracle DBE and customer information in the DB2 DBE are accessed independently most of the time, there are times when we need to cross-reference the information in these two systems. Suppose a manager wants to generate a report for each customer, listing details about the customer as well as details about all the company projects involving that customer. In order to

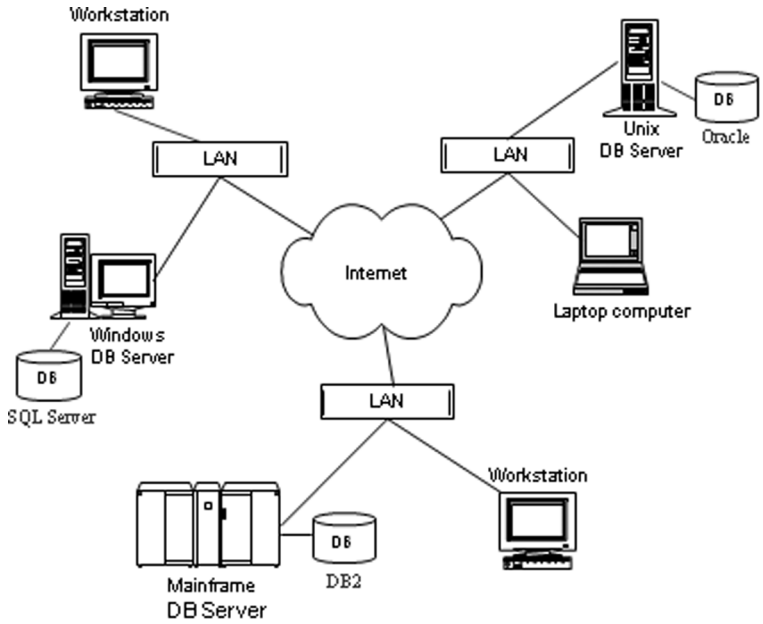

Figure 1.7 Example of the possible deployment of an organization's CDBEs.

satisfy this manager's requirements, we need to combine information that is stored in two different DBEs, controlled by two different types of DBMS. We could choose from two approaches when we attempt to satisfy this manager's requirements: a manual approach and an automated approach.

In the manual approach, we would log into each individual server, write some queries to get the information out of each DB, transfer results from each DB to our personal workstation, and then write a program that merges all the information into the final report format. This is a time-consuming process that requires someone familiar with each hardware platform, each DBE, the communications facilities needed to log into the systems and transfer the files, and the utilities or programming skills needed to fetch the information from each system and to merge the files into the final result. Even when we have someone who can perform all these tasks, we do not have any easy way to ensure that final report is valid.

For the automated approach, our company needs to combine the three CDBEs mentioned above into a new, single, DDBE that we need to implement. We can utilize the services of local DBMSs, each running on a separate computer, and the services of the communication subsystem (Ethernet, Token Ring, and Internet) to coordinate the necessary read or query operations. Ideally, users of our new DDBE will be completely unaware that the data content is dispersed across different computers and is controlled by different DBMS products—they have the illusion that all of our combined data content is stored and controlled locally by one system (the DDBE). Our DDBE users do not need to know anything about the DBMSs we have. They do not need to know the MLs required for each DBMS, anything about computer hardware running the DBEs, or any details about how these hardware and software systems are interconnected. Our users can send a query request to the DDBEs query processor that handles all the necessary coordination, execution, and merging automatically. Although using this new DDBE system is very easy, implementing it can be quite difficult.

1.6 A REFERENCE DDBE ARCHITECTURE

In discussing the theory behind a DDB, we will use the DDBE architecture discussed in this chapter as the reference. We will discuss other architectures in Chapter 12. For the rest of this chapter, and to set the basis for the discussion of theory and issues in a distributed database environment, we will further divide the architecture of a DDBE into an Information Architecture and a Software Architecture.

1.6.1 DDBE Information Architecture

The information architecture for a centralized database environment (CDBE) conforms to the American National Standards Institute (ANSI) standard proposed and approved by the Standard Planning and Requirements Committee (SPARC) [SPARC75] [SPARC76] [SPARC77] called the ANSI/SPARC three-level schema architecture as depicted in Figure 1.8. This architecture provides for three separate layers of abstraction—the external, conceptual, and internal schemas. Users of such a system are assigned external schemas (also known as external views), which they then use to access the information in the underlying database. These views are created from the conceptual schema that encompasses the entire set of data in the database. The

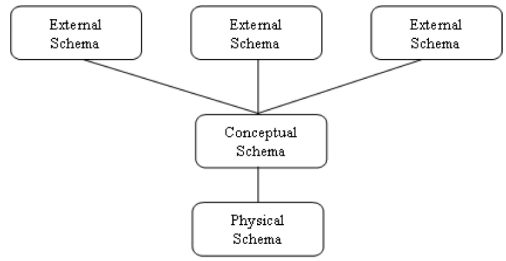

Figure 1.8 ANSI/SPARC three-level schema architecture.

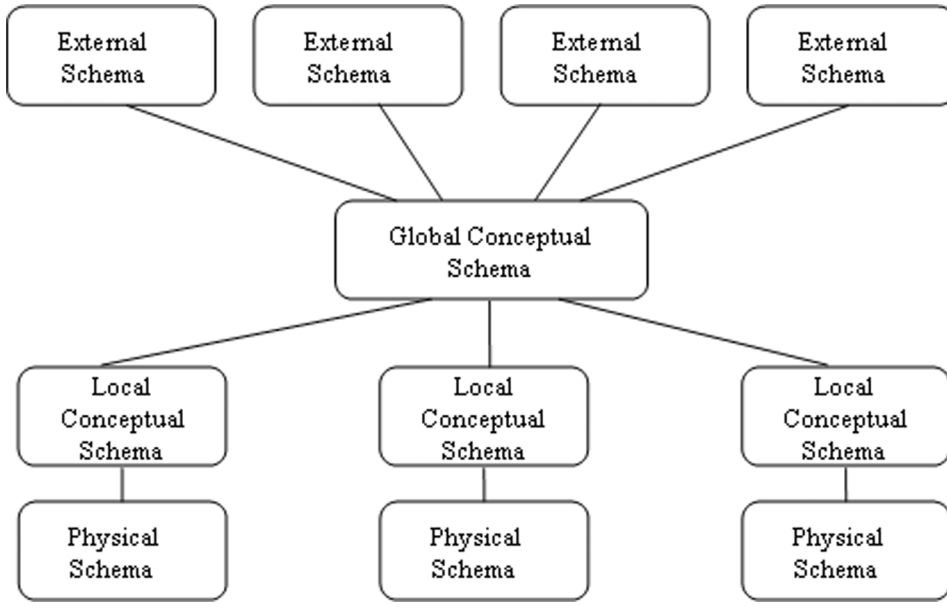

Figure 1.9 DDBE schema architecture.

conceptual schema in a relational system, for example, represents all the information in a database as a set of relations or tables. At the lowest layer of abstraction, the internal (or physical) schema represents the data in the raw format. This representation is not visible to the end user and provides for the application data independence feature of all DBMSs.

Although, the three-level schema architecture satisfies the needs of a CDBE, it is not sufficient for a DDBE [Sheth90]. In a DDBE, users' views must combine information across different S-DBEs. As such, their views need to be built on top of an integrated view of local conceptual schemas from the participating DBEs. This requirement adds a new level of abstraction, the **global conceptual schema (GCS)**, to the three-level schema architecture; this new architecture is depicted in Figure 1.9.

The GCS is an integrated view of all local conceptual schemas. It provides the basis for generating the external views for distributed system users. The local conceptual schemas provide a local view of data stored locally at each S-DBE. Therefore, GCS can only provide a global conceptual view of data and nothing more. In a distributed

system where each individual S-DBE is a DBMS using the relational system, the GCS provides information about all tables in the environment, all primary keys, all foreign keys, all constraints, and so on. However, the GCS does not contain any information about where any individual table is stored, how any individual table is fragmented, or even how many copies of each fragment there are in the DDBE. We need additional information (not contained in the GCS) to provide for location, fragmentation, and replication transparencies. We call this augmented GCS (with the additional required information included) a **global data dictionary** (**GDD**).

The GDD contains information about all the data that is available to a distributed system user. The GDD contains, in addition to what is in the GCS, information pertaining to data location, data fragmentation, and data replication.

For example, in the relational world, GDD contains five submodels:

1. *Global Conceptual Schema (GCS).* The GCS has information about the tables, columns, data types, column constraints, primary keys, foreign keys, and so on. This part of the GDD provides for application data independence, which is required by all DBMS systems according to the ANSI/SPARC standard.
2. *Data Directory (DD).* The DD has information about the location of the data fragments. This information typically identifies the site location by specifying the Universal Resource Locator (URL), site name, IP address, and so on for the site containing the data. This part of GDD enables a DDBE to provide for location transparency.
3. *Fragmentation Directory (FD).* The FD has information about data fragments in the system. The FD typically contains conditions used for creation of horizontal fragments, join column for vertical fragments, columns that are part of the vertical fragments, primary key of the fragments, and so on. This part of GDD provides for fragmentation transparency.
4. *Replication Directory (RD).* The RD has information about replication. This typically includes the number of copies that exist for each table or a fragment of a table. Note that this information in conjunction with the DD information is enough to locate every copy of any fragment or a table. This part of GDD allows a DDBE to provide for replication transparency.
5. *Network Directory (ND).* The ND has information about the topology, communication link speed, and so on for all the sites participating in the DDBE. This part of GDD enables a DDBE to provide for network transparency.

1.6.2 DDBE Software Architecture

Like a centralized DBE, a distributed DBE consists of two main software modules called the **application processor (AP)** and the **data processor (DP)**. Each DP provides the services necessary to connect a local DBE to the distributed environment. The actual DBE requirements and the specific requirements for the DP services depend on the approach we are using to implement the software components of the DDBE (Section 1.3 discussed these services, and Figure 1.6 provided four different examples of how a DP might bundle these services). There are two approaches to software implementation for a DDBE—they are called top–down and bottom–up. We will discuss the details of these approaches in Chapter 12. Here, we briefly outline the differences between them. Figure 1.10 depicts the architecture of a top–down DDBE implementation.

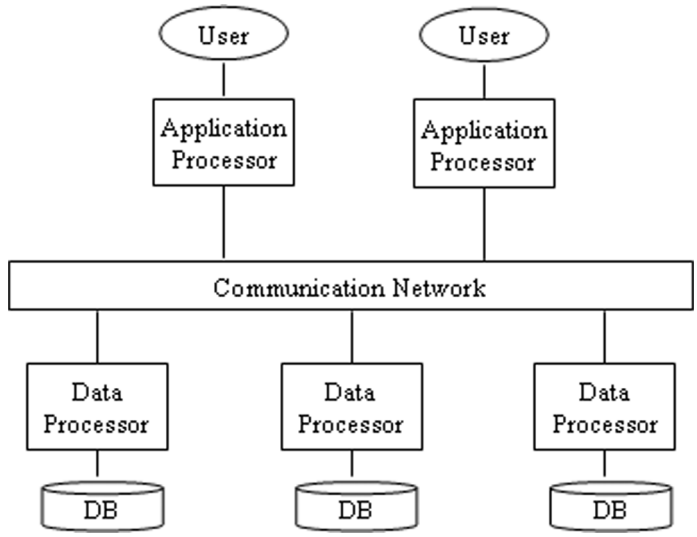

Figure 1.10 Top–down DDBE software architecture.

When we develop the software components of a DDBE, the APs and DPs can be deployed separately and can cooperate over a communication network. The AP is responsible for controlling the user interface and distributed transaction management, while the DP handles local query optimization, local transaction recovery, and local transaction execution support. This architecture parallels the information architecture discussed earlier. The AP uses the GCS and the GDD to handle users' requests at the global (distribution) level, while the DPs use the LCSs to execute the subrequests locally.

When we develop the software components of a DDBE bottom–up, we are really integrating existing database management systems as local DBEs into the DDBE. In this approach, many of the DP responsibilities can be delegated to the local DBEs. This reduces the DP component to nothing more than a thin wrapper surrounding the local DBE for each site. Figure 1.11 depicts the software components of this approach.

Figure 1.12 depicts the details of the application processor and data processor for both approaches mentioned above. This example shows a system with one AP and *N* DPs. Note that this is a generic logical architecture and does not imply deployment—we will cover software deployment in Chapter 12. The following two sections outline the software components of a DDBE. These components are necessary regardless of the approach used to develop the system.

1.6.2.1 Components of the Application Processor The application processor is composed of two main subsystems. They are the **global transaction manager (GTM)** and the **distributed execution manager (DEM)**. The GTM itself is divided into five subsystems—the **user interface (UI)** module, the Decomposer, the Merger, the Optimizer, and the Planner. The GTM also contains the **global data dictionary (GDD)**. The GTM's overall responsibility is to manage user requests (queries and commands wrapped inside transactions). The UI accepts user requests and translates them into an internal representation suitable for processing. After translation, the request is passed to

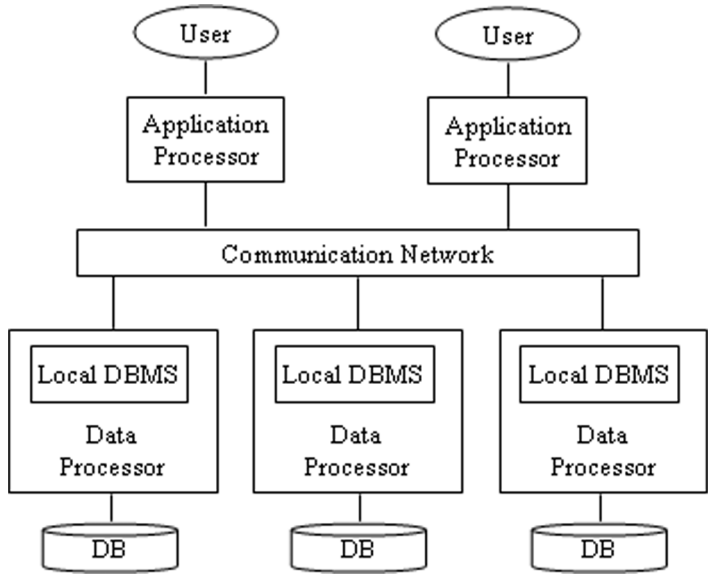

Figure 1.11 Bottom–up DDBE software architecture.

the Decomposer to break it up, if necessary, into subrequests that need to be processed by individual local DP (and DBMS) systems. The Decomposer parses the request first to find out the name of the tables (fragments), their columns, and predicates (join predicates and nonjoin predicates). The Decomposer then looks up the GDD to find out information about the location of the tables/fragments that are in the request. Based on this information, the Decomposer generates a set of local subrequests. These subrequests can run on individual local DP systems. The local subrequests are then handed out to the Optimizer. Similar to a local DBMS, the Optimizer and Planner work together and use the information they get from the GDD about replication, fragmentation, and communication link speeds to generate an optimized execution plan. The plan is then given to the DEM to be carried out.

The underlying DDBE Platform must supply the necessary services to guarantee the delivery of the requests and the responses. There are several implementation alternatives to choose from, but the primary role of whichever one we choose to use is to hide the specifics of these communication requirements from the DEM. We will discuss the DDBE Platform requirements in Chapter 14, and we will look at some of the implementation alternatives in Unit 3.

The job of the DEM is straightforward. The DEM simply executes the steps in the distributed execution plan by coordinating with the **local execution managers (LEMs)** contained within the DPs at the target sites. Once the local sites return the results to the DEM, they are passed to the Merger subsystem within GTM for assembly. For example, the Merger may have to join rows from two vertical partitions of a given table or may have to union rows from two horizontal partitions of a given table. We will discuss the rules for merging result fragments in Chapter 2.

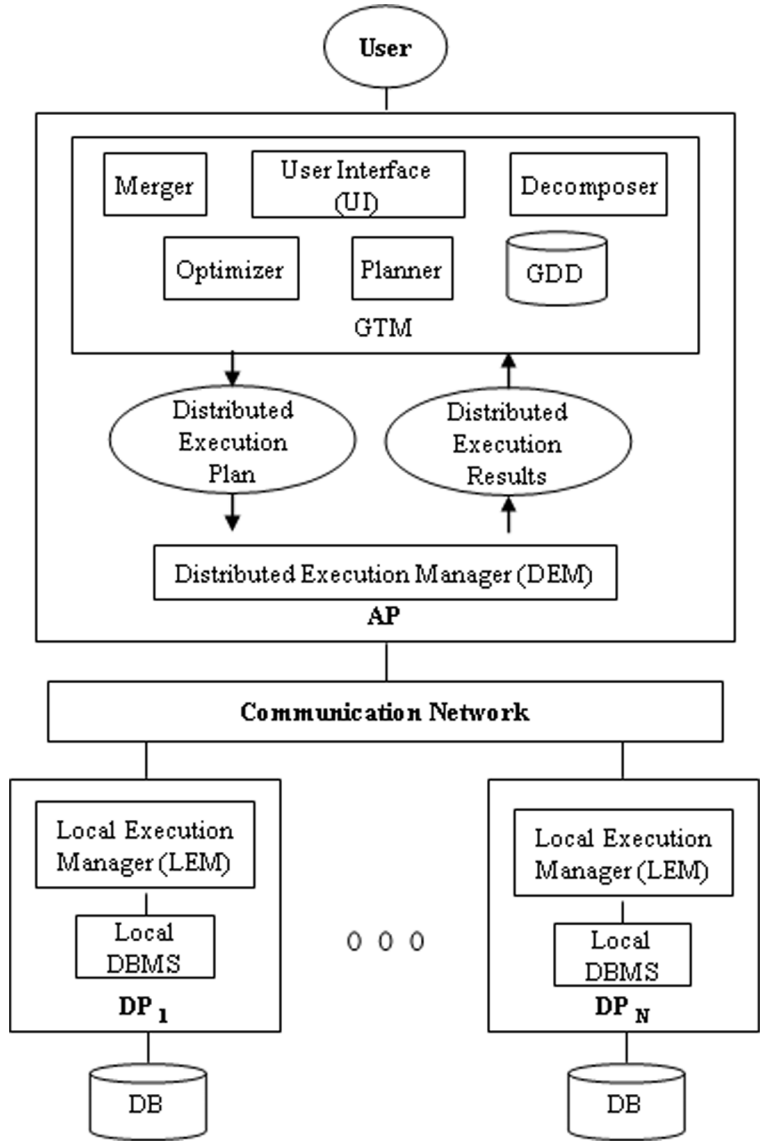

Figure 1.12 Generic DDBE software architecture.

1.6.2.2 Components of the Data Processor The data processor consists of the LEM and the necessary data services we discussed in Section 1.3. Once again, the DDBE Platform is responsible for providing the necessary communications support at the local site. It guarantees end-to-end message delivery and hides details and issues with communications from the rest of the modules at each site. The primary responsibility of each LEM is to act on behalf of the DEM at the local site. It will receive the subrequests that pertain to the data at its site, interface with the local DB (through the

local DBMS, DA, or DG), run the requests, collect the results, and then pass those results back to the DEM. When the subrequest is a query, these results contain actual data content; when the subrequests are commands, these results might contain details about the number of rows affected. In both cases, these results contain execution status information (success or failure), and perhaps additional details about the execution (warning messages, error messages, etc.)

1.7 TRANSACTION MANAGEMENT IN DISTRIBUTED SYSTEMS

As discussed, transactions in a distributed system can access data at more than one site. Each transaction consists of a number of subtransactions that must run at a site where the data is. Each subtransaction represents an agent for the distributed transaction. Each agent must be able to commit at a local site for the global transaction to commit. If the global transaction cannot commit, none of the subtransactions commit either. Figure 1.13 shows a fund transfer transaction (T1) that transfers the amount "amt" from account x at Site A to account y at Site B. We designate the agents of transaction T1 at Sites A and B as T1A and T1B.

How does the system carry out this distributed transaction? The answer depends on the overhead of communication involved in execution of the transaction. Before getting into the details of executing this transaction, let us review what we have already outlined in this chapter. The global transaction manager (GTM) produces the distributed execution plan. This plan, as we will discuss in Chapter 4, is an optimized plan that the DEM must execute. The DEM is responsible for carrying out the plan by coordinating the activities of some LEMs that act as the distributed transaction's agents.

```
begin T1
     begin T1A
               Read bal(x);
               if account not found then
                {abort(T1A); print "account not found"; exit;
                };
               bal(x) = bal(x) - amt;
               if bal(x) < 0 then
                {abort(T1A); print "insufficient funds"; exit;
                };
               write bal(x); commit T1A;
     end T1A;
     begin T1B
               Read bal(y);
               if account not found then
                {abort(T1B); print "account not found"; exit;
                };
               bal(y) = bal(y) + amt;
               write bal(y); commit T1B;
     end T1B;
     commit T1;
end T1;
```

Figure 1.13 A distributed fund transfer transaction.

```
Begin T1
     Select Bal Into X From Acct1
     Where A# = 100;

     Select Bal Into Y From Acct2
     Where A# = 200;

     Select Bal Into Z From Acct3
     Where A# = 300;

     REPORT X+Y+Z TO USER
End T1;
```

Figure 1.14 An example of a three-site distributed transaction.

Example 1.1 Consider a three-site system as shown in Figure 1.14, where transaction T1 enters the system at Site 1. Transaction T1 needs to reads the total balance for accounts 100, 200, and 300. Let's assume that the account table is horizontally fragmented across three sites. Site 1 holds accounts 1 to 150, Site 2 holds accounts 151 to 250, and Site 3 holds accounts 251 and higher. If X, Y, and Z represent the balances for accounts 100, 200, and 300, respectively, we can execute this distributed transaction as follows:

```
Send "necessary commands" to Site 1 to read "X" from DB1;
Send "necessary commands" to Site 2 to read "Y" from DB2;
Send "necessary commands" to Site 3 to read "Z" from DB3;
Receive "X" from Site 1;
Receive "Y" from Site 2;
Receive "Z" from Site 3;
Calculate Result = X + Y + Z;
Display Result to User;
```

Note that the Decomposer must know that the account table has three horizontal fragments called Acct1, Acct2, and Acct3. Based on the fragmentation assumptions we have made, account 100 is stored at Site 1, account 200 is stored at Site 2, and account 300 is stored at Site 3. For this example, the DEM needs to send two commands to the LEM at Site 2. One command is to read the balance of account 200. This command in SQL is "Select bal into Y from Account where A# = 200." The second command is an instruction to Site 2 to send the results back to Site 1. Site 1 packages these two commands in one message and sends the message to Site 2. In response, Site 2 sends the balance of account 200 in variable Y, as part of a response message, back to Site 1. Similar commands are sent to Site 1 and Site 3.

```
Distributed Execution Plan for DEM at Site 1:
Send "Select Bal into X From Acct1 Where A# = 100;
      Send X to Site 1"
to Site 1;
Send "Select Bal into Y From Acct2 Where A# = 200;
      Send Y to Site 1"
to Site 2;
Send "Select Bal into Z From Acct3 Where A# = 300;
      Send X to Site 1"
to Site 3;
Receive X from Site 1;
Receive Y from Site 2;
Receive Z from Site 3;
Calculate Result = X+Y+Z;
Display Result to User;

Plan for LEM at Site 1:
Select Bal Into X From Acct1 Where A# = 100;
Send X to Site 1;

Plan for LEM at Site 2:
Select Bal Into Y From Acct1 Where A# = 200;
Send Y to Site 1;

Plan for LEM at Site 3:
Select Bal Into Z From Acct1 Where A# = 300;
Send Z to Site 1;
```

Figure 1.15 Detailed execution plans for DEM and its LEMs.

Since Site 1 cannot add variables X, Y, and Z until all three variables arrive at its site, Site 1 must perform a blocking receive to achieve the necessary synchronization. A blocking receive is a receive command that blocks the process that executes it until the sought data has arrived at the site. Figure 1.15 shows the detailed distributed execution plan and individual plans for each of the three LEMs. In order for the DEM and the LEMs to be able to carry out their tasks, some services are required.

At a minimum, the system must have the ability to:

- Find the server address where the agent is going to run
- Remotely activate a process—DEM must activate its agents at remote sites
- Synchronize processes across sites—DEM must wait for LEMs to send their results to continue
- Transfer commands and receive results from its agents
- Transfer temporary data across sites—a select may return more than one row as its results

- Find out and ignore any commands that might be out of order or should not be processed
- Remotely deactivate a process—DEM needs to deactivate its agents

In the approach we used for coordinating the activities of the DEM and its LEMs, the DEM acted as the coordinator (Master) and the LEMs acted as the followers (Slaves) [Larson85]. Figure 1.16 depicts the Master–Slave approach to distributed execution management.

In the Master–Slave approach, the DEM synchronizes all temporary data movements. In this approach, LEMs send temporary data only to the DEM. Sometimes it is more beneficial to send temporary data from one LEM directly to another LEM. In this case, the DEM needs to send the proper commands and actions to the LEMs so that the LEMs can synchronize their own activities. Larson [Larson85] calls this type of control triangular distributed execution control. Figure 1.17 shows this type of control across one DEM and two LEMs.

Example 1.2 For this example, we assume the system has four sites. The user enters transaction T into the system at Site 0. There are three tables in the system defined as:

```
Person (SSN, Name, Profession, sal) stored at Site 1
Car (Name, Make, Year, SSN) stored at Site 2
Home (Address, No-of-rooms, SSN) stored at Site 3
```

For this distributed system, we want to print car and home address information for every person who makes more than $100,000. Applying a triangular distributed transaction control approach to this example results in the local execution plans shown in Figure 1.18. There are four LEMs in this system. These LEMs are numbered 0 through 3, indicating the site at which each runs. Although we do not have any data stored at Site 0, we must have one LEM there to perform the final join across the temporary tables that other LEMs send to it. This LEM acts as the merger, which we discussed in Section 1.1.

The synchronization mechanism used to coordinate the activities of the LEMs is implemented via the send and receive commands. When a LEM executes a receive command, it blocks itself until the corresponding data arrives at its site (this is an example of a blocking execution call or a blocking messaging call; see Section 14.3.1

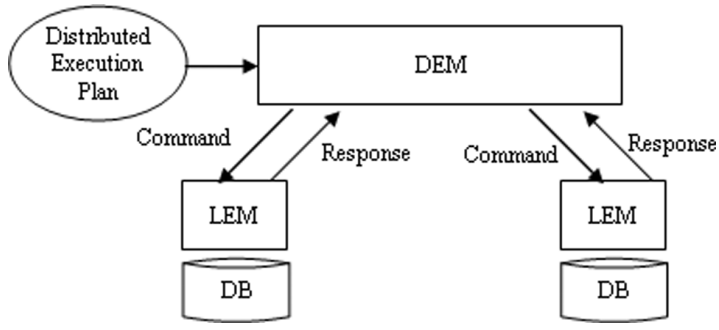

Figure 1.16 Master–Slave distributed transaction control.

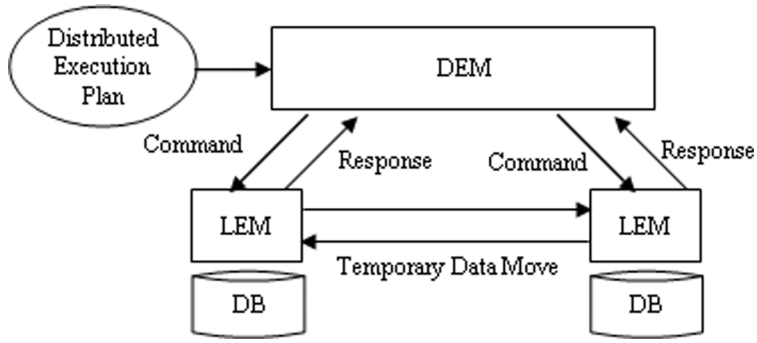

Figure 1.17 Triangular distributed transaction control.

```
@Site 1 do:
  Create table T1 as
    Select SSN, Name, Sal
    From Person
    Where sal > 100,000;
  Send T1 to S0;
  Create table T11 as
    Select SSN
    From Person
    Where sal > 100,000;
  Send T11 to Site 2;
  Send T11 to Site 3;
                (a)
```

```
@Site 2 do:
  Receive T11;
  Create table T2 as
    Select Name, Make, Year, T11.SSN
    From Car, T11
    Where T11.SSN = Car.SSN;
  Send T2 to Site 0;
                (b)
```

```
@ Site 3 do:
  Receive T11;
  Create table T3 as
    Select  Address, T11.SSN
    From Home, T11
    Where T11.SSN = Home.SSN;
  Send T3 to Site 0;

                (c)
```

```
@Site 0 do:
  Receive T1;
  Receive T2;
  Receive T3;
  Select *
  From T1, T2, T3
  Where T1.SSN= T2.SSN
  and T1.SSN=T3.SSN;
  Display Results to User;
                (d)
```

Figure 1.18 Local execution plans for Example 1.2.

for further details). For our example, the only LEM that can start processing immediately after activation is LEM 1 (see Figure 1.18a). That is because this LEM does not execute a receive command right away. On the other hand, all the other LEMs execute a receive command first. This action blocks these processes until the necessary data arrives at their sites.

It should be clear by now that the distributed execution plan—the plan that the DEM must run—contains the local commands that must be sent to the four LEMs.

```
Send          "Create table T1 as
              Select SSN, Name, Sal
              From Person Where sal > 100,000;
              Send T1 to Site 0;
              Create table T11 as
              Select SSN
              From Person
              Where sal > 100,000;
              Send T11 to Site 2;
              Send T11 to Site 3;"
To LEM @ Site 1;

Send          "Receive T11;
              Create table T2 as
              Select Make, Year, T11.SSN
              From Car, T11
              Where T11.SSN = Car.SSN;
              Send T2 to Site 0;"
To LEM @ Site 2;

Send          "Receive T11;
              Create table T3 as
              Select  Address, T11.SSN
              From Home, T11
              Where T11.SSN = Home.SSN;
              Send T3 to Site 0;"
To LEM @ Site 3;

Send          "Receive T1; Receive T2; Receive T3;
              Select *
              From T1, T2, T3
              Where T1.SSN= T2.SSN and T1.SSN=T3.SSN;
              Display Results to User;"
To LEM @ Site 0;
```

Figure 1.19 Distributed execution plan for Example 1.2.

Figure 1.19 depicts the distributed execution plan for this example. This execution strategy obviously uses a triangular control approach. As seen from this figure, the distributed execution plan contains only four messages that the DEM must send. Each one of these messages consists of the commands that one LEM must run. Obviously, since the DEM and LEM 0 are at the same site (Site 0), the message that the DEM sends to LEM 0 is a local message.

Example 1.3 Assume the table "EMP(EmpID, Name, Loc, Sal)" is horizontally fragmented into fragments MPLS_Farg, LA_Frag, and NY_Frag. Each horizontal fragment stores information about those employees who work at the corresponding location. LA_Frag stores information about employees who work in LA, NY_Frag contains information about employees who work in NY, and MPLS_Frag contains information about employees who work in MPLS. Transaction T enters the DEM deployed in St. Paul and needs to figure out the average salary for all employees in the company. An unoptimized query execution plan would try to reconstruct the EMP table from its fragments by sending all rows from all fragments to St. Paul and then using the SQL aggregate function AVG(sal) as

```
Select AVG(sal) From EMP;
```

If each row of the table takes one message to send from any site to any other site, this strategy would require as many messages as the total number of employees in the organization. However, we can execute this transaction using a small number of messages. The idea is not to materialize the EMP table but to figure out the average salary from the three fragments mathematically as

```
AVG(sal) =
(SAL_LA+SAL_MPLS+SAL_NY)/(count_LA+count_MPLS+count_NY)
```

In this formula, each "SAL" variable represents the total salary for the employees in its corresponding fragment, and each "count" variable indicates the total number of employees in its fragment. Once we realize this, we can use the Master–Slave control as shown Figure 1.20 to get the results. Note that in this figure, we use the LEM at St. Paul as the merger of the results from the local sites. Since this LEM does not have to perform any database operation, there is no need for a DBMS at St. Paul. In a

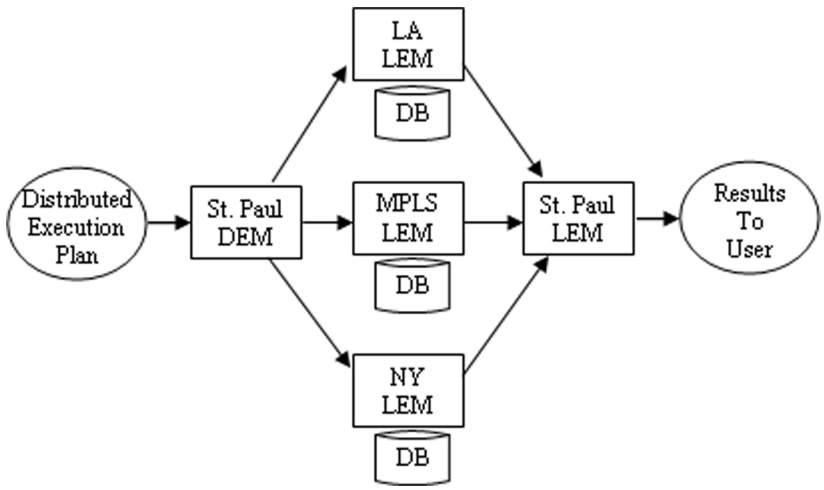

Figure 1.20 Master–Slave execution plan for query in Example 1.3.

```
LEM @ LA:
Select count(*), SUM(Salary) from LA_Frag into count_LA, SAL_LA;
Send "count_LA and SAL_LA" to LEM @ St. Paul;
Communication cost = 1 message

LEM @ MPLS:
Select count(*), SUM(Salary) from MPLS_Frag into count_MPLS, SAL_MPLS;
Send "count_MPLS and SAL_MPLS" to LEM @ St. Paul;
Communication cost = 1 message

LEM @ NY:
Select count(*), SUM(Salary) from NY_Frag into count_NY, SAL_NY;
Send "count_NY and SAL_NY" to LEM @ St. Paul;
Communication cost = 1 message

LEM @ St. Paul:
Receive count_LA and SAL_LA from LEM @ LA;
Receive count_MPLS and SAL_MPLS from LEM @ MPLS;
Receive count_NY and SAL_NY from LEM @ NY;
Calculate AVG = (SAL_LA+SAL_MPLS+SAL_NY)/(count_MPLS+count_LA+count_NY);
Display to the user;
Communication cost = 0

DEM @ NY:
Send "Select count(*), SUM(Salary) from MPLS_Frag into count_MPLS, SAL_MPLS;
      Send "count_MPLS and SAL_MPLS" to LEM @ St. Paul;" to LEM @ MPLS;
Send "Select count(*), SUM(Salary) from LA_Frag into count_LA, SAL_LA;
      Send "count_LA and SAL_LA" to LEM @ St. Paul;" to LEM @ LA;
Send "Select count(*), SUM(Salary) from NY_Frag into count_NY, SAL_NY;
      Send "count_NY and SAL_NY" to LEM @ St. Paul;" to LEM @ NY;
Communication cost = 3 messages
```

Figure 1.21 Distributed and local execution plans for query in Example 1.3.

nonoptimized approach as discussed above, the St. Paul LEM would require a DBMS to run the select statement that calculates the average salary.

Figure 1.21 shows the local execution plans for each LEM and the distributed execution plan for the DEM. As seen from this figure, the Master–Salve execution control takes six messages. A careful reader realizes that using a triangular execution control would require only four messages. We leave this as an exercise for the readers.

1.8 SUMMARY

Centralized database environments and specifically distributed database environments are complex systems composed of many subsystems, components, and services. This book discusses issues in implementing a DDBE. The emphasis of the book is on the practical aspects of implementing a DDBE. In this chapter, we have outlined the architecture of a DDBE and have discussed approaches to controlling the execution of a transaction.

1.9 GLOSSARY

Data Content A data value (or set of data values) stored in a database.

Data Model (DM) A representation of the structure of data.

Data Modeling Language (ML) See **Modeling Language**.

Data Read Service (Drd-S) A DBE service that provides the ability to retrieve data from the database.

Data Schema See **Data Model**.

Data Write Service (Dwr-S) A DBE service that provides the ability to write data to the database.

Database (DB) A collection of data organized according to one or more data models.

Database Client (DB-Client) An agent interfacing to a database either interactively or as an application.

Database Environment (DBE) A collection of one or more DBs along with any software providing at least the minimum set of required data operations and management facilities.

Database Management System (DBMS) A collection of software services or components that control access to, and modification of, data in a database.

Database Server (DB-Server) A collection of software services (or a particular deployed instance of those services) that handles the interface to the database for database clients.

Deadlock Handling Service (Dead-S) The service that handles deadlocks in a DBE.

Distributed Database (DDB) A collection of software that allows several databases to operate as though they were part of a single database, even though they are actually separate and possibly deployed at different sites.

Execution Optimization Service (Eopt-S) A DBE service that optimizes execution of the user's create, update, and delete commands.

Execution Service (Exec-S) A DBE service that provides the ability to create, update, and delete data.

Fallback and Recovery Service (Rec-S) A DBE service that guarantees availability of the database even when failures happen.

Locking Service (Lock-S) A service that provides for data component locking capability.

Metadata A piece or set of information about data.

Modeling Language (ML) A vocabulary and set of rules used to define a model. Typically, there are also hints or suggestions for diagrammatic representations of the model, but this is not strictly speaking a requirement. For example, consider the Entity Relationship Modeling technique described by Dr. Chen, the Relational Model as formalized by Dr. Codd and the ANSI standard, or each proprietary Structured Query Language (SQL) defined by a particular RDBMS implementation.

Persistent Data Data that is stored to secondary storage (hard drive).

Query A formulation of a user's request to retrieve data, typically involving some criteria that control the filtering or formatting of the data that is returned.

Query Optimization Service (Qopt-S) A DBE service that optimizes the plan used to execute queries.

Query Request Service (Qreq-S) A DBE service that allows users to query the data in a database.

Replication Service (Repl-S) A DBE service that manages multiple copies of the same data (including duplicate tables and/or duplicate databases).

Secondary Storage A storage facility that does not lose its contents upon power shutdown (hard drive).

Security Service (Sec-S) A service that guards the database against unwanted and unauthorized access.

Semantic Integrity Service (Semi-S) A service that preserves the integrity and consistency of the data in a database.

Service A logical collection (specification/design) of well-defined, cohesively related functionality or a software instance (physical collection) that implements this functionality.

Service Consumer A component or subsystem that uses a set of functionality implemented by some other component or subsystem (consumes a service implemented by some service provider).

Service Provider A component or subsystem that implements a set of functionality and makes it available to other components and subsystems (provides a service to be used by some service consumer).

Subsystem A collection of components and/or subsystems that is part of a larger system but also a system in its own right.

Timestamping Service (Time-S) A DBE service that creates a real and/or logical clock reading as a timestamp.

Transaction Management Service (Tran-S) A DBE service guaranteeing that either all the CRUD operations within a well-defined set (called a transaction or a logical-unit-of-work) are committed to the database or none of them are committed to the database.

REFERENCES

[Larson85] Larson, J., and Rahimi, S., *Tutorial: Distributed Database Management*, IEEE Computer Society Press, New York, 1985, pp. 91–94.

[Sheth90] Sheth, A., and Larson, J., "Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases," *Computing Surveys*, Vol. 22, No. 3, September 1990, pp. 183–236.

[SPARC75] ANSI/X3/SPARC Study Group on Data Base Management Systems: (1975), *Interim Report. FDT*, ACM SIGMOD bulletin, Vol. 7, No. 2.

[SPARC76] "The ANSI/SPARC DBMS Model," in *Proceedings of the Second SHARE Working Conference on Data Base Management Systems*, Montreal, Canada, April 26–30, 1976.

[SPARC77] Jardine, D., *The ANSI/SPARC DBMS Model*, North-Holland Publication, Amsterdam, 1977.