right. However, since the DC manager and the DBMS are clearly required to work harmoniously together, they are sometimes regarded as equal partners in a higher-level cooperative venture called the **database/data-communications system** (DB/DC system), in which the DBMS looks after the database and the DC manager handles all messages to and from the DBMS, or more accurately to and from applications that use the DBMS. In this book, however, we shall have comparatively little to say about message-handling as such (it is a large subject in its own right). Section 2.12 does briefly discuss the question of communication *between distinct systems* (i.e., between distinct machines in a communications network), but that is really a separate topic.

2.10 Client/Server Architecture

Preceding sections of this chapter have discussed the so-called ANSI/SPARC architecture for database systems in some detail. In particular, Fig. 2.3 gave a simplified picture of that architecture. In this section we take a look at database systems from a slightly different perspective. The overall purpose of such systems, of course, is to support the development and execution of database applications. From a high-level point of view, therefore, a database system can be regarded as having a very simple two-part structure, consisting of a **server** (also called the **backend**) and a set of **clients** (also called **frontends**). Refer to Fig. 2.4.

We explain the figure as follows.

- The server is the DBMS itself. It supports all of the basic DBMS functions discussed in Section 2.8—data definition, data manipulation, data security and integrity, and so on. In particular, it provides all of the external, conceptual, and internal level support discussed in Sections 2.3–2.6. Thus, "server" in this context is just another name for the DBMS.

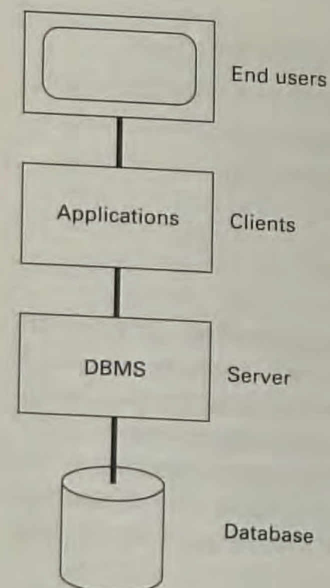

FIG. 2.4 Client/server architecture

- The clients are the various applications that run on top of the DBMS—both user-written applications and builtin applications, i.e., applications provided either by the vendor of the DBMS or by some "third-party" software vendor. As far as the server is concerned, of course, there is no difference between user-written and builtin applications—they all use the same interface to the server, namely the external-level interface discussed in Section 2.3.

  Note: Certain special "utility" applications might constitute an exception to the foregoing. As mentioned in Section 2.5, such applications sometimes need to operate directly at the internal level of the system. Such utilities are best regarded as integral components of the DBMS, rather than as applications in the usual sense. Utilities are discussed in more detail in the next section.

Applications in turn can be divided into several reasonably well-defined categories, as follows.

1. First, *user-written applications*. These are basically regular application programs, written (typically) either in a conventional programming language such as C or COBOL or in some proprietary language such as FOCUS—though in both cases the language needs to be coupled somehow with an appropriate data sublanguage, as explained in Section 2.3.
2. Second, *vendor-provided applications* (often called **tools**). The overall purpose of such tools is to assist in the process of creating and executing other applications!—i.e., applications that are tailored to some specific task (though the created application might not look much like an application in the conventional sense; indeed, the whole point of the tools is to allow users, especially end users, to create applications *without* having to write conventional programs). For example, one of the vendor-provided tools will be a query language processor, whose purpose of course is to allow end users to issue *ad hoc* queries to the system. Each such query is basically nothing more than a small (or maybe not so small) tailored application that is intended to perform some specific application function.

   Vendor-provided tools in turn divide into a number of distinct classes:

   - query language processors
   - report writers
   - business graphics subsystems
   - spreadsheets
   - natural language processors
   - statistical packages
   - copy management tools
   - application generators (including "4GL" processors)
   - other application development tools, including computer-aided software engineering (CASE) products

Meta Platforms, Inc.
Exhibit 1024
Page 024

and so on. Details of such tools are beyond the scope of this book; however, we remark that since (as stated above) the whole point of a database system is to support the creation and execution of applications, the quality of the available frontend tools is, or should be, a major factor in "the database decision" (i.e., the process of choosing the right system for a given customer). In other words, the DBMS *per se* is not the only factor that needs to be taken into account, nor even necessarily the most significant factor.

We close this section with a forward pointer. Since the overall system can be so neatly divided into two parts (server and clients), the possibility arises of running the two on *different machines*. In other words, the potential exists for **distributed processing**. Distributed processing means that distinct machines can be connected together into some kind of communications network, in such a way that a single data processing task can be spread across several machines in the network. (In fact, so attractive is this possibility—for a variety of reasons, mainly economic—that the term "client/server" has come to apply almost exclusively to the case where the server and clients are indeed on different machines. This usage is sloppy but very, very common.) We will discuss distributed processing in more detail in Section 2.12.

2.11 Utilities

Utilities are programs designed to help the DBA with various administration tasks. As mentioned in Section 2.10, some utility programs operate at the external level of the system, and thus are effectively nothing more than special-purpose applications; some might not even be provided by the DBMS vendor, but rather by some third-party software supplier. Other utilities, however, operate directly at the internal level (in other words, they are really part of the server), and hence must be provided by the DBMS vendor.

Here are some typical examples of the kind of utilities that are frequently needed in practice:

- **Load** routines, to create the initial version of the database from one or more non-database files
- **Unload/reload** routines, to unload the database, or portions thereof, to backup storage for recovery purposes and to reload data from such backup copies (of course, the "reload utility" is basically identical to the load utility just discussed)
- **Reorganization** routines, to rearrange the data in the database for various reasons (usually having to do with performance)—e.g., to cluster data together in some particular way on the disk, or to reclaim space occupied by data that has become obsolete
- **Statistical** routines, to compute various performance statistics such as file sizes or data value distributions or I/O counts, etc.
- **Analysis** routines, to analyze the statistics just mentioned

The foregoing list represents just a small sample of the range of functions that utilities typically provide. A wealth of other possibilities exist.

2.12 Distributed Processing

To repeat from Section 2.10, the term "distributed processing" means that distinct machines can be connected together into a communications network such that a single data processing task can span several machines in the network. (The term "parallel processing" is also sometimes used with essentially the same meaning, except that the distinct machines tend to be physically close together in a "parallel" system and need not be so in a "distributed" system—e.g., they might be geographically dispersed in the latter case.) Communication between the various machines is handled by some kind of network management software—possibly an extension of the DC manager discussed in Section 2.9, possibly a separate software component.

Many levels or varieties of distributed processing are possible. As mentioned in Section 2.10, one simple case involves running the DBMS backend (the server) on one machine and the application frontends (the clients) on another. Refer to Fig. 2.5.

As mentioned at the end of Section 2.10, "client/server"—although strictly speak-

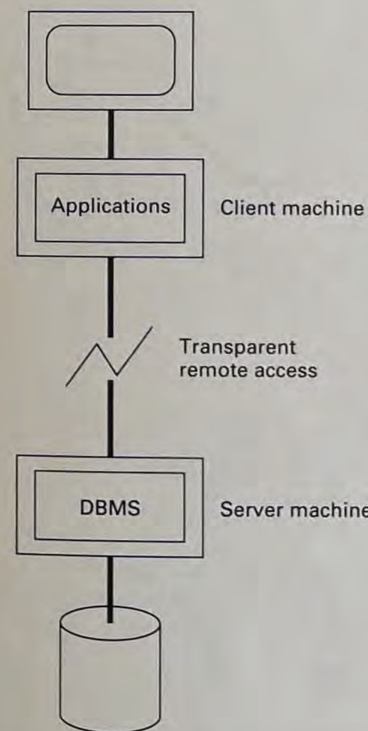

FIG. 2.5 Client and server running on different machines

Meta Platforms, Inc.
Exhibit 1024
Page 025

ing a purely architectural term—has come to be virtually synonymous with the arrangement illustrated in Fig. 2.5, in which client and server run on different machines. Indeed, there are many arguments in favor of such a scheme:

- The first is basically just the usual parallel processing argument—namely that many processors are now being applied to the overall task, and server (database) and client (application) processing are being done in parallel. Response time and throughput should thus be improved.
- Furthermore, the server machine might be a custom-built machine that is tailored to the DBMS function (a "database machine"), and might thus provide better DBMS performance.
- Likewise, the client machine might be a personal workstation, tailored to the needs of the end user and thus able to provide better interfaces, high availability, faster responses, and overall improved ease of use to the user.
- Several different client machines might be able (in fact, probably will be able) to access the same server machine. Thus, a single database might be shared across several distinct client systems (see Fig. 2.6).

In addition to the foregoing arguments, there is also the point that running the client(s) and the server on separate machines matches the way many enterprises actually

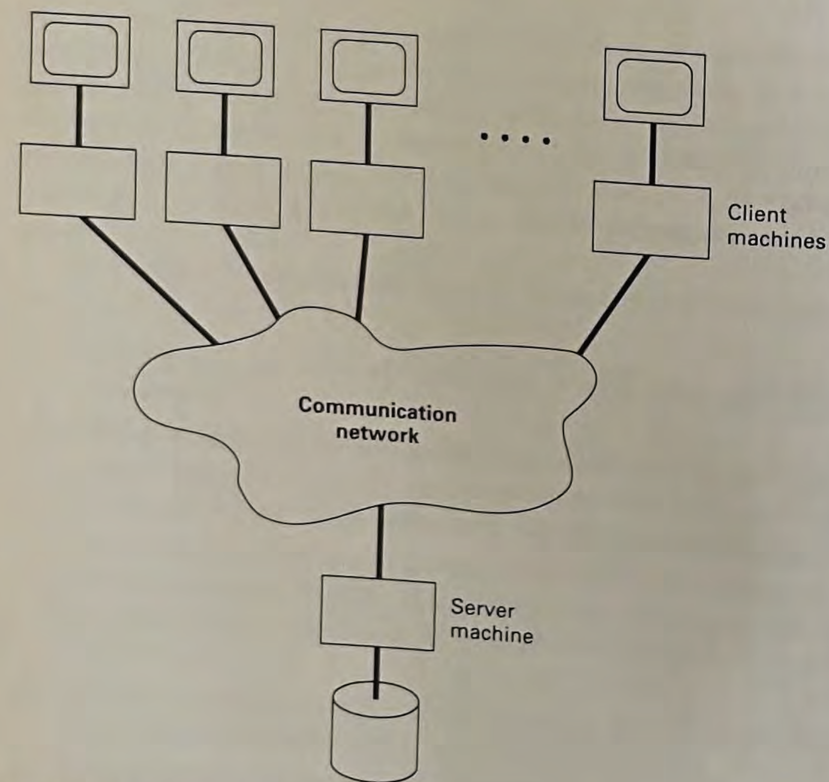

FIG. 2.6 One server, many clients

operate. It is quite common for a single enterprise—a bank, for example—to operate many computers, such that the data for one portion of the enterprise is stored on one computer and that for another portion is stored on another. It is also quite common for users on one computer to need at least occasional access to data stored on another. To pursue the banking example for a moment, it is very likely that users at one branch office will occasionally need access to data stored at another. Note, therefore, that the client machines might have stored data of their own, and the server machine might have applications of its own. In general, therefore, each machine will act as a server for some users and a client for others (see Fig. 2.7); in other words, each machine will support *an entire database system* (in the sense of earlier sections of this chapter).

The final point is that a single client machine might be able to access several dif-

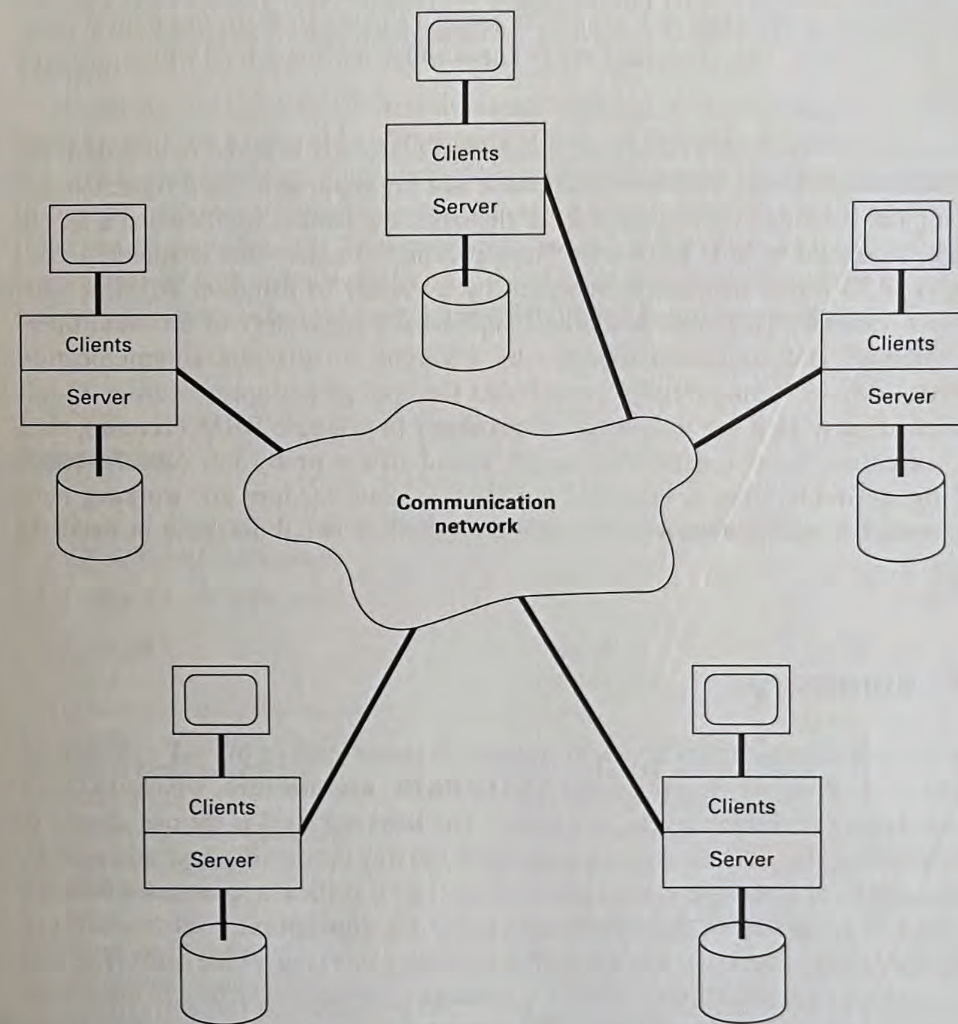

FIG. 2.7 Each machine is both client and server

Meta Platforms, Inc.
Exhibit 1024
Page 026

ferent server machines (the converse of the case illustrated in Fig. 2.6). This capability is desirable because, as mentioned above, enterprises do typically operate in such a manner that their total data collection is not stored on one single machine but rather is spread across many distinct machines, and applications will sometimes need the ability to access data from more than one machine. Such access can basically be provided in two ways:

1. A given client might be able to access any number of servers, but only one at a time (i.e., each individual database request must be directed to just one server). In such a system it is not possible, within a single request, to combine data from two or more different servers. Furthermore, the user in such a system has to know which particular machine holds which pieces of data.
2. The client might be able to access many servers simultaneously (i.e., a single database request might be able to combine data from several servers). In this case, the servers look to the client as if they were really a single server (from a logical point of view), and the user does not have to know which machines hold which pieces of data.

The second case here is an example of what is usually referred to as a **distributed database system**. Distributed database is a big topic in its own right; carried to its logical conclusion, full support for distributed database implies that a single application should be able to operate "transparently" on data that is spread across a variety of different databases, managed by a variety of different DBMSs, running on a variety of different machines, supported by a variety of different operating systems, and connected together by a variety of different communication networks—where "transparently" means that the application operates from a logical point of view as if the data were all managed by a single DBMS running on a single machine. Such a capability might sound like a pretty tall order!—but it is highly desirable from a practical perspective, and vendors are working hard to make such systems a reality. We will discuss distributed database in detail in Chapter 21.

2.13 Summary

In this chapter we have taken a look at database systems from an overall architectural point of view. First, we described the **ANSI/SPARC architecture,** which divides a database system into three **levels**, as follows: The **internal** level is the one closest to physical storage (i.e., it is the one concerned with the way the data is physically stored); the **external** level is the one closest to the users (i.e., it is the one concerned with the way the data is viewed by individual users); and the **conceptual** level is a level of indirection between the other two (it provides a *community view* of the data). The data as perceived at each level is described by a **schema** (or several schemas, in the case of the external level). **Mappings** describe the correspondence between (a) a given exter-

nal schema and the conceptual schema, and (b) the conceptual schema and the internal schema.

Users—i.e., end users and application programmers, both of whom operate at the external level—interact with the data by means of a **data sublanguage,** which breaks down into at least two components, a **data definition language** (DDL) and a **data manipulation language** (DML). The data sublanguage is embedded in a **host language**. *Note:* The dividing lines between the host language and the data sublanguage, and between the DDL and the DML, are primarily conceptual in nature; ideally they should be "transparent to the user."

We also took a closer look at the functions of the **DBA** and the **DBMS**. Among other things, the DBA is responsible for creating the internal schema (**physical database design**); creating the conceptual schema (**logical** or **conceptual** database design), by contrast, is the responsibility of the *data* administrator. And the DBMS is responsible (among other things) for implementing DDL and DML requests from the user. The DBMS is also responsible for providing some kind of **data dictionary** function.

Database systems can also be conveniently thought of as consisting of a **server** (the DBMS itself) and a set of **clients** (the applications). Client and server can and often will run on separate machines, thus providing one simple kind of **distributed processing**. In general, each server can serve many clients, and each client can access many servers. If the system provides total "transparency"—meaning that each client can behave as if it were dealing with a single server on a single machine, regardless of the actual physical state of affairs—then we have a true **distributed database system**.

Exercises

2.1 Draw a diagram of the database system architecture presented in this chapter (the ANSI/SPARC architecture).

2.2 Define the following terms:

backend	frontend
client	host language
conceptual DDL, schema, view	load
conceptual/internal mapping	logical database design
data definition language	internal DDL, schema, view
data dictionary	physical database design
data manipulation language	planned request
data sublanguage	reorganization
DB/DC system	server
DC manager	storage structure definition
distributed database	unload/reload
distributed processing	unplanned request
external DDL, schema, view	user interface
external/conceptual mapping	utility

Meta Platforms, Inc.
Exhibit 1024
Page 027

2.3 Explain the sequence of steps involved in retrieving a particular external record occurrence.

2.4 List the major functions performed by the DBMS.

2.5 List the major functions performed by the DBA.

2.6 Distinguish between the DBMS and a file management system.

2.7 Give some examples of vendor-provided frontends or tools.

2.8 Give some examples of database utilities.

2.9 Examine any database system that might be available to you. Try to map that system to the ANSI/SPARC architecture as described in this chapter. Does it cleanly support the three levels of the architecture? How are the mappings between levels defined? What do the various DDLs (external, conceptual, internal) look like? What data sublanguage(s) does the system support? What host languages? Who performs the DBA function? Are there any security or integrity facilities? Is there a dictionary? What vendor-provided applications does the system support? What utilities? Is there a separate DC manager? Are there any distributed processing capabilities?

References and Bibliography

Some of the following references are beginning to show their age, but they are all still relevant to the concepts introduced in the present chapter.

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

2.2 Dionysios C. Tsichritzis and Anthony Klug (eds.). "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems." *Information Systems 3* (1978).

These two documents [2.1–2.2] are the Interim and Final Reports, respectively, of the so-called ANSI/SPARC Study Group. The ANSI/X3/SPARC Study Group on Data Base Management Systems (to give it its full title) was established in late 1972 by the Standards Planning and Requirements Committee (SPARC) of ANSI/X3, the American National Standards Committee on Computers and Information Processing. The objectives of the Study Group were to determine which areas, if any, of database technology were appropriate for standardization, and to produce a set of recommendations for action in each such area. In working to meet these objectives, the Study Group took the position that *interfaces* were the only aspect of a database system that could possibly be suitable for standardization, and accordingly defined a generalized database system architecture, or framework, that emphasized the role of such interfaces. The Final Report provides a detailed description of that architecture and of some of the 42 identified interfaces. The Interim Report is an earlier working document that is still of some interest; in some areas it provides additional detail.

2.3 J. J. van Griethuysen (ed.). *Concepts and Terminology for the Conceptual Schema and the Information Base.* International Organization for Standardization Document No. ISO/TC97/SC5-N695 (March 1982).

ISO/TC97/SC5/WG3 is an ISO Working Group whose objectives include "the definition of concepts for conceptual schema languages." This Working Group report includes an introduction to three competing candidates (more accurately, three *sets* of candidates) for an appropriate conceptual schema formalism, and applies each of the three to a common example involving the activities of a hypothetical Car Registration Authority. The three sets of contenders are (1) "entity-attribute-relationship" approaches, (2) "binary relationship" approaches, and (3) "interpreted predicate logic" approaches. The report also includes a discussion of the fundamental concepts underlying the notion of the conceptual schema, and offers some principles for implementation of a system that properly supports that notion. Heavy going in places, but an important document for anyone seriously interested in the conceptual level of the system.

2.4 Data Dictionary Systems Working Party of the British Computer Society. *Report.* Joint Issue: *Data Base* (ACM SIGBDP newsletter) *9,* No. 2; *SIGMOD Record* (ACM SIGMOD bulletin) *9,* No. 4 (December 1977).

An excellent description of the role of the data dictionary; includes a brief but good discussion of the conceptual schema.

2.5 P. P. Uhrowczik. "Data Dictionary/Directories." *IBM Sys. J. 12,* No. 4 (1973).

A good introduction to the basic concepts of a data dictionary system. An implementation is outlined based on IMS (IBM's original Data Dictionary product in fact conformed to that broad outline).

2.6 Paul Winsberg. *Dictionary Standards: ANSI, ISO, and IBM;* and *Industry Views of the Dictionary Standards Muddle.* Both in *InfoDB 3,* No. 4 (Winter 1988/89).

An excellent introduction to, and analysis of, the world of dictionary standards—including, in particular, the ANSI Information Resource Dictionary Systems (IRDS) standard.

2.7 William Kent. *Data and Reality.* Amsterdam, Netherlands: North-Holland/New York, NY: Elsevier Science (1978).

A stimulating and thought-provoking discussion of the nature of information, and in particular of the conceptual schema. The book can be regarded in large part as a compendium of real-world problems that (it is suggested) existing database formalisms—in particular, formalisms that are based on conventional record-like structures, which includes the relational approach—have difficulty in dealing with. Recommended.

Meta Platforms, Inc.
Exhibit 1024
Page 028

3 An Introduction to Relational Databases

3.1 Introduction

As explained in Chapter 1, the emphasis in this book is very much on the relational approach. In particular, the next part of the book, Part II, covers the theoretical foundations of that approach—namely, the relational model—in depth. The purpose of the present chapter is just to give a preliminary and very informal introduction to the material to be addressed in Part II (and to some extent in subsequent parts also), in order to pave the way for a better understanding of those later parts of the book. Most of the topics mentioned will be discussed again more formally, and in much more detail, in those later chapters.

3.2 Relational Systems

We begin by defining a **relational database management system** ("relational system" for short) as a system in which, at a minimum:

1. The data is perceived by the user as tables (and nothing but tables); and
2. The operators at the user's disposal—e.g., for data retrieval—are operators that generate new tables from old, and those operators include at least SELECT (also known as RESTRICT), PROJECT, and JOIN.

This definition, though still very brief, is slightly more specific than the one given in Chapter 1.

A sample relational database, the departments-and-employees database, is shown in Fig. 3.1. As you can see, that database is indeed "perceived as tables" (and the meaning of those tables is intended to be self-explanatory). Fig. 3.2 shows some sample SELECT, PROJECT, and JOIN operations against that database. Here are (very loose!) definitions of those operations:

- The **SELECT** operation (also known as **RESTRICT**) extracts specified rows from a table.

DEPT

DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M
D3	Research	5M

EMP

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

FIG. 3.1 The departments-and-employees database (sample values)

- The **PROJECT** operation extracts specified columns from a table.
- The **JOIN** operation joins together two tables on the basis of common values in a common column.

Of the three examples, the only one that seems to need any further explanation is the JOIN example. First of all, observe that the two tables DEPT and EMP do indeed have a common column, namely DEPT#, so they can be joined together on the basis of

SELECT (RESTRICT):
DEPTs where BUDGET > 8M

Result:

DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M

PROJECT:
DEPTs over DEPT#, BUDGET

Result:

DEPT#	BUDGET
D1	10M
D2	12M
D3	5M

JOIN:
DEPTs and EMPs over DEPT#

Result:

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Lopez	40K
D1	Marketing	10M	E2	Cheng	42K
D2	Development	12M	E3	Finzi	30K
D2	Development	12M	E4	Saito	35K

FIG. 3.2 SELECT, PROJECT, and JOIN (examples)

Meta Platforms, Inc.
Exhibit 1024
Page 029

common values in that column. That is, a given row from table DEPT will join to a given row in table EMP—to produce a new, wider row—if and only if the two rows in question have a common DEPT# value. For example, the DEPT and EMP rows

DEPT#	DNAME	BUDGET
D1	Marketing	10M

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K

(column names shown for explicitness) can be joined together to produce the result row

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Lopez	40K

because they have the same value, D1, in the common column. The set of all possible such joined rows constitutes the overall result. Observe that the common (DEPT#) value appears just once, not twice, in each result row. Observe too that since no EMP row has a DEPT# value of D3 (i.e., no employee is currently assigned to that department), no row for D3 appears in the result, even though there *is* a row for D3 in table DEPT.

One point that Fig. 3.2 clearly illustrates is that *the result of each of the three operations is another table*. This is the relational property of **closure,** and it is very important. Basically, because the output of any operation is the same kind of object as the input—they are all tables—so *the output from one operation can become input to another.* Thus it is possible (for example) to take a projection of a join, or a join of two restrictions, etc., etc. In other words, it is possible to write *nested expressions*—i.e., expressions in which the operands themselves are represented by expressions, instead of just simple table names. This fact in turn has numerous important consequences, as we will see later (both in this chapter and in many subsequent ones).

Note: When we say that the output from each operation is another table, it is very important to understand that we are talking *from a conceptual point of view*. We do not necessarily mean to imply that the system actually has to materialize the result of every individual operation in its entirety. For example, suppose we are trying to compute a restriction of a join. Then, as soon as a given row of the join is constructed, the system can immediately apply the restriction to that row to see whether it belongs in the final result, and immediately discard it if not. In other words, the intermediate result that is the output from the join might never exist as a fully materialized table in its own right at all. As a general rule, in fact, the system tries very hard *not* to materialize intermediate results in their entirety, for obvious performance reasons.

Another point that Fig. 3.2 also clearly illustrates is that the operations are all **set-at-a-time,** not row-at-a-time; that is, the operands and results are all entire tables, not just single rows, and tables contain *sets* of rows. For example, the JOIN in Fig. 3.2 operates on two tables of three and four rows respectively, and returns a result table of four rows. This **set processing capability** is a major distinguishing characteristic of relational systems (see further discussion in Section 3.6 below). By contrast, the operations in nonrelational systems are typically at the row- or record-at-a-time level.

Let us return to Fig. 3.1 for a moment. There are a few additional points to be made in connection with the sample database of that figure:

- First, note that the "relational system" definition requires only that the database be *perceived by the user* as tables. Tables are the **logical** structure in a relational system, not the physical structure. At the physical level, in fact, the system is free to use any or all of the usual storage structures—sequential files, indexing, hashing, pointer chains, compression, etc.—provided only that it can map those structures into tables at the logical level. Another way of saying the same thing is that tables represent an *abstraction* of the way the data is physically stored—an abstraction in which numerous storage-level details, such as stored record placement, stored record sequence, stored data encodings, stored record prefixes, stored access structures such as indexes, and so forth, are all *hidden from the user.*

  Incidentally, the term "logical structure" in the foregoing paragraph is intended to encompass both the conceptual and external levels, in ANSI/SPARC terms. The point is that—as explained in Chapter 2—the conceptual and external levels in a relational system will be relational, but the internal or physical level will not. In fact, relational theory as such has nothing to say about the internal level at all; it is, to repeat, concerned with how the database looks to the *user*.

- Second, relational databases like that of Fig. 3.1 satisfy a very nice property: *The entire information content of the database is represented in one and only one way, namely as explicit data values*. This method of representation (as explicit values in column positions in rows in tables) is the *only* method available in a relational database. In particular, there are no *pointers* connecting one table to another. For example, there is a connection between the D1 row of table DEPT and the E1 row of table EMP, because employee E1 works in department D1; but that connection is represented, not by a pointer, but by the appearance of the *value* D1 in the DEPT# position of the EMP row for E1. In nonrelational systems, by contrast, such information is typically represented by some kind of pointer that is explicitly visible to the user.

  Note: When we say there are no pointers in a relational database, we do not mean that there cannot be pointers *at the physical level*—on the contrary, there certainly can be pointers at that level, and indeed there certainly will be. But as already explained, all such physical storage details are concealed from the user in a relational system.

- Finally, note that *all data values are atomic* (or **scalar**). That is, at every row-and-column position in every table there is always exactly one data value, never a group of several values. Thus, for example, in table EMP (considering the DEPT# and EMP# columns only, and for clarity showing them in that left-to-right order), we have

DEPT#	EMP#
D1	E1
D1	E2
..	..

Meta Platforms, Inc.
Exhibit 1024
Page 030

instead of

DEPT#	EMP#
D1	E1,E2
..	..

Column EMP# in the second version of this table is an example of what is usually called a **repeating group**. A repeating group is a column, or combination of columns, that contains several data values in each row (different numbers of values in different rows, in general). *Relational databases do not allow repeating groups;* the second version of the table above would not be permitted in a relational system. (The reason for this apparent limitation is basically *simplicity*. See Chapters 4 and 19 for further discussion.)

We close this section by remarking that the definition given for "relational system" at the beginning of the section is only a *minimal* definition (it is taken from reference [3.1], and is essentially the definition that was current in the early 1980s). There is, of course, far more to a relational system than we can or need to describe in the present section. In particular, please note that the relational model consists of much more than just "tables plus SELECT, PROJECT, and JOIN." See Section 3.4.

3.3 A Note on Terminology

If it is true that a relational database is basically just a database in which the data is perceived as tables—and of course it *is* true—then a good question to ask is: Why exactly do we call such a database relational anyway? The answer is simple: "Relation" is just a mathematical term for a table (to be precise, a table of a certain specific kind—details to be discussed in Chapter 4). Thus, for example, we can say that the departments-and-employees database of Fig. 3.1 contains two *relations.*

Now, in informal contexts it is usual to treat the terms "relation" and "table" as if they were synonymous; indeed, the term "table" is used much more frequently than the term "relation" in such contexts. But it is worth taking a moment to understand why the latter term was introduced in the first place. Briefly, the explanation is as follows.

- As already indicated, relational systems are based on what is called *the relational model of data*. The relational model, in turn, is an abstract theory of data that is based on certain aspects of mathematics (principally set theory and predicate logic).
- The principles of the relational model were originally laid down in 1969–70 by Dr. E. F. Codd, at that time a researcher in IBM. It was late in 1968 that Codd, a mathematician by training, first realized that the discipline of mathematics could be used to inject some solid principles and rigor into a field—database management—that, prior to that time, was all too deficient in any such qualities. Codd's ideas were first widely disseminated in a now classic paper, "A Relational Model of Data for Large Shared Data Banks" (see reference [4.1] in Chapter 4).

- Since that time, those ideas—by now almost universally accepted—have had a wide-ranging influence on just about every aspect of database technology, and indeed on other fields as well, such as the fields of artificial intelligence, natural language processing, and hardware system design.

Now, the relational model as originally formulated by Codd very deliberately made use of certain terms, such as the term "relation" itself, that were not familiar in IT circles at that time, even though the concepts in some cases were. The trouble was, many of the more familiar terms were very *fuzzy*—they lacked the precision necessary to a formal theory of the kind that Codd was proposing.

- *Example:* Consider the term "record." At different times that single term can mean either a record *occurrence* or a record *type;* a *COBOL-style* record (which allows repeating groups) or a *flat* record (which does not); a *logical* record or a *physical* record; a *stored* record or a *virtual* record; and perhaps other things as well.

The formal relational model therefore does not use the term "record" at all; instead, it uses the term "tuple" (short for "*n*-tuple"), which was given a precise definition by Codd when he first introduced it. We do not give that definition here; for present purposes, it is sufficient to say that the term "tuple" corresponds approximately to the notion of a *flat record instance* (just as the term "relation" corresponds approximately to the notion of a table). When we move on (in Part II) to study the more formal aspects of relational systems, we will make use of the formal terminology, but in this chapter we are not trying to be very formal, and we will mostly stick to terms such as "table," "row," and "column" that are reasonably familiar.

3.4 The Relational Model

So what exactly is the relational model? A good way to characterize it is as follows: The relational model is *a way of looking at data*—that is, it is a prescription for a way of representing data (namely, by means of tables), and a prescription for a way of manipulating such a representation (namely, by means of operators such as JOIN). More precisely, the relational model is concerned with three aspects of data: data **structure,** data **integrity,** and data **manipulation**. The structural and manipulative aspects have already been illustrated; to illustrate the integrity aspect (*very* superficially, please note!), we consider the departments-and-employees database of Fig. 3.1 once again. In all likelihood, that database would be subject to numerous integrity rules; for example, employee salaries might have to be in the range 25K to 95K, department budgets might have to be in the range 1M to 15M, and so on. However, there are certain rules that the database *must* obey if it is to conform to the prescriptions of the relational model. To be specific:

1. Each row in table DEPT must include a unique DEPT# value; likewise, each row in table EMP must include a unique EMP# value.
2. Each DEPT# value in table EMP must exist as a DEPT# value in table DEPT (to reflect the fact that every employee must be assigned to an existing department).

SLEY
T
dition, 1993
1993
er, 0, 1992
991-1994,
991, 1992
989,
ngs,
1989
ns, 83
375, 1994
ted and ms, 1994
iented and ms,
s:
1994
652, 1993
s, 1993
n, 3
992
d Edition, 1992
gement: 90
n, 989
sign, 4, 1989

Answers to Selected Exercises

20.1 (a) *unk*. (b) *true*. (c) *true*. (d) *unk* (note the counterintuitive nature of this one). (e) *false*. (f) *false* (note that IS_UNK never returns *unk*). (g) *false*. (h) *true*.

20.2 (a) *unk*. (b) *unk*. (c) *true*. (d) *false*. (e) *unk*. (f) *true*. (g) *false*.

20.3 Because "IS_UNK(x)" returns *true* if and only if "$x\ \theta\ y$" returns *unk* (for arbitrary θ and arbitrary y), and it returns *false* if and only if "$x = y$ OR $x \neq y$" returns *true* (for arbitrary nonUNK y).

20.4 Because (e.g.) "MAYBE_RESTRICT *R* WHERE *p*" is the same as "*R* WHERE MAYBE(*p*)."

20.5 The four monadic operators can be defined as follows (*A* is the single operand):

```
A
NOT(A)
A OR NOT(A)
A AND NOT(A)
```

The 16 dyadic operators can be defined as follows (*A* and *B* are the two operands):

```
A OR NOT(A) OR B OR NOT(B)
A AND NOT(A) AND B AND NOT(B)
A
NOT(A)
B
NOT(B)
A OR B
A AND B
A OR NOT(B)
A AND NOT(B)
NOT(A) OR B
NOT(A) AND B
NOT(A) OR NOT(B)
NOT(A) AND NOT(B)
(NOT(A) OR B) AND (NOT(B) OR A)
(NOT(A) AND B) OR (NOT(B) AND A)
```

Incidentally, to see that we do not need both AND and OR, observe that, e.g.,

```
A OR B  ≡  NOT(NOT(A) AND NOT(B))
```

20.6 See the annotation to reference [20.12].

20.7 (c). For further discussion, see reference [20.8].

20.8 We briefly describe the representation used in IBM's DB2 product. In DB2, a column that can accept nulls is physically represented in the stored database by two columns, the data column itself and a hidden indicator column, one byte wide, that is stored as a prefix to the actual data column. An indicator column value of binary ones indicates that the corresponding data column value is to be ignored (i.e., taken as null); an indicator column value of binary zeros indicates that the corresponding data column value is to be taken as genuine. But the indicator column is always (of course) hidden from the user.

20.9 It seems to this writer that this relation does not have a well-formed predicate at all. The best we can say is something like the following: *The part with the specified part number EITHER has the specified color OR has no color*. But this statement is essentially meaningless! To say that each x either has a y or it doesn't is to say **nothing at all**. It certainly does not serve as a very meaningful "criterion for update acceptability." To this writer, therefore, it looks strongly as if (once again) the notion of nulls undermines the very foundations of the relational model.

21 Distributed Database and Client/Server Systems

21.1 Introduction

We touched on the subject of **distributed databases** at the end of Chapter 2, where we said that ". . . full support for distributed database implies that a single application should be able to operate transparently on data that is spread across a variety of different databases, managed by a variety of different DBMSs, running on a variety of different machines, supported by a variety of different operating systems, and connected together by a variety of different communication networks—where the term *transparently* means that the application operates from a logical point of view as if the data were all managed by a single DBMS running on a single machine." We are now in a position to examine these ideas in some detail. To be specific, in this chapter we will explain exactly what a distributed database is, why distributed databases are important, and what some of the unsolved technical problems are in the distributed database field.

Chapter 2 also briefly discussed **client/server** systems, which can be regarded as a particularly simple special case of distributed systems in general. We will consider client/server systems specifically in Section 21.6.

The overall plan of the chapter is explained at the end of the next section.

21.2 Some Preliminaries

We begin with a working definition (necessarily a little imprecise at this stage):

- A distributed database system consists of a collection of **sites,** connected together via some kind of communications network, in which
  1. Each site is a database system site in its own right, but
  2. The sites have agreed to work together so that a user at any site can access data anywhere in the network exactly as if the data were all stored at the user's own site.

It follows that the so-called "distributed database" is really a kind of *virtual* object, whose component parts are physically stored in a number of distinct "real" databases at

Meta Platforms, Inc.
Exhibit 1024
Page 032

a number of distinct sites (in effect, it is the logical union of those real databases). Fig. 21.1 provides an example.

Note that, to repeat, **each site is a database system site in its own right**. In other words, each site has its own local "real" databases, its own local users, its own local DBMS and transaction management software (including its own local locking, logging, recovery, etc., software), and its own local data communications manager (DC manager). In particular, a given user can perform operations on data at that user's own local site exactly as if that site did not participate in the distributed system at all (at least, this is an objective). The distributed database system can thus be regarded as a kind of **partnership** among the individual local DBMSs at the individual local sites; a new software component at each site—logically an extension of the local DBMS—provides the necessary partnership functions, and it is the combination of this new component

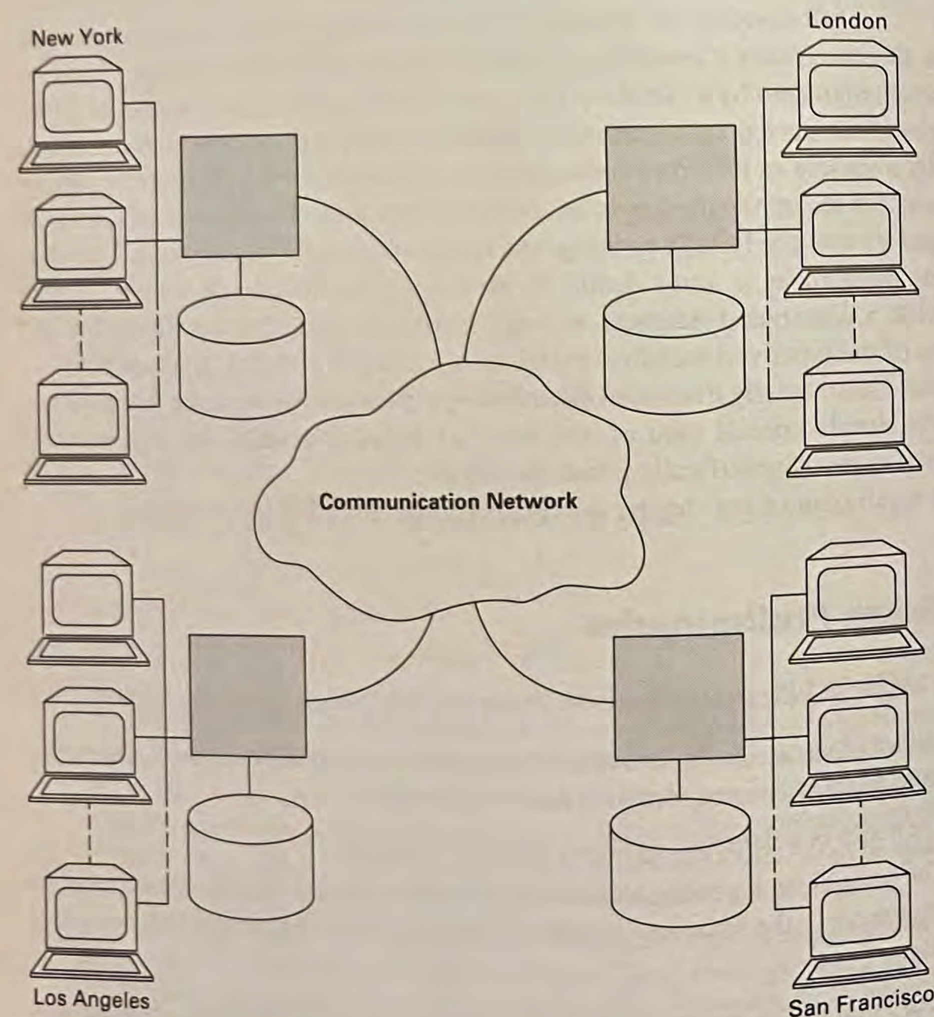

FIG. 21.1 A typical distributed database system

together with the existing DBMS that constitutes what is usually called the **distributed database management system** (sometimes abbreviated DDBMS).

Incidentally, it is common to assume that the component sites are physically dispersed—possibly in fact geographically dispersed also, as suggested by Fig. 21.1—although actually it is sufficient that they be dispersed *logically*. Two "sites" might even coexist on the same physical machine (especially during the period of initial system installation and testing). Indeed, the emphasis in distributed systems has shifted over the past few years: Whereas most of the original research tended to assume geographic distribution, most of the early commercial installations have involved *local* distribution instead, with (e.g.) several "sites" all in the same building and connected together by means of a local area network (LAN). From the database point of view, however, it makes little difference—essentially the same technical problems still have to be solved—and so we can reasonably regard Fig. 21.1 as representing a typical system for the purposes of this chapter.

Note: In order to simplify the exposition, we will assume until further notice that the system is *homogeneous,* in the sense that each site is running a copy of the same DBMS. We will refer to this as the **strict homogeneity** assumption. We will explore the possibility of relaxing this assumption in Section 21.5.

Advantages

Why are distributed databases desirable? The basic answer to this question is that enterprises normally *are* distributed already, at least logically (into divisions, departments, workgroups, etc.), and very likely physically too (into plants, factories, laboratories, etc.)—from which it follows that data normally is distributed already as well, because each organizational unit within the enterprise will necessarily maintain data that is relevant to its own operation. Thus, a distributed system enables the structure of the database to mirror the structure of the enterprise: Local data can be kept locally, where it most logically belongs, while at the same time remote data can be accessed when necessary.

An example will clarify the foregoing. Consider Fig. 21.1 once again. For simplicity, suppose there are only two sites, Los Angeles and San Francisco, and suppose the system is a banking system, with account data for Los Angeles accounts stored in Los Angeles and account data for San Francisco accounts stored in San Francisco. Then the advantages are surely obvious: The distributed arrangement combines **efficiency of processing** (the data is stored close to the point where it is most frequently used) with **increased accessibility** (it is possible to access a Los Angeles account from San Francisco and *vice versa,* via the communications network).

Allowing the structure of the database to mirror the structure of the enterprise is (as just explained) probably the number one advantage of distributed systems. Numerous additional benefits do also accrue, of course, but we will defer discussion of such additional benefits to appropriate points later in the chapter. However, we should mention that there are some disadvantages too, of which the biggest is the fact that distributed systems are *complex,* at least from a technical point of view. Ideally, of course, that

Meta Platforms, Inc.
Exhibit 1024
Page 033

complexity should be the implementer's problem, not the user's, but it is likely—to be pragmatic—that some aspects of that complexity will show through to users, unless very careful precautions are taken.

Sample Systems

For purposes of subsequent reference, we briefly mention some of the better known distributed system implementations. First, prototypes. Out of numerous research systems, three of the best known are (a) *SDD-1,* which was built in the research division of Computer Corporation of America in the late 1970s and early 1980s [21.26]; (b) *R** (pronounced "R star"), a distributed version of the System R prototype, built at IBM Research in the early 1980s [21.30]; and (c) *Distributed INGRES,* a distributed version of the INGRES prototype, also built in the early 1980s at the University of California at Berkeley [21.28].

As for commercial implementations, most of today's relational products offer some kind of distributed database support (with varying degrees of functionality, of course). Some of the best known include (a) *INGRES/STAR,* from The ASK Group Inc.'s Ingres Division; (b) the *distributed database option* of ORACLE7, from Oracle Corporation; and (c) the *distributed data facility* of DB2, from IBM. *Note:* These two lists are obviously not meant to be exhaustive; rather, they are meant to identify certain systems that either have been or are being particularly influential for one reason or another, or else have some special intrinsic interest.

It is worth pointing out that all of the systems listed above, both prototypes and products, are relational. Indeed, there are several reasons why, for a distributed system to be successful, that system *must* be relational; relational technology is a prerequisite to (effective) distributed technology [21.14]. We will see some of the reasons for this state of affairs as we proceed through the chapter.

A Fundamental Principle

Now it is possible to state what might be regarded as **the fundamental principle of distributed database** [21.13]:

- *To the user, a distributed system should look exactly like a NONdistributed system.*

In other words, users in a distributed system should behave exactly as if the system were *not* distributed. All of the problems of distributed systems are—or should be—internal or implementation-level problems, not external or user-level problems.

Note: The term "users" in the foregoing paragraph refers specifically to users (end users or application programmers) who are performing *data manipulation* operations. All data manipulation operations should remain logically unchanged. Data *definition* operations, by contrast, will require some extension in a distributed system—for example, so that a user at site *X* can specify that a given stored relation be divided into "fragments" that are to be stored at sites *Y* and *Z* (see the discussion of fragmentation in the next section).

The fundamental principle identified above leads to a number of subsidiary rules

or objectives*—actually twelve of them—which will be discussed in Section 21.3. For reference, we list those twelve objectives here:

1. Local autonomy
2. No reliance on a central site
3. Continuous operation
4. Location independence
5. Fragmentation independence
6. Replication independence
7. Distributed query processing
8. Distributed transaction management
9. Hardware independence
10. Operating system independence
11. Network independence
12. DBMS independence

These twelve objectives are *not* all independent of one another, nor are they necessarily exhaustive, nor are they all equally significant (different users will attach different degrees of importance to different objectives in different environments). However, they *are* useful as a basis for understanding distributed technology and as a framework for characterizing the functionality of specific distributed systems. We will therefore use them as an organizing principle for the rest of the chapter. Section 21.3 presents a brief discussion of each objective; Sections 21.4 and 21.5 then home in on certain specific issues in more detail. Section 21.6 (as previously mentioned) discusses client/server systems. Finally, Section 21.7 addresses the question of SQL support, and Section 21.8 offers a summary and a few concluding remarks.

One final introductory point: It is important to distinguish true, generalized, distributed database systems from systems that merely provide some kind of remote data access (which is all that client/server systems really do, incidentally). In a "remote data access" system, the user might be able to operate on data at a remote site, or even on data at several remote sites simultaneously, but *the seams show;* the user is definitely aware—to a greater or lesser extent—that the data is remote, and has to behave accordingly. In a true distributed database system, by contrast, the seams are *hidden.* (Much of the rest of this chapter is concerned with what it means in this context to say that the seams are hidden.) In what follows, we will use the term "distributed system" to refer specifically to a true, generalized, distributed database system, as opposed to a simple remote data access system (barring explicit statements to the contrary).

* "Rules" was the term used in the paper in which they were first introduced [21.13] (and the "fundamental principle" was referred to as *Rule Zero*). However, "objectives" is really a better term—"rules" sounds much too dogmatic. We will stay with the milder term "objectives" in the present chapter.

Meta Platforms, Inc.
Exhibit 1024
Page 034