CHAPTER 1

An Overview of Database Management

1.1 INTRODUCTION

A database system is basically just a *computerized record-keeping system*. The database itself can be regarded as a kind of electronic filing cabinet; that is, it is a repository or container for a collection of computerized data files. Users of the system can perform (or request the system to perform, rather) a variety of operations involving such files—for example:

- Adding new files to the database
- Inserting data into existing files
- Retrieving data from existing files
- Deleting data from existing files
- Changing data in existing files
- Removing existing files from the database

Fig. 1.1 shows a very small database containing just one file, called CELLAR, which in turn contains data concerning the contents of a wine cellar. Fig. 1.2 shows an example of a retrieval request against that database, together with the data returned by that request. (Throughout this book we show database requests, file names, and other such material in uppercase for clarity. In practice it is often more convenient to enter such material in lowercase. Most systems will accept both.) Fig. 1.3 gives examples, all more or less self-explanatory, of insert, delete, and change requests on the wine cellar database. Examples of adding and removing entire files are given in later chapters.

Several points arise immediately from Figs. 1.1–1.3:

1. First of all, the SELECT, INSERT, DELETE, and UPDATE requests (also called *statements*, *commands*, or *operators*) in Figs. 1.2 and 1.3 are all expressed in a language called SQL. Originally a proprietary language from IBM, SQL is now an international standard that is supported by just about every database product commercially available; in fact, the market is totally dominated by SQL products at the time of writing. Because of its commercial importance, therefore, Chapter 4 presents a general overview of the SQL standard, and most subsequent chapters include a section called "SQL Facilities" that describes those aspects of that standard that are pertinent to the topic of the chapter in question.

   By the way, the name SQL originally stood for *Structured Query Language* and was pronounced *sequel*. Now that it is a standard, however, the name is officially just a name—it is not supposed to be an abbreviation for anything at all—and it is officially pronounced *ess-cue-ell*. We will assume this latter pronunciation throughout this book.

2. Note from Fig. 1.3 that SQL uses the keyword UPDATE to mean "change" specifically. This fact can cause confusion, because the term *update* is also used to refer to the three operators INSERT, DELETE, and UPDATE considered as a group. We will distinguish between the two meanings in this book by using lowercase when the generic meaning is intended and uppercase to refer to the UPDATE operator specifically.

   Incidentally, you might have noticed that we have now used both the term *operator* and the term *operation*. Strictly speaking, there is a difference between the two (the *operation* is what is performed when the *operator* is invoked); in informal discussions, however, the terms tend to be used interchangeably.

3. In SQL, computerized files such as CELLAR in Fig. 1.1 are called tables (for obvious reasons); the rows of such a table can be thought of as the *records* of the file, and the columns can be thought of as the *fields*. In this book, we will use the terminology of files, records, and fields when we are talking about database systems in general (mostly just in the first two chapters); we will use the terminology of tables, rows, and columns when we are talking about SQL systems in particular. (And when we get to our more formal discussions in Chapter 3 and later parts of the book, we will meet yet another set of terms: *relations*, *tuples*, and *attributes* instead of tables, rows, and columns.)

BIN#	WINE	PRODUCER	YEAR	BOTTLES	READY
2	Chardonnay	Buena Vista	2001	1	2003
3	Chardonnay	Geyser Peak	2001	5	2003
6	Chardonnay	Simi	2000	4	2002
12	Joh. Riesling	Jekel	2002	1	2003
21	Fumé Blanc	Ch. St. Jean	2001	4	2003
22	Fumé Blanc	Robt. Mondavi	2000	2	2002
30	Gewürztraminer	Ch. St. Jean	2002	3	2003
43	Cab. Sauvignon	Windsor	1995	12	2004
45	Cab. Sauvignon	Geyser Peak	1998	12	2006
48	Cab. Sauvignon	Robt. Mondavi	1997	12	2008
50	Pinot Noir	Gary Farrell	2000	3	2003
51	Pinot Noir	Fetzer	1997	3	2004
52	Pinot Noir	Dehlinger	1999	2	2002
58	Merlot	Clos du Bois	1998	9	2004
64	Zinfandel	Cline	1998	9	2007
72	Zinfandel	Rafanelli	1999	2	2007

Fig. 1.1 The wine cellar database (file CELLAR)

```
Retrieval:
SELECT WINE, BIN#, PRODUCER
FROM   CELLAR
WHERE  READY = 2004 ;
```

Result (as shown on, e.g., a display screen):

WINE	BIN#	PRODUCER
Cab. Sauvignon	43	Windsor
Pinot Noir	51	Fetzer
Merlot	58	Clos du Bois

Fig. 1.2 Retrieval example

```
Inserting new data:
INSERT
INTO   CELLAR ( BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY )
VALUES ( 53, 'Pinot Noir', 'Saintsbury', 2001, 6, 2005 ) ;

Deleting existing data:
DELETE
FROM   CELLAR
WHERE  BIN# = 2 ;

Changing existing data:
UPDATE CELLAR
SET    BOTTLES = 4
WHERE  BIN# = 3 ;
```

Fig. 1.3 Insert, delete, and change examples

4. With respect to the CELLAR table, we have made a tacit assumption for simplicity that columns WINE and PRODUCER contain character-string data and all other columns contain integer data. In general, however, columns can contain data of arbitrary

complexity. For example, we might extend the CELLAR table to include additional columns as follows:

- LABEL (photograph of the wine bottle label)
- REVIEW (text of a review from some wine magazine)
- MAP (map showing where the wine comes from)
- NOTES (nudio recording containing our own tasting notes)

and many other things. For obvious reasons, the majority of examples in this book involve only very simple kinds of data, but do not lose sight of the fact that more exotic possibilities are always available. We will consider this question of column data types in more detail in later chapters (especially Chapters 5–6 and 26–27).

5. Column BIN# constitutes the primary key for table CELLAR (meaning, loosely, that no two CELLAR rows ever contain the same BIN# value). In figures like Fig. 1.1 we use *double underlining* to indicate primary key columns.

One last point to close this preliminary section: While a full understanding of this chapter and the next is necessary to a proper appreciation of the features and capabilities of a modern database system, it cannot be denied that the materinl is somewhat abstract and rather dry in places (also, it does tend to involve a large number of concepts and terms that might be new to you). In Chapters 3 and 4 you will find material that is much less abstract and hence more immediately understandable, perhaps. You might therefore prefer just to give these first two chapters a "once over lightly" reading for now, and to reread them more carefully later as they become more directly relevant to the topics at hand.

1.2 WHAT IS A DATABASE SYSTEM?

To repeat from the previous section, a database system is basically a computerized record-keeping system; in other words, it is a computerized system whose overall purpose is to store information and to allow users to retrieve and update that information on demand. The information in question can be anything that is of significance to the individual or orgnnization concerned—anything, in other words, that is needed to assist in the general process of running the business of that individual or organizntion.

Incidentally, please note that we treat the terms *data* and *information* as synonyms in this book. Some writers prefer to distinguish between the two, using *data* to refer to what is actually stored in the database and *information* to refer to the *meaning* of that data as understood by some user. The distinction is clearly important—so important that it seems preferable to make it explicit, where appropriate, instead of relying on a somewhat arbitrary differentiation between two essentially synonymous terms.

Fig. 1.4 is a simplified picture of a database system. As the figure shows, such a system involves four major components: data, hardware, software, and users. We consider these four components briefly here. Later we will discuss each in much more detail (except for the hardware component, details of which are mostly beyond the scope of this book).

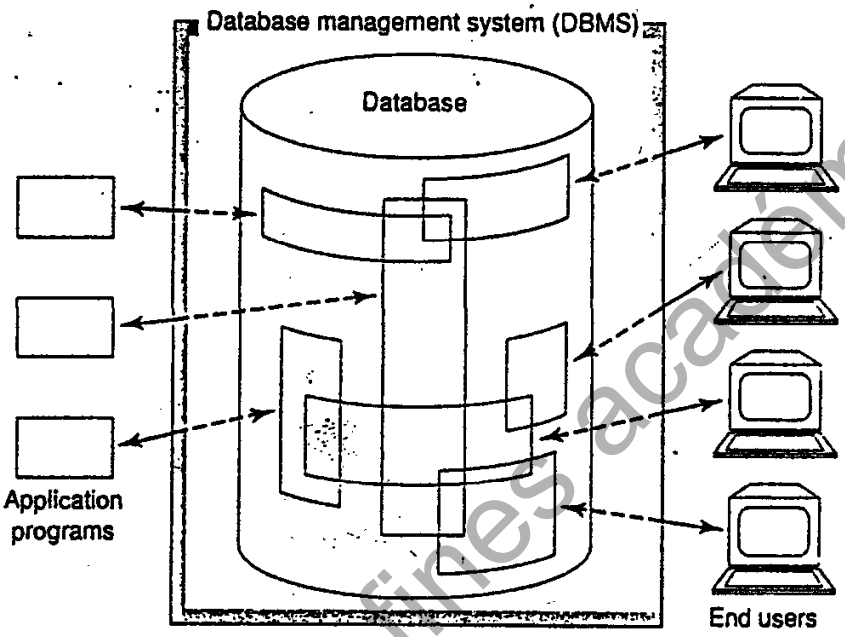

Fig. 1.4 Simplified picture of a database system

Data

Database systems are available on machines that range all the way from the smallest hand-held or personal computers to the largest mainframes or clusters of mainframes. Needless to say, the facilities provided by any given system are determined to some extent by the size and power of the underlying machine. In particular, systems on large machines ("large systems") tend to be *multi-user*: whereas those on smaller machines ("small systems") tend to be *single-user*. A single-user system is a system in which at most one user can access the database at any given time; a multi-user system is a system in which many users can access the database at the same time. As Fig. 1.4 suggests, we will normally assume the latter case in this book, for generality; in fact, however, the distinction is largely irrelevant so far as most users are concerned, because it is precisely an objective of multi-user systems in general to allow each user to behave as if he or she were working with a *single*-user system instead. The special problems of multi-user systems are primarily problems that are internal to the system, not ones that are visible to the user (see Part IV of this book, especially Chapter 16).

Now, it is convenient to assume for the sake of simplicity that the totality of data in the system is all stored in a single database, and we will usually make that assumption in this book, since it does not materially affect any of our other discussions. In practice, however, there might be good reasons, even in a small system, why the data should be split across several distinct databases. We will touch on some of those reasons later, in Chapter 2 and elsewhere.

In general, then, the data in the database—at least in a large system—will be both *integrated* and *shared.* As we will see in Section 1.4, these two aspects, data integration and data sharing, represent a major advantage of database systems in the "large" environment, and data integration, at least, can be significant in the "small" environment as well. Of course, there are many additional advantages also, to be discussed later, even in the small environment. But first let us explain what we mean by the terms *integrated* and *shared:*

- By integrated, we mean the database can be thought of as a unification of several otherwise distinct files, with any redundancy among those files partly or wholly eliminated. For example, a given database might contain both an EMPLOYEE file, giving employee names, addresses, departments, salaries, and so on, and an ENROLLMENT file, representing the enrollment of employees in training courses (refer to Fig. 1.5). Now suppose that, in order to carry out the process of training course administration, it is necessary to know the department for each enrolled student. Then there is clearly no need to include that information redundantly in the ENROLLMENT file, because it can always be discovered by referring to the EMPLOYEE file instead.
- By shared, we mean the database can be shared among different users, in the sense that different users can have access to the same data, possibly even at the same time ("concurrent access"). Such sharing, concurrent or otherwise, is partly a consequence of the fact that the database is integrated. In the example of Fig. 1.5, for instance, the department information in the EMPLOYEE file would typically be shared by users in the Personnel Department and users in the Education Department. (A database that is not shared in the foregoing sense is sometimes said to be "personal" or "application-specific.")

Another consequence of the foregoing facts—that the database is integrated and shared—is that any given user will typically be concerned only with some small portion of the total database; moreover, different users' portions will overlap in various ways. In other words, a given database will be perceived by different users in many different ways. In fact, even when two users share the same portion of the database, their views of that portion might differ considerably at a detailed level. This latter point is discussed more fully in Section 1.5 and in later chapters (especially Chapter 10).

We will have more to say regarding the nature of the data component of the system in Section 1.3.

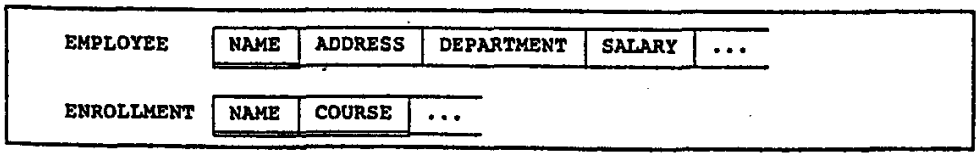

Fig. 1.5 The EMPLOYEE and ENROLLMENT files

Hardware

The hardware components of the system consist of:

- The secondary storage volumes—typically magnetic disks—that are used to hold the stored data, together with the associated I/O devices (disk drives, etc.), device controllers, I/O channels, and so forth
- The hardware processor(s) and associated main memory that are used to support the execution of the database system software (see the next subsection)

This book does not concern itself very much with the hardware aspects of the system, for the following reasons among others: First, those aspects form a major topic in their own right; second, the problems encountered in this area are not peculiar to database systems; and third, those problems have been very thoroughly investigated and documented elsewhere.

Software

Between the physical database itself—that is, the data as physically stored—and the users of the system is a layer of software, known variously as the database manager or database server or, most commonly, the database management system (DBMS). All requests for access to the database are handled by the DBMS: the facilities sketched in Section 1.1 for adding and removing files (or tables), retrieving data from and updating data in such files or tables, and so on, are all facilities provided by the DBMS. One general function provided by the DBMS is thus *the shielding of database users from hardware-level details* (much as programming language systems shield application programmers from hardware-level details). In other words, the DBMS provides users with a perception of the database that is elevated somewhat above the hardware level, and it supports user operations (such as the SQL operations discussed briefly in Section 1.1) that are expressed in terms of that higher-level perception. We will discuss this function, and other functions of the DBMS, in considerably more detail throughout the body of the book.

A couple of further points:

- The DBMS is easily the most important software component in the overall system, but it is not the only one. Others include utilities, application development tools, design aids, report writers, and (most significant) the *transaction manager* or *TP monitor*. See Chapters 2, 3, and especially 15 and 16 for further discussion of these components.
- The term *DBMS* is also used to refer generically to some particular product from some particular vendor—for example, IBM's DB2 Universal Database product. The term *DBMS instance* is then sometimes used to refer to the particular copy of such a product that happens to be running at some particular computer installation. As you will surely appreciate, sometimes it is necessary to distinguish carefully between these two concepts.

That said, you should be aware that people often use the term *database* when they really mean *DBMS* (in either of the foregoing senses). Here is a typical example: "Vendor *X*'s database outperformed vendor *Y*'s database by a factor of two to one." This usage is sloppy, and deprecated, but very, very common. (The problem is: If we call the DBMS the database, what do we call the database? *Caveat lector!*)

Users

We consider three broad (and somewhat overlapping) classes of users:

- First, there are **application programmers**, responsible for writing database application programs in some programming language, such as COBOL, PL/I, C++, Java, or some higher-level "fourth-generation" language (see Chapter 2). Such programs access the database by issuing the appropriate request (typically an SQL statement) to the DBMS. The programs themselves can be traditional batch applications, or they can be online applications, whose purpose is to allow an end user—see the next paragraph—to access the database interactively (e.g., from an online workstation or terminal or a personal computer). Most modern applications are of the online variety.
- Next, there are **end users**, who access the database interactively as just described. A given end user can access the database via one of the online applications mentioned in the previous paragraph, or he or she can use an interface provided as an integral part of the system. Such vendor-provided interfaces are also supported by means of online applications, of course, but those applications are built in, not user-written. Most systems include at least one such built-in application, called a query language processor, by which the user can issue database requests such as SELECT and INSERT to the DBMS interactively. SQL is a typical example of a database query language. (As an aside, we remark that the term *query language*, common though it is, is really a misnomer, inasmuch as the verb "to query" suggests *retrieval* only, whereas query languages usually—not always—provide update and other operators as well.)

  Most systems also provide additional built-in interfaces in which end users do not issue explicit database requests such as SELECT and INSERT at all, but instead operate by (e.g.) choosing items from a menu or filling in boxes on a form. Such menu- or forms-driven interfaces tend to be easier to use for people who do not have a formal training in IT (IT = information technology; the abbreviation IS, short for information systems, is also used with much the same meaning). By contrast, command-driven interfaces—that is, query languages—do tend to require a certain amount of professional IT expertise, though perhaps not much (obviously not as much as is needed to write an application program in a language like COBOL). Then again, a command-driven interface is likely to be more flexible than a menu- or forms-driven one, in that query languages typically include certain features that are not supported by those other interfaces.
- The third class of user, not illustrated in Fig. 1.4, is the **database administrator** or DBA. Discussion of the database administration function—and the associated (very

important) data administration function—is deferred to Section 1.4 and Chapter 2 (Section 2.7).

This completes our preliminary description of the major aspects of a database system. We now go on to discuss the ideas in more detail.

1.3 WHAT IS A DATABASE?

Persistent Data

It is customary to refer to the data in a database as "persistent" (though it might not actually persist for very long!). By *persistent*, we mean, intuitively, that database data differs in kind from other more ephemeral data, such as input data, output data, work queues, software control blocks, SQL statements, intermediate results, and more generally any data that is transient in nature. More precisely, we say that data in the database "persists" because, once it has been accepted by the DBMS for entry into the database in the first place, *it can subsequently be removed from the database only by some explicit request to the DBMS*, not as a mere side effect of (e.g.) some program completing execution. This notion of persistence thus allows us to give a slightly more precise definition for the term *database:*

- A database is a collection of persistent data that is used by the application systems of some given enterprise.

The term *enterprise* here is simply a convenient generic term for any reasonably self-contained commercial, scientific, technical, or other organization. An enterprise might be a single individual (with a small personal database), or a complete corporation or similar large body (with a large shared database), or anything in between. Here are some examples:

1. A manufacturing company
2. A bank
3. A hospital
4. A university
5. A government department

Any enterprise must necessarily maintain a lot of data about its operation. That data is the "persistent data" referred to in the definition. The enterprises just mentioned would typically include the following (respectively) among their persistent data:

1. Product data
2. Account data
3. Patient data
4. Student data
5. Planning data

Note: The first six editions of this book used the term *operational data* in place of *persistent data.* That earlier term reflected the original emphasis in database systems on operational or production applications—that is, routine, highly repetitive applications that were executed over and over again to support the day-to-day operation of the enterprise (for example, an application to support the deposit or withdrawal of cash in a banking system). The term ***online transaction processing*** (OLTP) has come to be used to refer to this kind of environment. However, databases are now increasingly used for other kinds of applications as well—that is, decision support applications—and the term *operational data* is thus no longer entirely appropriate. Indeed, enterprises nowadays frequently maintain two separate databases, one containing operational data and one, often called the data warehouse, containing decision support data. The data warehouse often includes *summary information* (e.g., totals, averages), where the summary information in question is extracted from the operational database on a periodic basis—say once a day or once a week. See Chapter 22 for an in-depth treatment of decision support databases and applications.

Entities and Relationships

We now consider the example of a manufacturing company ("KnowWare Inc.") in a little more detail. Such an enterprise will typically wish to record information about the *projects* it has on hand; the *parts* that are used in those projects; the *suppliers* who are under contract to supply those parts; the *warehouses* in which those parts are stored; the *employees* who work on those projects; and so on. Projects, parts, suppliers, and so on, thus constitute the basic entities about which KnowWare Inc. needs to record information (the term *entity* is commonly used in database circles to mean any distinguishable object that is to be represented in the database). Refer to Fig. 1.6.

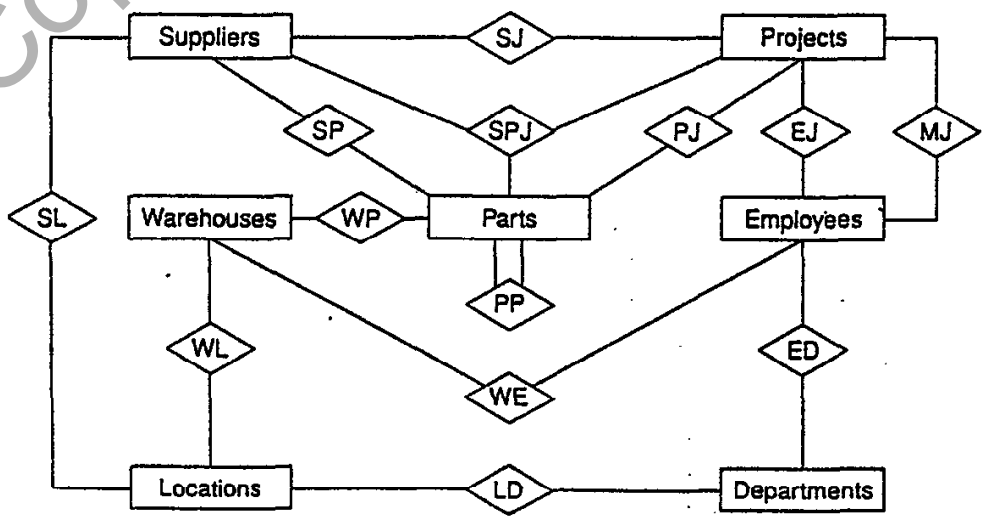

Fig. 1.6 Entity/relationship (E/R) diagram for KnowWare Inc.

In addition to the basic entities themselves (suppliers, parts, and so on, in the example), there will also be relationships linking those basic entities together. Such relationships are represented by diamonds and connecting lines in Fig. 1.6. For example, there is a relationship ("SP" or *shipments*) between suppliers and parts: Each supplier supplies certain parts, and conversely each part is supplied by certain suppliers (more accurately, each supplier supplies certain *kinds* of parts, each *kind* of part is supplied by certain suppliers). Similarly, parts are used in projects, and conversely projects use parts (relationship PJ); parts are stored in warehouses, and warehouses store parts (relationship WP); and so on. Note that these relationships are all *bidirectional*—that is, they can be traversed in either direction. For example, relationship SP between suppliers and parts can be used to answer both of the following queries:

- Given a supplier, get the parts supplied by that supplier.
- Given a part, get the suppliers who supply that part.

The significant point about this relationship (and all of the others illustrated in the figure) is that *they are just as much a part of the data as are the basic entities.* They must therefore be represented in the database, just like the basic entities.[1]

We note in passing that Fig. 1.6 is an example of what is called (for obvious reasons) an entity/relationship diagram (E/R diagram for short). We will consider such diagrams in detail in Chapter 14.

Fig. 1.6 also illustrates a number of other important points:

1. Although most of the relationships in that figure involve two entity types—that is, they are *binary* relationships—it is by no means the case that all relationships are binary in this sense. In the example there is one relationship ("SPJ") involving three entity types (suppliers, parts, and projects): a *ternary* relationship. The intended interpretation is that certain suppliers supply certain parts to certain projects. Note carefully that this ternary relationship ("suppliers supply parts to projects") is *not* equivalent, in general, to the combination of the three binary relationships "suppliers supply parts," "parts are used in projects," and "projects are supplied by suppliers." For example, the statement[2] that

   a. Smith supplies monkey wrenches to the Manhattan project

   tells us *more* than the following three statements do:

   b. Smith supplies monkey wrenches

   c. Monkey wrenches are used in the Manhattan project

   d. The Manhattan project is supplied by Smith

[1] In a relational database specifically (see Section 1.6), the basic entities and the relationships connecting them are both represented by means of *relations*, or in other words by tables like the one shown in Fig. 1.1, loosely speaking. Note carefully, therefore, that the term *relationship* as used in the present section and the term *relation* as used in the context of relational databases do not mean the same thing.

[2] The term *statement* is unfortunately used in the database world to mean two rather different things: It can be used, as here, to mean an *assertion of fact*, or what logicians call a *proposition* (see the subsection "Data and Data Models" later in this section); it can also be used, as we already know from earlier discussions, as a synonym for *command*, as in the expression "SQL statement."

—we cannot (validly!) infer *a* knowing only *b*, *c*, and *d*. More precisely, if we know *b*, *c*, and *d*, then we might be able to infer that Smith supplies monkey wrenches to *some* project (say project *Jz*), that *some* supplier (say supplier *Sx*) supplies monkey wrenches to the Manhattan project, and that Smith supplies *some* part (say part *Py*) to the Manhattan project—but we cannot validly infer that *Sx* is Smith or *Py* is monkey wrenches or *Jz* is the Manhattan project. False inferences such as these are examples of what is sometimes called the connection trap.

2. The figure also shows one relationship (PP) involving just *one* entity type (parts). The relationship here is that certain parts include other parts as immediate components (the so-called bill-of-materials relationship); for example, a screw is a component of a hinge assembly, which is also a part and might in turn be a component of some higher-level part such as a lid. Note that this relationship is still binary; it is just that the two entity types involved, parts and parts, happen to be one and the same.
3. In general, a given set of entity types might be involved in any number of distinct relationships. In the example in Fig. 1.6, there are two distinct relationships involving projects and employees: One (EJ) represents the fact that employees are assigned to projects; the other (MJ) represents the fact that employees manage projects.

We now observe that *a relationship can be regarded as an entity in its own right.* If we take as our definition of entity "any object about which we wish to record information," then a relationship certainly fits the definition. For instance, "part P4 is stored in warehouse W8" is an entity about which we might well wish to record information—for example, the corresponding quantity. Moreover, there are definite advantages (beyond the scope of the present chapter) to be obtained by not making any unnecessary distinctions between entities and relationships. In this book, therefore, we will tend to treat relationships as just a special kind of entity.

Properties

As just indicated, an entity is any object about which we wish to record information. It follows that entities (relationships included) can be regarded as having properties, corresponding to the information we wish to record about them. For example, suppliers have *locations;* parts have *weights;* projects have *priorities;* assignments (of employees to projects) have *start dates;* and so on. Such properties must therefore be represented in the database. For example, an SQL database might include a table called S representing suppliers, and that table might include a column called CITY representing supplier locations.

Properties in general can be as simple or as complex as we please. For example, the "supplier location" property is presumably quite simple, consisting as it does of just a city name, and can be represented in the database by a simple character string. By contrast, a warehouse might have a "floor plan" property, and that property might be quite complex, consisting perhaps of an entire architectural drawing and associated descriptive text. As noted in Section 1.1, in other words, the kinds of data we might want to keep in (for example) columns of SQL tables can be arbitrarily complex. As also noted in that same section, we will return to this topic later (principally in Chapters 5–6 and 26–27); until

then, we will mostly assume, where it makes any difference, that properties are "simple" and can be represented by "simple" data types. Examples of such "simple" types include numbers, character strings, dates, times, and so forth.

Data and Data Models

There is another (and important) way of thinking about what data and databases really are. The word *data* derives from the Latin for "to give"; thus, data is really *given facts*, from which additional facts can be inferred. (Inferring additional facts from given facts is exactly what the DBMS does when it responds to a user query.) A "given fact" in turn corresponds to what logicians call a *true proposition*; for example, the statement "Supplier S1 is located in London" might be such a true proposition. (A proposition in logic is something that is either true or false, unequivocally. For instance, "William Shakespeare wrote *Pride and Prejudice*" is a proposition—a false one, as it happens.) It follows that a database is really a collection of true propositions.

Now, we have already said that SQL products have come to dominate the marketplace. One reason for this state of affairs is that SQL products are based on a formal theory called the relational model of data, and that theory in turn supports the foregoing interpretation of data and databases very directly—almost trivially, in fact. To be specific, in the relational model:

1. Data is represented by means of rows in tables,[3] and such rows can be directly interpreted as true propositions. For example, the row for BIN# 72 in Fig. 1.1 can be interpreted in an obvious way as the following true proposition:

   Bin number 72 contains two bottles of 1999 Rafanelli Zinfandel, which will be ready to drink in 2007

2. Operators are provided for operating on rows in tables, and those operators directly support the process of inferring additional true propositions from the given ones. As a simple example, the relational *project* operator (see Section 1.6) allows us to infer, from the true proposition just quoted, the following additional true proposition among others:

   Some bottles of Zinfandel will be ready to drink in 2007

   (More precisely: "Some bottles of Zinfandel in some bin, produced by some producer in some year, will be ready to drink in 2007.")

The relational model is not the only data model, however; others do exist (see Section 1.6)—though most of them differ from the relational model in being *ad hoc* to a degree, instead of being firmly based as the relational model is on formal logic. Be that as it may, the question arises: What in general *is* a data model? Following reference [1.1] (but paraphrasing considerably), we can define the concept thus:

- A data model is an abstract, self-contained, logical definition of the objects, operators, and so forth, that together constitute the *abstract machine* with which users

[3] More precisely, by *tuples* in *relations* (see Chapter 3).

interact. The objects allow us to model the *structure* of data. The operators allow us to model its *behavior.*

We can then draw a useful (and very important!) distinction between the model and its *implementation:*

- An implementation of a given data model is a physical realization on a real machine of the components of the abstract machine that together constitute that model.

In a nutshell: The model is what users have to know about; the implementation is what users do not have to know about.

As you can see from the foregoing, the distinction between model and implementation is really just a special case—a very important special case—of the familiar distinction between *logical* and *physical.* Sadly, however, many of today's database systems, even ones that profess to be relational, do not make these distinctions as clearly as they should. Indeed, there seems to be a fairly widespread lack of understanding of these distinctions and the importance of making them. As a consequence, there is all too often a gap between database *principles* (the way database systems ought to be) and database *practice* (the way they actually are). In this book we are concerned primarily with principles, but it is only fair to warn you that you might therefore be in for a few surprises, mostly of an unpleasant nature, if and when you start using a commercial product.

In closing this section, we should mention the fact that *data model* is another term that is used in the literature with two quite different meanings. The first meaning is as already explained. The second is as follows: A data model (second sense) is *a model of the persistent data of some particular enterprise* (e.g., the manufacturing company KnowWare Inc. discussed earlier in this section). The difference between the two meanings can be characterized thus:

- A data model in the first sense is like a *programming language*—albeit one that is somewhat abstract—whose constructs can be used to solve a wide variety of specific problems, but in and of themselves have no direct connection with any such specific problem.
- A data model in the second sense is like a *specific program* written in that language. In other words, a data model in the second sense takes the facilities provided by some model in the first sense and applies them to some specific problem. It can be regarded as *a specific application* of some model in the first sense.

In this book, the term *data model* will be used only in the first sense from this point forward, barring explicit statements to the contrary.

1.4 WHY DATABASE?

Why use a database system? What are the advantages? To some extent the answer to these questions depends on whether the system in question is single- or multi-user (or rather, to be more accurate, there are numerous *additional* advantages in the multi-user case). We consider the single-user case first.

Refer back to the wine cellar example once again (Fig. 1.1), which we can regard as illustrative of the single-user case. Now, that particular database is so small and so simple that the advantages might not be all that obvious. But imagine a similar database for a large restaurant, with a stock of perhaps thousands of bottles and very frequent changes to that stock; or think of a liquor store, with again a very large stock and high turnover on that stock. The advantages of a database system over traditional, paper-based methods of record-keeping are perhaps easier to see in these cases. Here are some of them:

- *Compactness:* There is no need for possibly voluminous paper files.
- *Speed:* The machine can retrieve and update data far faster than a human can. In particular, *ad hoc*, spur-of-the-moment queries (e.g., "Do we have more Zinfandel than Pinot Noir?") can be answered quickly without any need for time-consuming manual or visual searches.
- *Less drudgery:* Much of the sheer tedium of maintaining files by hand is eliminated. Mechanical tasks are always better done by machines.
- *Currency:* Accurate, up-to-date information is available on demand at any time.
- *Protection:* The data can be better protected against unintentional loss and unlawful access.

The foregoing benefits apply with even more force in a multi-user environment, where the database is likely to be much larger and more complex than in the single-user case. However, there is one overriding additional advantage in such an environment: *The database system provides the enterprise with centralized control of its data* (which, as you should realize by now, is one of its most valuable assets). Such a situation contrasts sharply with that found in an enterprise without a database system, where typically each application has its own private files—quite often its own private tapes and disks, too—so that the data is widely dispersed and difficult to control in any systematic way.

Data Administration and Database Administration

We elaborate briefly on this concept of centralized control. The concept implies that there will be some identifiable person in the enterprise who has this central responsibility for the data, and that person is the **data administrator** (DA for short) mentioned briefly at the end of Section 1.2. Given that, to repeat, the data is one of the enterprise's most valuable assets, it is imperative that there should be some person who understands that data, and the needs of the enterprise with respect to that data, *at a senior management level*. The data administrator is that person. Thus, it is the data administrator's job to decide what data should be stored in the database in the first place, and to establish policies for maintaining and dealing with that data once it has been stored. An example of such a policy might be one that dictates who can perform what operations on what data in what circumstances—in other words, a *data security* policy (see the next subsection).

Note carefully that the data administrator is a manager, not a technician (although he or she certainly does need to have some broad appreciation of the capabilities of database systems at a technical level). The *technical* person responsible for implementing the data

administrator's decisions is the **data*base* administrator** (DBA for short). The DBA, unlike the data administrator, is thus an information technology (IT) professional. The job of the DBA is to create the actual database and to put in place the technical controls needed to enforce the various policy decisions made by the data administrator. The DBA is also responsible for ensuring that the system operates with adequate performance and for providing a variety of other technical services. The DBA will typically have a staff of system programmers and other technical assistants (i.e., the DBA function will typically be performed in practice by a team of people, not just by one person); for simplicity, however, it is convenient to assume that the DBA is indeed a single individual. We will discuss the DBA function in more detail in Chapter 2.

Benefits of the Database Approach

In this subsection we identify some more specific advantages that accrue from the foregoing notion of centralized control.

- *The data can be shared.*

  We discussed this point in Section 1.2, but for completeness we mention it again here. Sharing means not only that existing applications can share the data in the database, but also that new applications can be developed to operate against that same data. In other words, it might be possible to satisfy the data requirements of new applications without having to add any new data to the database.

- *Redundancy can be reduced.*

  In nondatabase systems each application has its own private files. This fact can often lead to considerable rednndancy in stored data, with resultant waste in storage space. For example, a personnel application and an education records application might both own a file that includes department information for employees. As suggested in Section 1.2, however, those two files can be integrated, and the redundancy eliminated, as long as the data administrator is aware of the data requirements for both applications—that is, as long as the enterprise has the necessary overall control.

  Note: We do not mean to say that *all* redundancy can or necessarily should be eliminated. Sometimes there are sound business or technical reasons for maintaining several distinct copies of the same data. However, we do mean that any such redundancy should be carefully controlled; that is, the DBMS should be aware of it, if it exists, and should assume responsibility for "propagating updates" (see the point immediately following).

- *Inconsistency can be avoided (to some extent).*

  This is really a corollary of the previous point. Suppose a given fact about the real world—say the fact that employee E3 works in department D8—is represented by two distinct entries in the database. Suppose also that the DBMS is not aware of this duplication (i.e., the redundancy is not controlled). Then there will necessarily be occasions on which the two entries will not agree: namely, when one of the two has been updated and the other not. At such times the database is said to be *inconsistent*. Clearly, a data-

base that is in an inconsistent state is capable of supplying incorrect or contradictory information to its users.

Of course, if the given fact is represented by a single entry (i.e., if the redundancy is removed), then such an inconsistency cannot occur. Alternatively, if the redundancy is not removed but is *controlled* (by making it known to the DBMS), then the DBMS can guarantee that the database is never inconsistent *as seen by the user*, by ensuring that any change made to either of the two entries is automatically applied to the other one as well. This process is known as propagating updates.

- *Transaction support can be provided.*

  A transaction is a logical unit of work (more precisely, a logical unit of *database* work), typically involving several database operations—in particular, several update operations. The standard example involves the transfer of a cash amount from one account *A* to another account *B*. Clearly two updates are required here, one to withdraw the cash from account *A* and the other to deposit it to account *B*. If the user has made the two updates part of the same transaction, then the system can effectively guarantee that either both of them are done or neither is—even if, for example, the system fails (say because of a power outage) halfway through the process.

  Note: The transaction *atomicity* feature just illustrated is not the only benefit of transaction support, but unlike some of the others it is one that applies, at least in principle, even in the single-user case. (On the other hand, single-user systems often do not provide any transaction support at all but simply leave the problem to the user.) A full description of all of the various advantages of transaction support and how they can be achieved appears in Chapters 15 and 16.

- *Integrity can be maintained.*

  The problem of integrity is the problem of ensuring (as far as possible) that the data in the database is correct. Inconsistency between two entries that purport to represent the same fact is an example of lack of integrity (see the discussion of this point earlier in this subsection); of course, this particular problem can arise only if redundancy exists in the stored data. Even if there is no redundancy, however, the database might still contain incorrect information. For example, an employee might be shown as having worked 400 hours in the week instead of 40, or as belonging to a department that does not exist. Centralized control of the database can help in avoiding such problems—insofar as they can be avoided—by permitting the data administrator to define, and the DBA to implement, integrity constraints to be checked when update operations are performed.

  It is worth pointing out that data integrity is even more important in a database system than it is in a "private files" environment, precisely because the data is shared. For without appropriate controls it would be possible for one user to update the database incorrectly, thereby generating bad data and so "infecting" other innocent users of that data. It should also be mentioned that most database products are still quite weak in their support for integrity constraints (though there have been some recent improvements in this area). This fact is unfortunate, given that (as we will see in Chapter 9) integrity constraints are both fundamental and crucially important—much more so than is usually realized, in fact.

- *Security can be enforced.*

  Having complete jurisdiction over the database, the DBA (under appropriate direction from the data administrator) can ensure that the only means of access to the database is through the proper channels, and hence can define security constraints to be checked whenever access is attempted to sensitive data. Different constraints can be established for each type of access (retrieve, insert, delete, etc.) to each piece of information in the database. Note, however, that without such constraints the security of the data might actually be more at risk than in a traditional (dispersed) filing system: that is, the centralized nature of a database system in a sense *requires* that a good security system be in place also.

- *Conflicting requirements can be balanced.*

  Knowing the overall requirements of the enterprise, as opposed to the requirements of individual users, the DBA (again under the data administrator's direction) can so structure the system as to provide an overall service that is "best for the enterprise." For example, a physical representation can be chosen for the data in storage that gives fast access for the most important applications (possibly at the cost of slower access for certain other applications).

- *Standards can be enforced.*

  With central control of the database, the DBA (under the direction of the data administrator once again) can ensure that all applicable standards are observed in the representation of the data, including any or all of the following: departmental, installation, corporate, industry, national, and international standards. Standardizing data representation is particularly desirable as an aid to *data interchange*, or movement of data between systems (this consideration is becoming particularly important with the advent of distributed systems—see Chapters 2, 21, and 27). Likewise, data naming and documentation standards are also very desirable as an aid to data sharing and understandability.

Now, most of the advantages listed so far are probably fairly obvious. However, one further point—which might not be as obvious, although it is in fact implied by several of the others—needs to be added to the list: *the provision of data independence*. (Strictly speaking, this is an *objective* for database systems, rather than an advantage as such.) The concept of data independence is so important that we devote a separate section to it.

1.5 DATA INDEPENDENCE

We begin by observing that there are two kinds of data independence, physical and logical [1.3, 1.4]; for the time being, however, we will concern ourselves with the physical kind only. Until further notice, therefore, the unqualified term *data independence* should be understood to mean physical data independence specifically. (We should perhaps add that the term *data independence* is not very apt—it does not capture very well the essence of what is really going on—but it is the term traditionally used, and we will stay with it in this book.)

Data independence can most easily be understood by first considering its opposite. Applications implemented on older systems—prerelational or even predatabase systems—tend to be *data-dependent*. What this means is that the way the data is physically represented in secondary storage, and the techniques used to access it, are both dictated by the requirements of the application under consideration, and moreover that *knowledge of that physical representation and those access techniques is built into the application code.* For example, suppose we have an application that uses the EMPLOYEE file of Fig. 1.5, and suppose it is decided, for performance reasons, that the file is to be indexed on its "employee name" field (see Appendix D, online). In an older system, the application in question will typically be aware of the fact that the index exists, and aware also of the sequence of records as defined by that index, and the internal structure of the application will be built around that knowledge. In particular, the precise form of the various data access and exception-checking routines within the application will depend very heavily on details of the interface presented to the application by the data management software.

We say that an application such as the one in this example is data-dependent, because it is impossible to change the physical representation (how the data is physically represented in storage) and access techniques (how it is physically accessed) without affecting the application, probably drastically. For instance, it would not be possible to replace the index in the example by a hashing scheme without making major modifications to the application code. What is more, the portions of that code requiring alteration in such a situation are precisely those portions that communicate with the data management software; the difficulties involved are quite irrelevant to the problem the application was originally written to solve—that is, they are difficulties *introduced* by the nature of the data management interface.

In a database system, however, it would be extremely undesirable to allow applications to be data-dependent in the foregoing sense, for at least the following two reasons:

1. Different applications will require different views of the same data. For example, suppose that before the enterprise introduces its integrated database there are two applications, *A* and *B*, each owning a private file that includes the field "customer balance." Suppose, however, that application *A* stores that field in decimal, whereas application *B* stores it in binary. It will still be possible to integrate the two files, and to eliminate the redundancy, provided the DBMS is ready and able to perform all necessary conversions between the stored representation chosen (which might be decimal or binary or something else again) and the form in which each application wishes to see it. For example, if it is decided to store the field in decimal, then every access by *B* will require a conversion to or from binary.

   This is a fairly trivial example of the kind of difference that might exist in a database system between the data as seen by a given application and the data as physically stored. We will consider many other possible differences later in this section.

2. The DBA—or possibly the DBMS—must have the freedom to change the physical representation and access technique in response to changing requirements, without existing applications having to be modified. For example, new kinds of data might be added to the database; new standards might be adopted; application priorities (and therefore relative performance requirements) might change; new storage devices

might become available; and so on. If applications are data-dependent, such changes will typically require corresponding changes to program code, thus tying up programmer effort that would otherwise be available for the creation of new applications. It is still not uncommon, even today, to find a significant fraction of the available programming effort devoted to this kind of maintenance (think of all the work that went into addressing the "Year 2000" problem)—hardly the best use of a scarce and valuable resource.

It follows that the provision of data independence is a major objective for database systems. Data independence can be defined as the immunity of applications to change in physical representation and access technique—which implies that the applications in question do not depend on any particular physical representation or access technique. In Chapter 2, we describe an architecture for database systems that provides a basis for achieving this objective. Before then, however, let us consider in more detail some examples of the types of changes that the DBA might wish to make, and that we might therefore wish applications to be immune to.

We start by defining three terms: *stored field*, *stored record*, and *stored file* (refer to Fig. 1.7).

- A stored field is, loosely, the smallest unit of stored data. The database will contain many occurrences (or instances) of each of several types of stored field. For example, a database containing information about different kinds of parts might include a stored field type called "part number," and then there would be one occurrence of that stored field for each kind of part (screw, hinge, lid, etc.).

  Note: The foregoing paragraph notwithstanding, you should be aware that it is common in practice to drop the qualifiers *type* and *occurrence* and to rely on context to indicate which is meant. Despite a small risk of confusion, the practice is convenient, and we will adopt it ourselves from time to time in what follows. (These remarks apply to stored records as well—see the paragraph immediately following.)

- A stored record is a collection of related stored fields. Again we distinguish between type and occurrence. A stored record occurrence (or instance) consists of a group of related stored field occurrences. For example, a stored record occurrence in the "parts" database might consist of an occurrence of each of the following stored fields: part number, part name, part color, and part weight. We say that the database contains many occurrences of the "part" stored record type—again, one occurrence for each kind of part.

- Finally, a stored file is the collection of all currently existing occurrences of one type of stored record. (We assume for simplicity that any given stored file contains just one type of stored record. This simplification does not materially affect any of our subsequent discussions.)

Now, in nondatabase systems it is normally the case that any given *logical* record as seen by some application is identical to some corresponding *stored* record. As we have already seen, however, this is not necessarily the case in a database system, because the DBA might need to be able to make changes to the stored representation of data—that is.

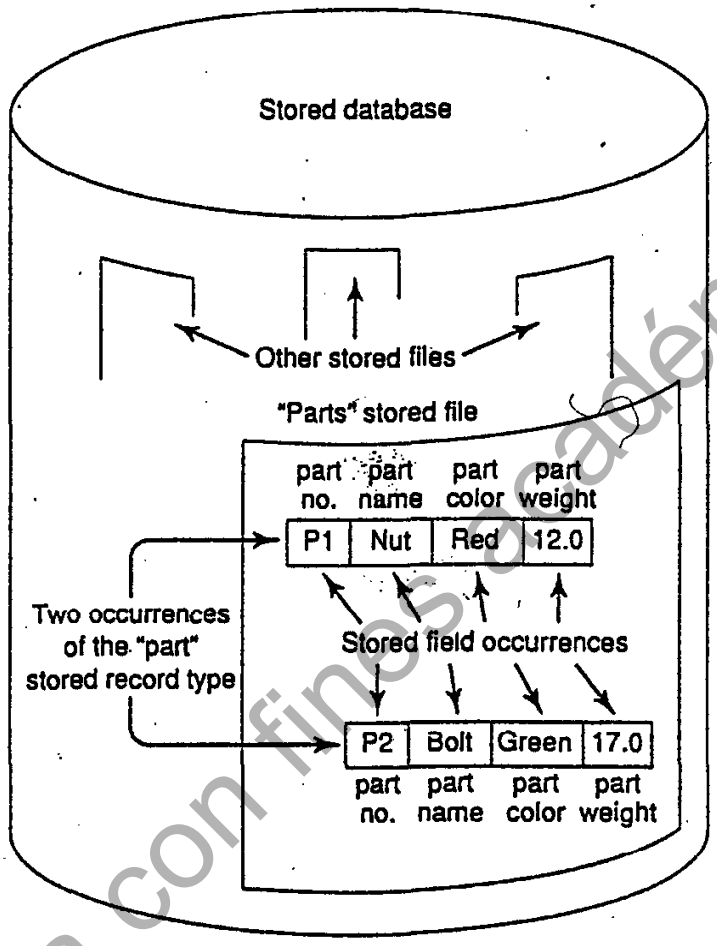

Fig. 1.7 Stored fields, records, and files

to the stored fields, records, and files—while the data as seen by applications does *not* change. For example, the SALARY field in the EMPLOYEE file might be stored in binary to economize on storage space, whereas a given COBOL application might see it as a character string. And later the DBA might decide for some reason to change the stored representation of that field from binary to decimal, say, and yet still allow the COBOL application to see it in character form.

As stated earlier, a difference such as this one—involving data type conversion on some field on each access—is comparatively minor. In general, however, the difference between what the application sees and what is physically stored might be quite considerable. To amplify this remark, we briefly consider some aspects of the stored representation that might be subject to change. You should consider in each case what the DBMS would have to do to make applications immune to such change (and indeed whether such immunity can always be achieved).

- *Representation of numeric data*

  A numeric field might be stored in internal arithmetic form (e.g., packed decimal) or as a character string. Either way, the DBA must choose whether to use *fixed* or *floating point;* what *base* or *radix* (e.g., binary or decimal) to use; what the *precision* (number of digits) should be; and, if fixed point, what the *scale factor* (number of digits after the radix point) should be. Any of these aspects might need to be changed to improve performance or to conform to a new standard or for many other reasons.

- *Representation of character data*

  A character string field might be stored using any of several distinct coded character sets—for example, ASCII, EBCDIC, or Unicode.

- *Units for numeric data*

  The units in a numeric field might change—from inches to centimeters, for example, during a process of metrication.

- *Data coding*

  In some situations it might be desirable to represent data in storage by coded values. For example, the "part color" field, which an application sees as a character string ("Red" or "Blue" or "Green" . . .), might be stored as a single decimal digit, interpreted according to the coding scheme 1 = "Red," 2 = "Blue," and so on.

- *Data materialization*

  In practice the logical field seen by an application usually does correspond directly to some specific stored field (although there might be differences in data type, coding, and so on, as we have seen). If it does, then the process of *materialization*—that is, constructing an occurrence of the logical field from the corresponding stored field occurrence and presenting it to the application—is said to be *direct.* Sometimes, however, a logical field will have no single stored counterpart; instead, its values will be materialized by means of some computation, perhaps on several stored field occurrences. For example, values of the logical field "total quantity" might be materialized by summing a number of individual stored quantities. In such a case, the materialization process is said to be *indirect.*

- *Structure of stored records*

  Two existing stored records might be combined into one. For example, the stored records

part no.	part color

  and

part no.	part weight

  might be combined to form

part no.	part color	part weight

  Such a change might occur as new applications are integrated into the database system. The implication is that a given application's logical record might consist of a

proper subset of the corresponding stored record—that is, certain fields in that stored record would be invisible to the application in question.

Alternatively, a single stored record type might be split into two. Reversing the previous example, the stored record

part no.	part color	part weight

might be split into

part no.	part color

and

part no.	part weight

Such a split would allow less frequently used portions of the original record to be stored on a slower device, for example. The implication is that a given application's logical record might contain fields from several distinct stored records—that is, it might be a proper superset of any given one of those stored records.

- *Structure of stored files*

  A given stored file can be physically implemented in storage in a wide variety of ways (see Appendix D, online). For example, it might be entirely contained within a single storage volume (e.g., a single disk), or it might be spread across several volumes (possibly on several different device types); it might or might not be physically sequenced according to the values of some stored field; it might or might not be sequenced in one or more additional ways by some other means (e.g., by one or more indexes or one or more embedded pointer chains or both); it might or might not be accessible via some hashing scheme; the stored records might or might not be physically grouped into blocks; and so on. But none of these considerations should affect applications in any way (other than performance, of course).

This concludes our partial list of aspects of the stored data representation that are subject to possible change. Note that the list implies in particular that the database should be able to grow without impairing existing applications; indeed, enabling the database to grow without logically impairing existing applications is one of the most important reasons for requiring data independence in the first place. For example, it should be possible to extend an existing stored record by the addition of new stored fields, representing, typically, further information concerning some existing type of entity (e.g., a "unit cost" field might be added to the "part" stored record). Such new fields should simply be invisible to existing applications. Likewise, it should be possible to add entirely new stored record types and new stored files, again without requiring any change to existing applications; such new records and files would typically represent new entity types (e.g., a "supplier" record type might be added to the "parts" database). Again, such additions should be invisible to existing applications.

As you might have realized by now, data independence is one of the reasons why separating the data model from its implementation, as discussed near the end of Section 1.3, is so important: To the extent we do *not* make that separation, we will not achieve data independence. The widespread failure to make the separation properly, in today's SQL systems in particular, is thus particularly distressing. *Note:* We do not mean to suggest by

these remarks that today's SQL systems provide no data independence at all—only that they provide much less than relational systems are theoretically capable of.[4] In other words, data independence is not an absolute; different systems provide it to different degrees, and few if any provide none at all. Today's SQL systems typically provide more data independence than older systems did, but they are still far from perfect, as we will see in the chapters to come.

1.6 RELATIONAL SYSTEMS AND OTHERS

We have said that SQL systems have come to dominate the DBMS marketplace, and that one important reason for this state of affairs is that such systems are based on *the relational model of data*. Informally, indeed, SQL systems are often referred to as *relational systems* specifically.[5] In addition, the vast majority of database research over the last 30 years or so has also been based (albeit a little indirectly, in some cases) on the relational model. Indeed, it is fair to say that the introduction of the relational model in 1969–70 was *the single most important event in the entire history of the database field*. For these reasons, plus the fact that the relational model is solidly based on logic and mathematics and therefore provides an ideal vehicle for teaching database foundations and principles, the emphasis in this book is heavily on relational systems.

What then exactly is a relational system? It is obviously not possible to answer this question properly at this early point in the book—but it is possible, and desirable, to give a rough and ready answer, which we can make more precise later. Briefly, then (albeit very loosely), a relational system is a system in which:

1. The data is perceived by the user as tables (and nothing but tables).
2. The operators available to the user for (e.g.) retrieval are operators that derive "new" tables from "old" ones. For example, there is one operator, *restrict*, which extracts a subset of the rows of a given table, and another, *project*, which extracts a subset of the columns—and a row subset and a column subset of a table can both be regarded in turn as tables in their own right, as we will see in just a moment.

So why are such systems called "relational"? The reason is that *relation* is basically just a mathematical term for a table. (Indeed, the terms *relation* and *table*—sometimes *relational* table for emphasis—can be taken as synonymous, at least for informal purposes. See Chapters 3 and 6 for further discussion.) Please note that the reason is definitely *not* that *relation* is "basically just a mathematical term for" a *relationship* in the sense of entity/relationship diagrams as described in Section 1.3; in fact, as noted in that section, there is very little direct connection between relational systems and such diagrams.

As promised, we will make the foregoing definitions much more precise later, but they will serve for the time being. Fig. 1.8 provides an illustration. The data—see part *a* of the figure—consists of a single table, named CELLAR (in fact, it is a scaled-down version

[4] A striking example of what relational systems *are* capable of in this respect is described in Appendix A.

[5] Despite the fact that in many respects SQL is quite notorious for its *departures* from the relational model, as we will see.

a. *Given table:* CELLAR

WINE	YEAR	BOTTLES
Zinfandel	1999	2
Fumé Blanc	2000	2
Pinot Noir	1997	3
Zinfandel	1998	9

b. *Operators (examples):*

1. *Restrict:* *Result:*

```
SELECT WINE, YEAR, BOTTLES
FROM CELLAR
WHERE YEAR > 1998 ;
```

WINE	YEAR	BOTTLES
Zinfandel	1999	2
Fumé Blanc	2000	2

2. *Project:* *Result:*

```
SELECT WINE, BOTTLES
FROM CELLAR ;
```

WINE	BOTTLES
Zinfandel	2
Fumé Blanc	2
Pinot Noir	3
Zinfandel	9

Fig. 1.8 Data structure and operators in a relational system (examples)

of the CELLAR table from Fig. 1.1, reduced in size to make it more manageable). Two sample retrievals, one involving a *restriction* or row-subsetting operation and the other a *projection* or column-subsetting operation, are shown in part *b* of the figure. The examples are expressed in SQL once again.

We can now distinguish between relational and nonrelational systems. In a relational system, the user sees the data as tables, and nothing but tables (as already explained). In a nonrelational system, by contrast, the user sees *other data structures* (either instead of or as well as the tables of a relational system). Those other structures, in turn, require other operators to access them. For example, in a hierarchic system like IBM's IMS, the data is represented to the user in the form of trees (hierarchies), and the operators provided for accessing such trees include operators for *following pointers* (namely, the pointers that implement the hierarchic paths up and down the trees). By contrast, as the examples in this chapter have shown, it is precisely an important distinguishing characteristic of relational systems that they involve no pointers (at least, no pointers visible to the user—i.e., no pointers at the model level—though there might well be pointers at the level of the physical implementation).

As the foregoing discussion suggests, database systems can be conveniently categorized according to the data structures and operators they present to the user. According to this scheme, the oldest (prerelational) systems fall into three broad categories: inverted list, hierarchic, and network systems.[6] (*Note:* The term *network* here has nothing to do

[6] By analogy with the relational model, earlier editions of this book referred to inverted list, hierarchic, and network *models* (and much of the literature still does). To talk in such terms is a little misleading, however, because—unlike the relational model—the inverted list, hierarchic, and network "models" were all invented *after the fact;* that is, commercial inverted list, hierarchic, and network products were implemented *first*, and the corresponding "models" were defined *subsequently* by a process of induction (in this context, a polite term for guesswork) from those existing implementations. See the annotation to reference [1.1] for further discussion.

with networks in the data communications sense, as described in the next chapter.) We do not discuss these categories in detail in this book because—from a technological point of view, at least—they must be regarded as obsolete. You can find tutorial descriptions of all three in reference [1.5] if you are interested.

As an aside, we remark that network systems are sometimes called either **CODASYL** systems or **DBTG** systems, after the committee that proposed them: namely, the Data Base Task Group (DBTG) of the Conference on Data Systems Languages (CODASYL). Probably the best-known example of such a system is IDMS, from Computer Associates International Inc. Like hierarchic systems (but unlike relational ones), such systems all expose pointers to the user.

The first **relational** products began to appear in the late 1970s and early 1980s. At the time of writing, the vast majority of database systems are relational (at least, they support SQL), and they run on just about every kind of hardware and software platform available. Leading examples include, in alphabetical order, DB2 (various versions) from IBM Corp., Ingres II from Computer Associates International Inc., Informix Dynamic Server from Informix Software Inc.,[7] Microsoft SQL Server from Microsoft Corp., Oracle 9i from Oracle Corp., and Sybase Adaptive Server from Sybase Inc. *Note:* When we have cause to refer to any of these products later in this book, we will refer to them (as most of the industry does, informally) by the abbreviated names DB2, Ingres (pronounced "ingress"), Informix, SQL Server, Oracle, and Sybase, respectively.

Subsequently, **object** and **object/relational** products began to become available—object systems in the late 1980s and early 1990s, object/relational systems in the late 1990s. The object/relational systems are extended versions of certain of the original SQL products (e.g., DB2, Informix); the object—sometimes *object-oriented*—systems represent attempts to do something entirely different, as in the case of GemStone from GemStone Systems Inc. and Versant ODBMS from Versant Object Technology. Such systems are discussed in Part VI of this book. (We should note that the term *object* as used in this paragraph has a rather specific meaning, which we will explain when we get to Part VI. Prior to that point, we will use the term in its normal generic sense, barring explicit statements to the contrary.)

In addition to the approaches already mentioned, research has proceeded over the years on a variety of alternative schemes, including the **multi-dimensional** approach and the **logic-based** (also called *deductive* or *expert*) approach. We discuss multi-dimensional systems in Chapter 22 and logic-based systems in Chapter 24. Also, the recent explosive growth of the World Wide Web and the use of XML has generated much interest in what has become known (not very aptly) as the **semistructured** approach. We discuss "semi-structured" systems in Chapter 27.

1.7 SUMMARY

We close this introductory chapter by summarizing the main points discussed. First, a database system can be thought of as a computerized record-keeping system. Such a sys-

[7] The DBMS division of Informix Software Inc. was acquired by IBM Corp. in 2001.

tem involves the **data** itself (stored in the **database**), **hardware**, **software** (in particular the **database management system** or DBMS), and—most important!—**users**. Users in turn can be divided into **application programmers**, **end users**, and the **database administrator** or DBA. The DBA is responsible for administering the database and database system in accordance with policies established by the **data administrator** or DA.

Databases are **integrated** and (usually) **shared**; they are used to store **persistent data**. Such data can usefully, albeit informally, be considered as representing **entities**, together with **relationships** among those entities—although in fact a relationship is really just a special kind of entity. We very briefly examined the idea of **entity/relntionship diagrams**.

Database systems provide a number of benefits, of which one of the most important is (physical) **data independence**. Data independence can be defined as the immunity of application programs to changes in the way the data is physically stored and accessed. Among other things, data independence requires that a sharp distinction be made between the data model and its **implementation**. (We remind you in passing that the term *data model*, perhaps unfortunately, has two rather different meanings.)

Database systems also usually support **transactions** or logical units of work. One advantage of transactions is that they are guaranteed to be atomic (all or nothing), even if the system fails in the middle of the transaction in question.

Finally, database systems can be based on a number of different approaches. Relational systems in particular are based on a formal theory called the **relational model**, according to which data is represented as rows in tables (interpreted as **true propositions**), and operators are provided that directly support the process of **inferring** additional true propositions from the given ones. From both an economic and a theoretical perspective, relational systems are easily the most important (and this state of affairs is not likely to change in the foreseeable future). We have seen a few simple examples of SQL, the standard language for relational systems (in particular, examples of the SQL **SELECT**, **INSERT**, **DELETE**, and **UPDATE** statements). This book is heavily based on relational systems, although—for reasons explained in the preface—not so much on SQL *per se*.

EXERCISES

1.1 *Explain the following in your own words:*

binary relationship	menu-driven interface
command-driven interface	multi-user system
concurrent access	online application
data administration	persistent data
database	property
database system	query language
data independence	redundancy
DBA	relationship
DBMS	security

entity	sharing
entity/relationship diagram	stored field
forms-driven interface	stored file
integration	stored record
integrity	transaction

1.2 What are the advantages of using a database system? What are the disadvantages?

1.3 What do you understand by the term *relational system?* Distinguish between relational and nonrelational systems.

1.4 What do you understand by the term *data model?* Explain the difference between a data model and its implementation. Why is the difference important?

1.5 Show the effects of the following SQL retrieval operations on the wine cellar database of Fig. 1.1:

a.
```
SELECT WINE, PRODUCER
FROM   CELLAR
WHERE  BIN# = 72 ;
```

b.
```
SELECT WINE, PRODUCER
FROM   CELLAR
WHERE  YEAR > 2000 ;
```

c.
```
SELECT BIN#, WINE, YEAR
FROM   CELLAR
WHERE  READY < 2003 ;
```

d.
```
SELECT WINE, BIN#, YEAR
FROM   CELLAR
WHERE  PRODUCER = 'Robt. Mondavi'
AND    BOTTLES > 6 ;
```

1.6 Give in your own words an interpretation as a true proposition of a typical row from each of your answers to Exercise 1.5.

1.7 Show the effects of the following SQL update operations on the wine cellar database of Fig. 1.1:

a.
```
INSERT
INTO   CELLAR ( BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY )
VALUES ( 80, 'Syrah', 'Meridian', 1998, 12, 2003 ) ;
```

b.
```
DELETE
FROM   CELLAR
WHERE  READY > 2004 ;
```

c.
```
UPDATE CELLAR
SET    BOTTLES = 5
WHERE  BIN# = 50 ;
```

d.
```
UPDATE CELLAR
SET    BOTTLES = BOTTLES + 2
WHERE  BIN# = 50 ;
```

1.8 Write SQL statements to perform the following operations on the wine cellar database:

a. Get bin number, name of wine, and number of bottles for all Geyser Peak wines.

b. Get bin number and name of wine for all wines for which there are more than five bottles in stock.

c. Get bin number for all red wines.

d. Add three bottles to bin number 30.

e. Remove all Chardonnay from stock.

f. Add an entry for a new case (12 bottles) of Gary Farrell Merlot: bin number 55, year 2000, ready in 2005.

1.9 Suppose you have a music collection consisting of CDs and/or minidiscs and/or LPs and/or audiotapes, and you want to build a database that will let you find which recordings you have for a specific composer (e.g., Sibelius) or conductor (e.g., Simon Rattle) or soloist (e.g., Arthur Grumiaux) or work (e.g., Beethoven's Fifth) or orchestra (e.g., the New York Philharmonic) or kind of work (e.g., violin concerto) or chamber group (e.g., the Kronos Quartet). Draw an entity/relationship diagram like that of Fig. 1.6 for this database.

REFERENCES AND BIBLIOGRAPHY

1.1 E. F. Codd: "Data Models in Database Management," Proc. Workshop on Data Abstraction, Databases, and Conceptual Modelling, Pingree Park, Colo. (June 1980), *ACM SIGMOD Record 11*, No. 2 (February 1981) and elsewhere.

Codd was the inventor of the relational model, which he first described in reference [6.1]. Reference [6.1], however, did not in fact define the term *data model* as such—but the present much later paper does. It also addresses the question: What purposes are data models in general, and the relational model in particular, intended to serve? And it goes on to offer evidence to support the claim that, contrary to popular belief, the relational model was in fact the first data model to be defined. In other words, Codd has some claim to being the inventor of the data model concept in general, as well as of the relational data model in particular.

1.2 Hugh Darwen: "What a Database *Really* Is: Predicates and Propositions," in C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings 1994–1997*. Reading, Mass.: Addison-Wesley (1998).

This paper gives a very approachable (informal but accurate) explanation of the idea, discussed briefly near the end of Section 1.3, that a database is best thought of as a collection of true propositions.

1.3 C. J. Date and P. Hopewell: "Storage Structures and Physical Data Independence," Proc. 1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control, San Diego, Calif. (November 1971).

1.4 C. J. Date and P. Hopewell: "File Definition and Logical Data Independence," Proc. 1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control, San Diego, Calif. (November 1971).

References [1.3] and [1.4] were the first papers to define and distinguish between physical and logical data independence.

1.5 C. J. Date: *Relational Database Writings 1991–1994*. Reading, Mass.: Addison-Wesley (1995).

CHAPTER 2

Database System Architecture

2.1 INTRODUCTION

We are now in a position to present an architecture for a database system. Our aim in presenting this architecture is to provide a framework on which subsequent chapters can build. Such a framework is useful for describing general database concepts and for explaining the structure of specific database systems—but we do not claim that every system can neatly be matched to this particular framework, nor do we mean to suggest that

this particular architecture provides the only possible framework. In particular, "small" systems (see Chapter 1) will probably not support all aspects of the architecture. However, the architecture does seem to fit most systems reasonably well; moreover, it is basically identical to the architecture proposed by the ANSI/SPARC Study Group on Data Base Management Systems (the so-called ANSI/SPARC architecture—see references [2.1] and [2.2]). We choose not to follow the ANSI/SPARC terminology in every detail, however.

Caveat: This chapter resembles Chapter 1 inasmuch as, while an understanding of the material it contains is essential to a full appreciation of the structure and capabilities of a modern database system, it is again somewhat abstract and dry. As with Chapter 1, therefore, you might prefer just to give the material a "once over lightly" reading for now and come back to it later as it becomes more directly relevant to the topics at hand.

2.2 THE THREE LEVELS OF THE ARCHITECTURE

The ANSI/SPARC architecture is divided into three levels, usually referred to as the internal level, the external level, and the conceptual level (see Fig. 2.1), though other names are also used. Broadly speaking:

- The internal level (also known as the *storage* level) is the one closest to physical storage—that is, it is the one concerned with the way the data is stored inside the system.
- The external level (also known as the *user logical* level) is the one closest to the users—that is, it is the one concerned with the way the data is seen by individual users.
- The conceptual level (also known as the *community logical* level, or sometimes just the *logical* level, unqualified) is a level of indirection between the other two.

Observe that the external level is concerned with *individual* user perceptions, while the conceptual level is concerned with a *community* user perception. As we saw in Chapter 1, most users will not be interested in the total database, but only in some restricted

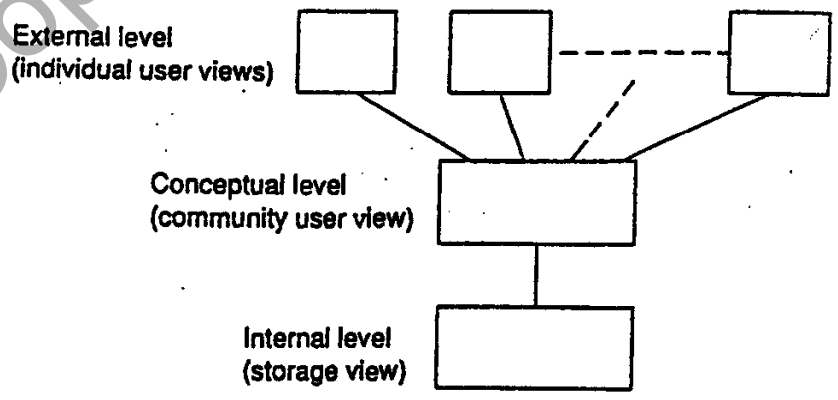

Fig. 2.1 The three levels of the architecture

portion of it; thus, there will be many distinct "external views," each consisting of a more or less abstract representation of some portion of the total database, and there will be precisely one "conceptual view," consisting of a similarly abstract representation of the database in its entirety. And then there will be precisely one "internal view," representing the database as stored internally. Note that (to use the terminology of Chapter 1) the external and conceptual levels are both *model* levels, while the internal level is an *implementation* level; in other words, the external and conceptual levels are defined in terms of user-oriented constructs such as records and fields, while the internal level is defined in terms of machine-oriented constructs such as bits and bytes.

An example will help to make these ideas clearer. Fig. 2.2 shows the conceptual view, the corresponding internal view, and two corresponding external views (one for a PL/I user and one for a COBOL user[1]), all for a simple personnel database. Of course, the example is completely hypothetical—it is not intended to resemble any real system—and many irrelevant details have deliberately been omitted. *Explanation:*

1. At the conceptual level, the database contains information concerning an entity type called EMPLOYEE. Each individual employee has an EMPLOYEE_NUMBER (six characters), a DEPARTMENT_NUMBER (four characters), and a SALARY (five decimal digits).
2. At the internal level, employees are represented by a stored record type called STORED_EMP, 20 bytes long. STORED_EMP contains four stored fields: a 6-byte prefix (presumably containing control information such as codes, flags, or pointers), and three data fields corresponding to the three properties of employees. In addition, STORED_EMP records are indexed on the EMP# field by an index called EMPX, whose definition is not shown.

External (PL/I)

```
DCL 1 EMPP,
    2 EMP# CHAR(6),
    2 SAL FIXED BIN(31);
```

External (COBOL)

```
01 EMPC.
   02 EMPNO PIC X(6).
   02 DEPTNO PIC X(4).
```

Conceptual

```
EMPLOYEE
    EMPLOYEE_NUMBER     CHARACTER(6)
    DEPARTMENT_NUMBER   CHARACTER(4)
    SALARY              DECIMAL(5)
```

Internal

```
STORED_EMP    BYTES=20
   PREFIX     BYTES=6,OFFSET=0
   EMP#       BYTES=6,OFFSET=6,INDEX=EMPX
   DEPT#      BYTES=4,OFFSET=12
   PAY        BYTES=4,ALIGN=FULLWORD,OFFSET=16
```

Fig. 2.2 An example of the three levels

[1] We apologize for using such ancient languages as the basis for this example, but the fact is that PL/I and COBOL are both still widely used in commercial installations.

3. The PL/I user has an external view of the database in which each employee is represented by a PL/I record containing two fields (department numbers are of no interest to this user and have therefore been omitted). The record type is defined by an ordinary PL/I structure declaration in accordance with normal PL/I rules.
4. Similarly, the COBOL user has an external view in which each employee is represented by a COBOL record containing, again, two fields (this time, salaries have been omitted). The record type is defined by an ordinary COBOL record description in accordance with normal COBOL rules.

Notice that corresponding data items can have different names at different points in the foregoing scheme. For example, the employee number is called EMP# in the PL/I external view, EMPNO in the COBOL external view, EMPLOYEE_NUMBER in the conceptual view, and EMP# again in the internal view. Of course, the system must be aware of the correspondences; for example, it must be told that the COBOL field EMPNO is derived from the conceptual field EMPLOYEE_NUMBER, which in turn is derived from the stored field EMP# at the internal level. Such correspondences, or mappings, are not explicitly shown in Fig. 2.2; see Section 2.6 for further discussion.

Now, it makes little difference for the purposes of the present chapter whether the system under consideration is relational or otherwise. However, it might be helpful to indicate briefly how the three levels of the architecture are typically realized in a relational system specifically:

- First, the conceptual level in such a system will definitely be relational, in the sense that the objects visible at that level will be relational tables and the operators will be relational operators (including in particular the *restrict* and *project* operators discussed briefly in Chapter 1).
- Second, a given external view will typically be relational too, or something very close to it; for example, the PL/I and COBOL record declarations of Fig. 2.2 might loosely be regarded as PL/I and COBOL analogs of the declaration of a relational table in a relational system. *Note:* We should mention in passing that the term *external view* (usually abbreviated to just *view*) unfortunately has a rather specific meaning in relational contexts that is *not* identical to the meaning assigned to it in this chapter. See Chapters 3 and (especially) 10 for an explanation and discussion of the relational meaning.
- Third, the internal level will *not* be relational, because the objects at that level will not be just (stored) relational tables—instead, they will be the same kinds of objects found at the internal level of any other kind of system (stored records, pointers, indexes, hashes, etc.). In fact, the relational model as such has **nothing whatsoever** to say about the internal level; it is, to repeat from Chapter 1, concerned with how the database looks to the *user*.

We now proceed to discuss the three levels of the architecture in considerably more detail, starting with the external level. Throughout our discussions we will be making repeated references to Fig. 2.3, which shows the major components of the architecture and their interrelationships.

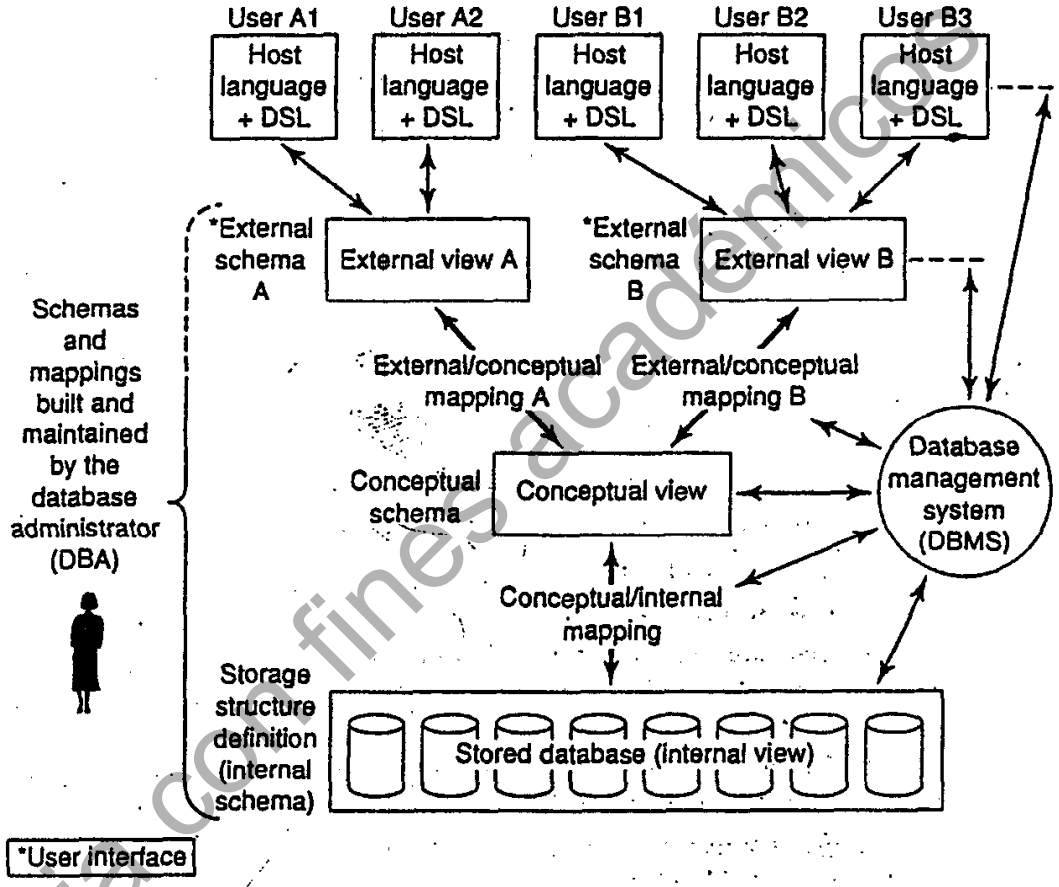

Fig. 2.3 Detailed system architecture

2.3 THE EXTERNAL LEVEL

The external level is the individual user level. As explained in Chapter 1, a given user can be an application programmer or an end user of any degree of sophistication. (The DBA is an important special case; unlike other users, however, the DBA will need to be interested in the conceptual and internal levels also. See the next two sections.)

Each user has a language at his or her disposal:

- For the application programmer, that language will be either a conventional programming language (e.g., Java, C++, or PL/I) or perhaps a proprietary language that is specific to the system in question. Such proprietary languages are sometimes called *fourth-generation* languages or 4GLs, on the (fuzzy!) grounds that (a) machine code, assembler language, and languages such as Java or C++ or PL/I can be regarded as three earlier language "generations," and (b) the proprietary languages represent the same kind of improvement over third-generation languages (3GLs) as those languages did over assembler language and assembler language did over machine code.

- For the end user, the language will be either a query language (probably SQL) or some special-purpose language, perhaps forms- or menu-driven, tailored to that user's requirements and supported by some online application as explained in Chapter 1.

For our purposes, the important thing about all such languages is that they will include a **data sublanguage**—that is, a subset of the total language that is concerned specifically with database objects and operations. The data sublanguage (abbreviated DSL in Fig. 2.3) is said to be **embedded** within the corresponding host language. The host language is responsible for providing various nondatabase facilities, such as local variables, computational operations, branching logic, and so on. A given system might support any number of host languages and any number of data sublanguages; however, one particular data sublanguage that is supported by almost all current systems is the language SQL, discussed briefly in Chapter 1. Most such systems allow SQL to be used both *interactively* as a stand-alone query language and also *embedded* in other languages such as Java or C++ or PL/I (see Chapter 4 for further discussion).

Now, although it is convenient for architectural purposes to distinguish between the data sublanguage and its containing host language, the two might in fact *not* be distinct as far as the user is concerned; indeed, it is probably preferable from the user's perspective if they are not. If they are not distinct, or if they can be distinguished only with difficulty, we say they are **tightly coupled** (and the combination is called a *database programming language*[2]). If they are clearly and easily separable, we say they are **loosely coupled**. Some commercial systems—including in particular certain SQL products, such as Oracle—support tight coupling, but not all do (tight coupling provides a more uniform set of facilities for the user but obviously involves more effort on the part of the system implementers, a fact that presumably accounts for the *status quo*).

In principle, any given data sublanguage is really a combination of at least two subordinate languages—a **data definition language** (DDL), which supports the *definition* or "declaration" of database objects, and a **data manipulation language** (DML), which supports the *processing* or "manipulation" of such objects.[3] For example, consider the PL/I user of Fig. 2.2 in Section 2.2. The data sublanguage for that user consists of those PL/I features that are used to communicate with the DBMS:

- The DDL portion consists of those declarative constructs of PL/I that are needed to declare database objects—the DECLARE (DCL) statement itself, certain PL/I data types, and possibly special extensions to PL/I to deal with new kinds of objects not supported by existing PL/I.
- The DML portion consists of those executable statements of PL/I that transfer information into and out of the database—again, possibly including special new statements.

[2] The language Tutorial D that we will be using in later chapters as a basis for examples—see the remarks on this topic in the preface to this book—is a database programming language in this sense.

[3] This rather inappropriate use of the term *manipulation* has become sanctioned by usage.

(In the interest of accuracy, we should make it clear that PL/I does not in fact include any specific database features at the time of writing. The "DML" statements in particular are typically just PL/I CALL statements that invoke the DBMS—though those calls might be syntactically disguised in some way to make them a little more user-friendly; see the discussion of *embedded SQL* in Chapter 4.)

To return to the architecture: We have already indicated that an individual user will generally be interested only in some portion of the total database; moreover, that user's view of that portion will generally be somewhat abstract when compared with the way the data is physically stored. The ANSI/SPARC term for an individual user's view is an external view. An external view is thus the content of the database as seen by some particular user: to that user, in other words, the external view *is* the database. For example, a user from the Personnel Department might regard the database as a collection of department and employee record occurrences, and might be quite unaware of the supplier and part record occurrences seen by users in the Purchasing Department.

In general, then, an external view consists of many occurrences of many types of external record (*not* necessarily the same thing as a stored record).[4] The user's data sublanguage is thus defined in terms of external records; for example, a DML *retrieve* operation will retrieve external record occurrences, not stored record occurrences. (Incidentally, we can now see that the term *logical record* used a couple of times in Chapter 1 actually referred to an external record. From this point forward, in fact, we will generally avoid the term *logical record*.)

Each external view is defined by means of an external schema, which consists basically of definitions of each of the various external record types in that external view (again, refer back to Fig. 2.2 for a couple of simple examples). The external schema is written using the DDL portion of the user's data sublanguage. (That DDL is therefore sometimes referred to as an external DDL.) For example, the employee external record type might be defined as a six-character employee number field plus a five-digit (decimal) salary field, and so on. In addition, there must be a definition of the *mapping* between the external schema and the underlying *conceptual* schema (see the next section). We will consider that mapping later, in Section 2.6.

2.4 THE CONCEPTUAL LEVEL

The conceptual view is a representation of the entire information content of the database, again (as with an external view) in a form that is somewhat abstract in comparison with the way in which the data is physically stored. It will also be quite different, in general, from the way in which the data is viewed by any particular user. Broadly speaking, the

[4] We are assuming here that all information is represented at the external level in the form of records specifically. However, some systems allow information to be represented in other ways instead of or as well as records. For a system using such alternative methods, the definitions and explanations given in this section will require suitable modification. Analogous remarks apply to the conceptual and internal levels also. Detailed consideration of such matters is beyond the scope of this early part of the book; see Chapters 14 (especially the "References and Bibliography" section) and 25 for further discussion. See also—in connection with the internal level in particular—Appendix A.

conceptual view is intended to be a view of the data "as it really is," rather than as users are forced to see it by the limitations of (for example) the particular language or hardware they might be using.

The conceptual view consists of many occurrences of many types of conceptual record. For example, it might consist of a collection of department record occurrences, plus a collection of employee record occurrences, plus a collection of supplier record occurrences, plus a collection of part record occurrences, and so on. A conceptual record is not necessarily the same as either an external record, on the one hand, or a stored record, on the other.

The conceptual view is defined by means of the conceptual schema, which includes definitions of each of the various conceptual record types (again, refer to Fig. 2.2 for a simple example). The conceptual schema is written using another data definition language, the conceptual DDL. If physical data independence is to be achieved, then those conceptual DDL definitions must not involve any considerations of physical representation or access technique at all—they must be definitions of information content *only*. Thus, there must be no reference in the conceptual schema to stored field representation, stored record sequence, indexes, hashing schemes, pointers, or any other storage and access details. If the conceptual schema is made truly data-independent in this way, then the external schemas, which are defined in terms of the conceptual schema (see Section 2.6), will *a fortiori* be data-independent too.

The conceptual view, then, is a view of the total database content, and the conceptual schema is a definition of that view. However, it would be misleading to suggest that the conceptual schema is nothing more than a set of definitions much like the simple record definitions found in (e.g.) a COBOL program today. The definitions in the conceptual schema are intended to include a great many additional features, such as the security and integrity constraints mentioned in Chapter 1. Some authorities would go as far as to suggest that the ultimate objective of the conceptual schema is to describe the complete enterprise—not just its data *per se*, but also how that data is used: how it flows from point to point within the enterprise, what it is used for at each point, what audit or other controls are to be applied at each point, and so on [2.3]. It must be emphasized, however, that no system today actually supports a conceptual schema of anything approaching this degree of comprehensiveness;[5] in most existing systems, the "conceptual schema" is little more than a simple union of all of the individual external schemas, plus certain security and integrity constraints. But it is certainly possible that systems of the future will be much more sophisticated in their support of the conceptual level.

2.5 THE INTERNAL LEVEL

The third level of the architecture is the internal level. The internal view is a low-level representation of the entire database; it consists of many occurrences of many types of internal record. *Internal record* is the ANSI/SPARC term for the construct that we have

[5] Some might argue that the so-called *business rule* systems come close (see Chapters 9 and 14).

been calling a *stored* record (and we will continue to use this latter term). The internal view is thus still at one remove from the physical level, since it does not deal in terms of *physical* records—also called **blocks** or **pages**—nor with any device-specific considerations such as cylinder or track sizes. In other words, the internal view effectively assumes an unbounded linear address space; details of how that address space is mapped to physical storage are highly system-specific and are deliberately omitted from the general architecture. *Note:* In case you are not familiar with the term, we should explain that the block, or page, is the unit of I/O—that is, it is the amount of data transferred between secondary storage and main memory in a single I/O operation. Typical page sizes can be anywhere from 1KB or less to 64KB or so, where (as we will see later) 1KB = one kilobyte = 1024 bytes.

The internal view is described by means of the internal schema, which not only defines the various stored record types but also specifies what indexes exist, how stored fields are represented, what physical sequence the stored records are in, and so on (once again, see Fig. 2.2 for a simple example; see also Appendix D, online). The internal schema is written using yet another data definition language—the internal DDL.

Note: In what follows, we will tend to use the more intuitive terms *stored database* and *stored database definition* in place of *internal view* and *internal schema*, respectively. Also, we observe that, in certain exceptional situations, application programs—in particular, those of a utility nature (see Section 2.11)—might be permitted to operate directly at the internal level rather than at the external level. Needless to say, the practice is not recommended; it represents a security risk (since the security constraints are bypassed) and an integrity risk (since the integrity constraints are bypassed likewise), and the program will be data-dependent to boot; but sometimes it might be the only way to obtain the required functionality or performance—just as an application programmer might occasionally have to descend to assembler language in order to satisfy certain functionality or performance objectives in a programming language system.

2.6 MAPPINGS

In addition to the three levels *per se*, the architecture of Fig. 2.3 involves certain mappings—one conceptual/internal mapping and several external/conceptual mappings, in general:

- The *conceptual/internal* mapping defines the correspondence between the conceptual view and the stored database; it specifies how conceptual records and fields are represented at the internal level. If the structure of the stored database is changed—that is, if a change is made to the stored database definition—then the conceptual/internal mapping must be changed accordingly, so that the conceptual schema can remain invariant. (It is the responsibility of the DBA, or possibly the DBMS, to manage such changes.) In other words, the effects of such changes must be isolated below the conceptual level, in order to preserve physical data independence.
- An *external/conceptual* mapping defines the correspondence between a particular external view and the conceptual view. In general, the differences that can exist

dump/restore purposes. In this connection, note that *multi-terabyte systems*[6]—that is, commercial database installations that store several trillions of bytes of data, loosely speaking—already exist, and systems of the future are predicted to be much larger. It goes without saying that such *VLDB* ("very large database") systems require very careful and sophisticated administration, especially if there is a requirement for continuous availability (which there usually is). Nevertheless, we will continue to talk (for the sake of simplicity) as if there were in fact just a single database.

- *Monitoring performance and responding to changing requirements*

  As indicated in Chapter 1, the DBA is responsible for organizing the system in such a way as to get the performance that is "best for the enterprise," and for making the appropriate adjustments—that is, tuning—as requirements change. For example, it might be necessary to reorganize the stored database from time to time to ensure that performance levels remain acceptable. As already mentioned, any change to the internal level of the system must be accompanied by a corresponding change to the definition of the conceptual/internal mapping, so that the conceptual schema can remain constant.

Of course, the foregoing is not an exhaustive list—it is merely intended to give some idea of the extent and nature of the DBA's responsibilities.

2.8 THE DATABASE MANAGEMENT SYSTEM

The **database management system** (DBMS) is the software that handles all access to the database. Conceptually, what happens is the following (refer to Fig. 2.3 once again):

1. A user issues an access request, using some particular data sublanguage (typically SQL).
2. The DBMS accepts that request and analyzes it.
3. The DBMS inspects, in turn, (the object versions of) the external schema for that user, the corresponding external/conceptual mapping, the conceptual schema, the conceptual/internal mapping, and the stored database definition.
4. The DBMS executes the necessary operations on the stored database.

By way of an example, consider what is involved in the retrieval of a particular external record occurrence. In general, fields will be required from several conceptual record occurrences, and each conceptual record occurrence in turn will require fields from several stored record occurrences. Conceptually, then, the DBMS must first retrieve all required stored record occurrences, then construct the required conceptual record occur-

[6] 1024 bytes = 1 kilobyte (KB); 1024KB = 1 megabyte (MB); 1024MB = 1 gigabyte (GB); 1024GB = 1 terabyte (TB); 1024TB = 1 petabyte (PB); 1024PB = 1 exabyte (EB or XB); 1024XB = 1 zettabyte (ZB); 1024ZB = 1 yottabyte (YB). Note in particular that a gigabyte is a billion bytes, loosely speaking (the abbreviation BB is sometimes used instead of GB). Incidentally (and contrary to popular belief), *gigabyte* is pronounced with a soft initial *g* and the *i* is long (as in *gigantic*).

rences, and then construct the required external record occurrence. At each stage, data type or other conversions might be necessary.

Of course, the foregoing description is very much simplified: in particular, it suggests that the entire process is interpretive, inasmuch as it describes the processes of analyzing the request, inspecting the various schemas, and so on, as if they were all done at run time. Interpretation, in turn, often implies poor performance, because of the run-time overhead. In practice, however, it might be possible for access requests to be *compiled* prior to run time (in particular, several of today's SQL products do this—see, for example, the annotation to references [4.13] and [4.27] in Chapter 4).

Let us now examine the functions of the DBMS in a little more detail. Those functions will include support for at least all of the following (refer to Fig. 2.4, overleaf):

- *Data definition*

  The DBMS must be able to accept data definitions (external schemas, the conceptual schema, the internal schema, and all associated mappings) in source form and convert them to the appropriate object form. In other words, the DBMS must include DDL processor or DDL compiler components for each of the various data definition languages (DDLs). The DBMS must also "understand" the DDL definitions, in the sense that, for example, it "understands" that EMPLOYEE external records include a SALARY field, and it must be able to use this knowledge in analyzing and responding to data manipulation requests (e.g., "Get employees with salary < $50,000").

- *Data manipulation*

  The DBMS must be able to handle requests to retrieve, update, or delete existing data in the database or to add new data to the database. In other words, the DBMS must include a DML processor or DML compiler component to deal with the data manipulation language (DML).

  In general, DML requests can be planned or unplanned:

  a. A planned request is one for which the need was foreseen in advance of the time at which the request is made. The DBA will probably have tuned the physical database design in such a way as to guarantee good performance for planned requests.

  b. An unplanned request, by contrast, is an *ad hoc* query or (less likely) update—that is, a request for which the need was not seen in advance, but instead arose in a spur-of-the-moment fashion. The physical database design might or might not be well suited for the specific request under consideration.

  To use the terminology introduced in Chapter 1 (Section 1.3), planned requests are characteristic of *operational* or *production* applications, while unplanned requests are characteristic of *decision support* applications. Furthermore, planned requests will typically be issued from prewritten application programs, whereas unplanned requests, by definition, will be issued interactively, typically via some *query language processor.* (In fact, as we saw in Chapter 1, the query language processor is really a built-in online application, not part of the DBMS *per se;* we include it in Fig. 2.4 only for completeness.)

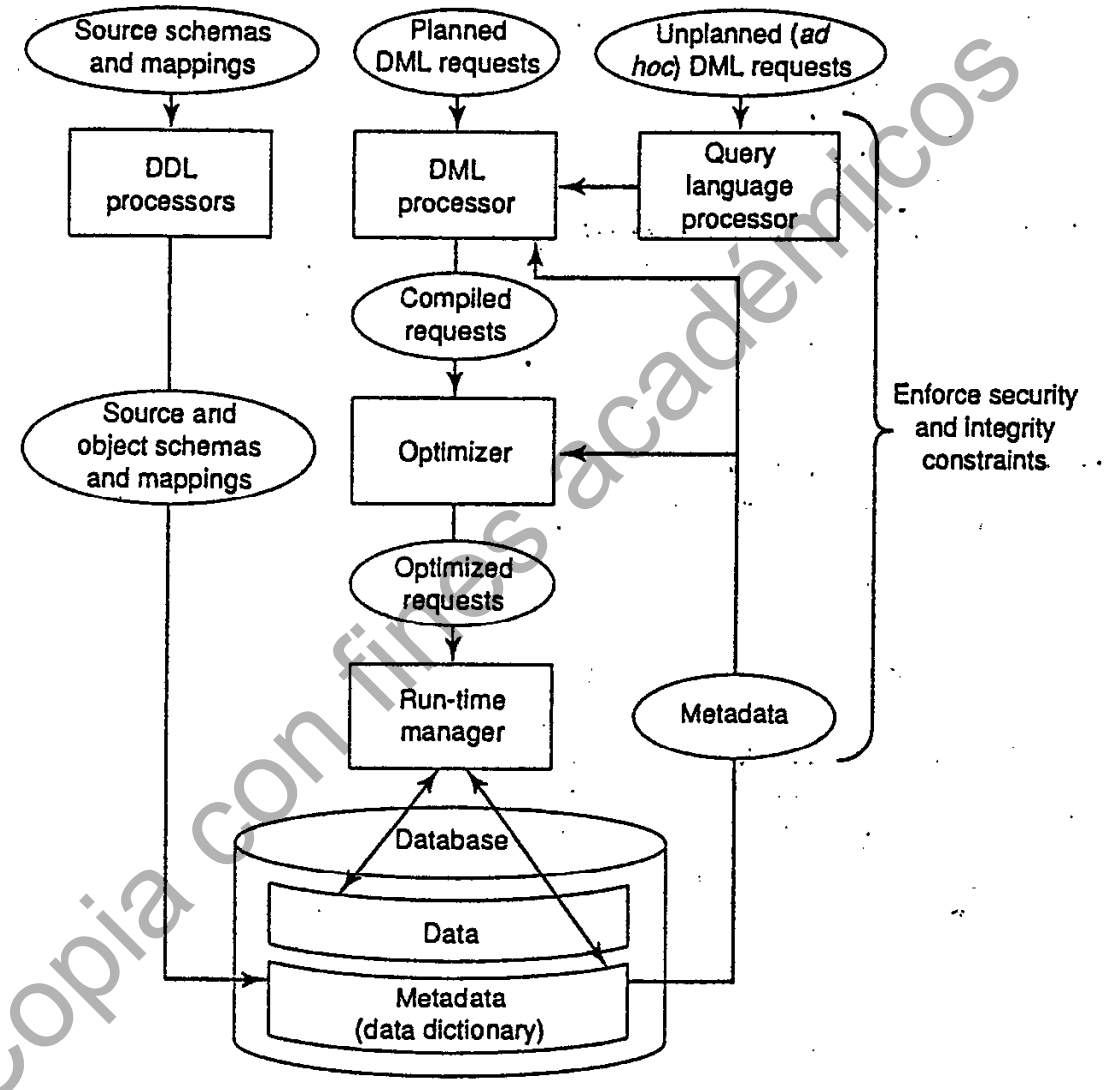

Fig. 2.4 Major DBMS functions and components

- *Optimization and execution*

  DML requests, planned or unplanned, must be processed by the optimizer component, whose purpose is to determine an efficient way of implementing the request.[7] Optimization is discussed in detail in Chapter 18. The optimized requests are then executed under the control of the run-time manager. (In practice, the run-time manager will probably invoke some kind of *file* or *storage manager* to access the stored data. File managers are discussed briefly at the end of the present section.)

[7] Throughout this book we take the term *optimization* to refer to the optimization of DML requests specifically, barring explicit statements to the contrary.

- *Data security and integrity*

  The DBMS, or some subsystem invoked by the DBMS, must monitor user requests and reject any attempt to violate the security and integrity constraints defined by the DBA (see the previous section). These tasks can be carried out at compile time or run time or some mixture of the two.

- *Data recovery and concurrency*

  The DBMS—or, more likely, another related software component called the **transaction manager** or **TP monitor**—must enforce certain recovery and concurrency controls. Details of these aspects of the system are beyond the scope of this chapter; see Part IV of this book for an in-depth discussion. The transaction manager is not shown in Fig. 2.4 because it is usually not part of the DBMS *per se*.

- *Data dictionary*

  The DBMS must provide a **data dictionary** function. The data dictionary can be regarded as a database in its own right (but a system database rather than a user database); it contains "data about the data" (sometimes called **metadata** or **descriptors**)—that is, *definitions* of other objects in the system, instead of just "raw data." In particular, all of the various schemas and mappings (external, conceptual, etc.) and all of the various security and integrity constraints will be kept, in both source and object form, in the dictionary. A comprehensive dictionary will also include much additional information, showing, for instance, which programs use which parts of the database, which users require which reports, and so on. The dictionary might even—in fact, probably should—be integrated into the database it defines and thus include its own definition. Certainly it should be possible to query the dictionary just like any other database, so that, for example, it is possible to tell which programs and/or users are likely to be affected by some proposed change to the system. See Chapter 3 for further discussion.

  Note: We are touching here on an area in which there is much terminological confusion. Some people would refer to what we are calling the dictionary as a *directory* or a *catalog*—with the tacit implication that directories and catalogs are somehow inferior to a genuine dictionary—and would reserve the term *dictionary* to refer to a specific (important) kind of application development tool. Other terms that are also sometimes used to refer to this latter kind of object are *data repository* (see Chapter 14) and *data encyclopedia*.

- *Performance*

  It goes without saying that the DBMS should perform all of its tasks as efficiently as possible.

We can summarize all of the foregoing by saying that the overall purpose of the DBMS is to provide the **user interface** to the database system. The user interface can be defined as a boundary in the system below which everything is invisible to the user. By definition, therefore, the user interface is at the *external* level. In today's SQL products, however, there are some situations—mostly having to do with update operations—in which the external level is unlikely to differ very significantly from the relevant portion of the underlying conceptual level. We will elaborate on this issue in Chapter 10.

We conclude this section by briefly contrasting database management systems (DBMSs) with *file* management systems (*file managers* or *file servers* for short). Basically, the file manager is that component of the underlying operating system that manages stored files; loosely speaking, therefore, it is "closer to the disk" than the DBMS is. (In fact, Appendix D, online, explains how the DBMS is often built *on top of* some kind of file manager.) Thus, the user of a file management system will be able to create and destroy stored files and perform simple retrieval and update operations on stored records in such files. In contrast to the DBMS, however:

- File managers are not aware of the internal structure of stored records and hence cannot handle requests that rely on a knowledge of that structure.
- File managers typically provide little or no support for security and integrity constraints.
- File managers typically provide little or no support for recovery and concurrency controls.
- There is no genuine data dictionary concept at the file manager level.
- File managers provide much less data independence than the DBMS does.
- Files are typically not "integrated" or "shared" in the same sense that the database is, but instead are usually private to some particular user or application.

2.9 DATA COMMUNICATIONS

In this section, we briefly consider the topic of **data communications**. Database requests from an end user are actually transmitted from the user's computer or workstation—which might be physically remote from the database system itself—to some online application, built-in or otherwise, and thence to the DBMS, in the form of *communication messages*. Likewise, responses back from the DBMS and online application to the user's workstation are also transmitted in the form of such messages. All such message transmissions take place under the control of another software component, the **data communications manager** (DC manager).

The DC manager is not part of the DBMS but is an autonomous system in its own right. However, since it is clearly required to work in harmony with the DBMS, the two are sometimes regarded as equal partners in a higher-level cooperative venture called a **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 will 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* (e.g., between distinct machines in a communications network such as the Internet), but that is really a separate topic.

2.10 CLIENT/SERVER ARCHITECTURE

So far in this chapter we have been discussing database systems from the point of view of the so-called ANSI/SPARC architecture. In particular, we gave a simplified picture of that architecture in Fig. 2.3. In this section we offer a slightly different perspective on the subject.

The overall purpose of a database system is to support the development and execution of database applications. From a high-level point of view, therefore, such a system can be regarded as having a very simple two-part structure, consisting of a *server*, also called the *back end*, and a set of *clients*, also called the *front ends* (refer to Fig. 2.5). *Explanation:*

1. The server is just 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 other words, "server" in this context is just another name for the DBMS.
2. The clients are the various applications that run on top of the DBMS—both user-written applications and built-in applications (i.e., applications provided by the DBMS vendor or some third party). As far as the server is concerned, of course, there is no difference between user-written and built-in applications: they all use the same interface to the server—namely, the external-level interface discussed in Section 2.3. (We note as an aside that, as mentioned in Section 2.5, certain special "utility" applications might constitute an exception to the foregoing, inasmuch as they might sometimes need to operate directly at the *internal* level of the system. Such utilities are

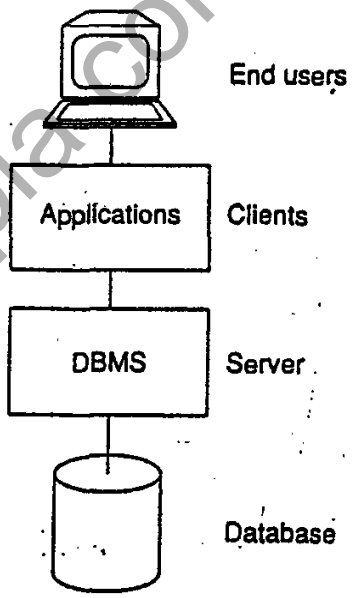

Fig. 2.5 Client/server architecture

best regarded as integral components of the DBMS, rather than as applications in the usual sense. They are discussed in more detail in the next section.)

We elaborate briefly on the question of user-written *vs.* vendor-provided applications:

- **User-written applications** are basically regular application programs, written (typically) either in a conventional 3GL such as C++ or COBOL or in some proprietary 4GL—though in both cases the language needs to be coupled somehow with an appropriate data sublanguage, as explained in Section 2.3.
- **Vendor-provided applications** (often called tools) are applications whose basic purpose is to assist in the creation and execution of other applications! The applications that are created are applications that are tailored to some specific task (they might not look much like applications 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 programs in a conventional programming language). For example, one of the vendor-provided tools will be a *report writer*, whose purpose is to allow end users to obtain formatted reports from the system on request. Any given report request can be regarded as a small application program, written in a very high-level (and special-purpose) *report writer language*.

  Vendor-provided tools can be divided into several more or less distinct classes:

  a. Query language processors
  b. Report writers
  c. Business graphics subsystems
  d. Spreadsheets
  e. Natural language processors
  f. Statistical packages
  g. Copy management or "data extract" tools
  h. Application generators (including 4GL processors)
  i. Other application development tools, including computer-aided software engineering (CASE) products
  j. Data mining and visualization tools

  and many others. Details of most such tools are beyond the scope of this book; however, we remark that since (as stated near the opening of this section) the whole point of a database system is to support the creation and execution of applications, the quality of the available tools is, or should be, a major factor in "the database decision" (i.e., the process of choosing the right database product). 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 reference. 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 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 client and server are indeed on different machines. 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 the previous section, 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. 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 examples of the kind of utilities that are typically needed in practice:

- Load routines, to create the initial version of the database from regular data files
- Unload/reload (or dump/restore) routines, to unload the database or portions thereof to backup storage and to reload data from such backup copies (of course, the "reload utility" is basically identical to the load utility just mentioned)
- Reorganization routines, to rearrange the data in the stored database for various reasons (usually having to do with performance)—for example, to cluster data in some particular way on the disk, or to reclaim space occupied by logically deleted data
- Statistical routines, to compute various performance statistics such as file sizes, value distributions, I/O counts, and so on
- Analysis routines, to analyze the statistics just mentioned

Of course, this list represents just a small sample of the range of functions that utilities typically provide; numerous other possibilities exist.

2.12 DISTRIBUTED PROCESSING

To repeat from Section 2.10, the term *distributed processing* means that distinct machines can be connected into a communications network—the Internet provides the obvious example—such that a single data processing task can span several machines in the network. (The term *parallel processing* is also 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; that is, they might be geographically dispersed in the latter case.) Communication among the various machines is handled by some kind of network management software, possibly an extension of the DC manager (discussed in Section 2.9), more likely a separate software component.

Many levels or varieties of distributed processing are possible. To repeat from Section 2.10, one simple case involves running the DBMS back end (the server) on one machine and the application front ends (the clients) on another. Refer to Fig. 2.6.

As mentioned at the end of Section 2.10, *client/server*, though strictly speaking a purely architectural term, has come to be almost synonymous with the arrangement illustrated in Fig. 2.6, 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, two or more machines 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, typically will be able—to access the same server machine. Thus, a single database might be shared across several distinct clients (see Fig. 2.7).

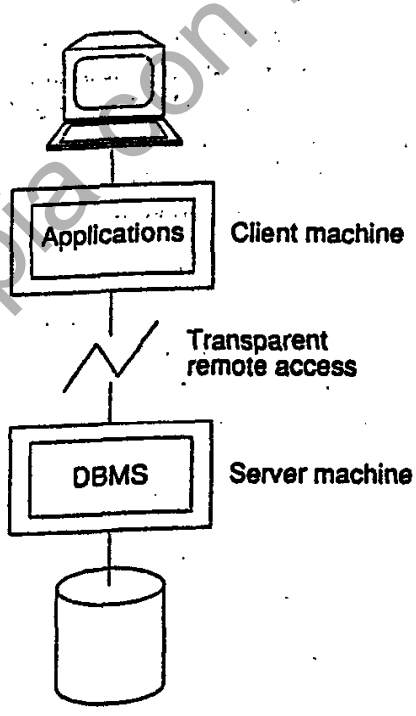

Fig. 2.6 Client(s) and server running on different machines

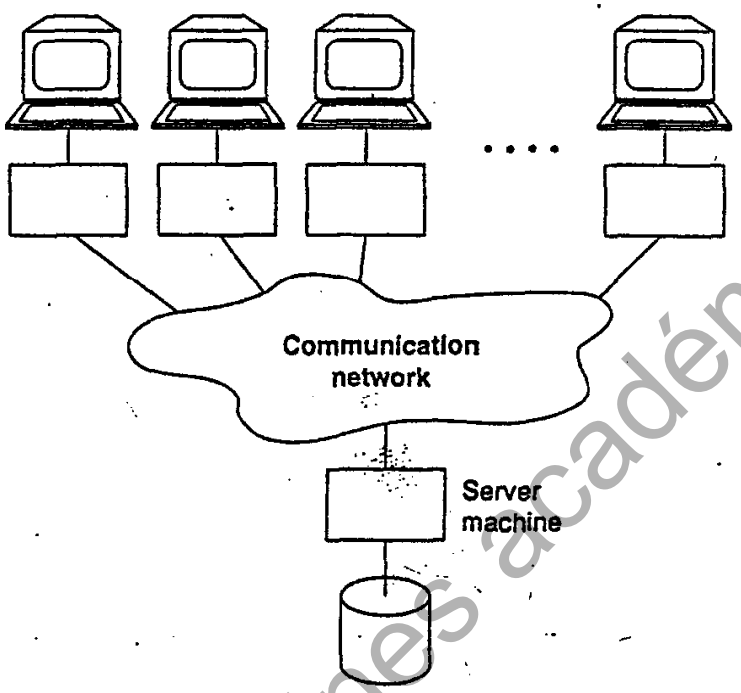

Fig. 2.7 One server machine, many client machines

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 enterprises actually 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 the data 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. 2.8); 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 different server machines (the converse of the case illustrated in Fig. 2.7). This capability is desirable because, as already mentioned, enterprises do typically operate in such a manner that the totality of their data 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 different ways:

- 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

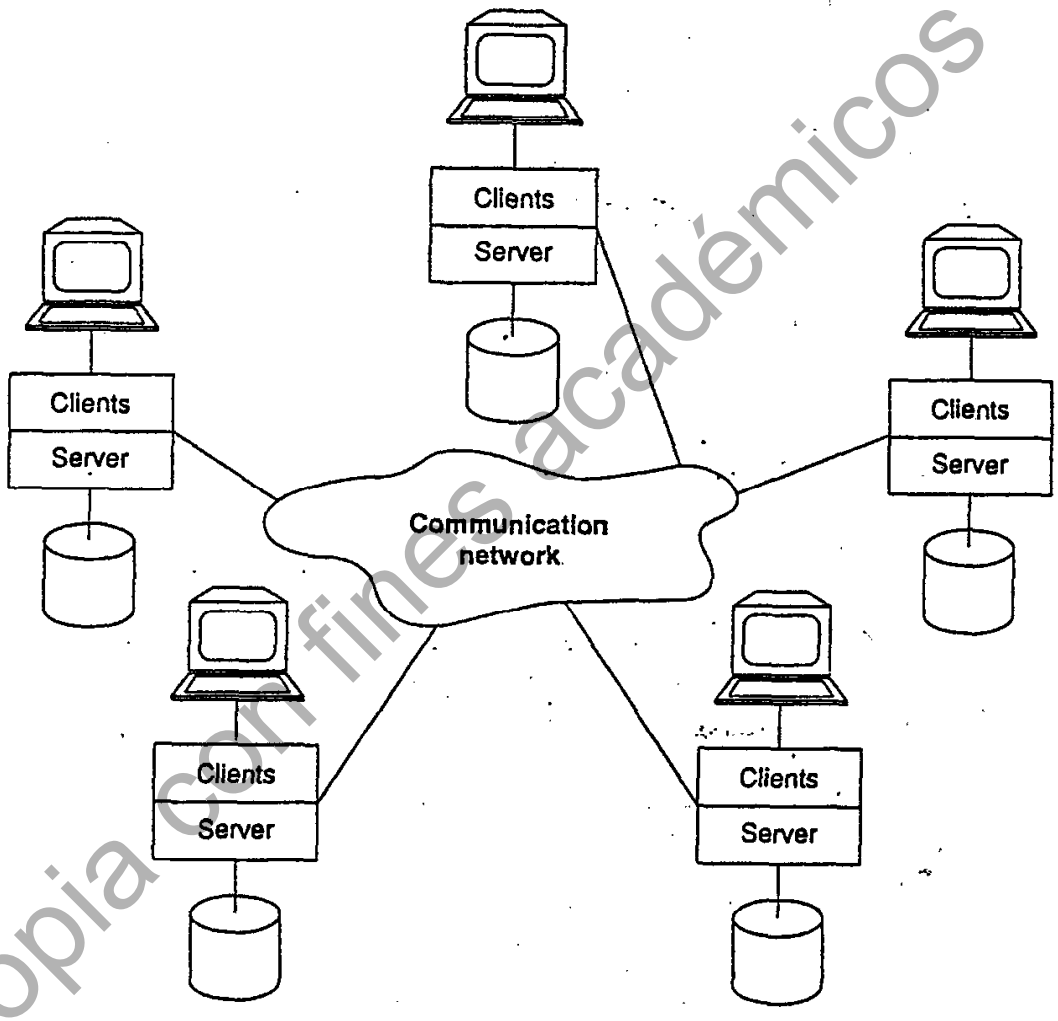

Fig. 2.8 Each machine runs both client(s) and server

different servers. Furthermore, the user in such a system has to know which particular machine holds which pieces of data.

- 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 from a logical point of view as if they were really a single server, and the user does not have to know which machines hold which pieces of data.

This latter case constitutes what is usually called a **distributed database system**. Distributed database is a big topic in its own right; carried to its logical conclusion, full distributed database support 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 by a variety of different communication networks—where "transparently" means 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 much effort has been devoted to making such systems a reality. We will discuss such systems in detail in Chapter 21.

2.13 SUMMARY

In this chapter we have looked at database systems from an 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 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 define the correspondence between (a) a given external schema and the conceptual schema, and (b) the conceptual schema and the internal schema. Those mappings are the key to the provision of **logical** and **physical data independence**, respectively.

Users—that is, end users and application programmers, both of whom operate at the external level—interact with the data by means of a **data sublanguage**, which consists of at least two components, a **data definition language** (DDL) and a **data manipulation language** (DML). The data sublanguage is embedded in a **host language**. Please note, however, that the boundaries (a) between the host language and the data sublanguage and (b) 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**); by contrast, creating the conceptual schema (**logical** or **conceptual** database design) 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 distinct 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 overall physical state of affairs—then we have a genuine **distributed database system**.

EXERCISES

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

2.2 Explain the following in your own words:

back end	front end
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	stored database definition
distributed database	unload/reload
distributed processing	unplanned request
external DDL, schema, view	user interface
external/conceptual mapping	utility

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

2.4 List the major functions performed by the DBMS.

2.5 Distinguish between logical and physical data independence.

2.6 What do you understand by the term *metadata*?

2.7 List the major functions performed by the DBA.

2.8 Distinguish between the DBMS and a file management system.

2.9 Give some examples of vendor-provided tools.

2.10 Give some examples of database utilities.

2.11 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? Is it self-describing? 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

Most of the following references are fairly old by now, but they are still relevant to the concepts introduced in the present chapter. See also the references in Chapter 14.

2.1 ANSI/X3/SPARC Study Group on Data Base Management Systems: Interim Report. *FDT* (bulletin of ACM SIGMOD) 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).

References [2.1] and [2.2] are the Interim and Final Report, 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 the American National Standards Institute (ANSI) Committee on Computers and Information Processing (X3). (*Note:* Some 25 years later, the name X3 was changed to NCITS—National Committee on Information Technology Standards. A few years later again, it was changed to INCITS—IN for International instead of National.) 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 reasonable 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 (ISO) Technical Report ISO/TR 9007:1987(E) (March 1982; revised July 1987).

This document is the report of an ISO Working Group whose objectives included "the definition of concepts for conceptual schema languages." It includes an introduction to three competing candidates (more accurately, three *sets* of candidates) for an appropriate set of formalisms, 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" schemes, (2) "binary relationship" schemes, and (3) "interpreted predicate logic" schemes. The report also includes a discussion of the fundamental concepts underlying the notion of the conceptual schema, and suggests some principles as a basis 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 William Kent: *Data and Reality*. Amsterdam, Netherlands: North-Holland/New York, N.Y.: Elsevier Science (1978).

A stimulating and thought-provoking discussion of the nature of information, and in particular of the conceptual schema. "This book projects a philosophy that life and reality are at bottom amorphous, disordered, contradictory, inconsistent, nonrational, and nonobjective" (excerpt from the final chapter). 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 model—have difficulty dealing with. Recommended.

2.5 Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis: "The GMAP: A Versatile Tool for Physical Data Independence," Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (September 1994).

GMAP stands for *Generalized Multi-level Access Path*. The authors of the paper note correctly that today's database products "force users to frame their queries in terms of a logical schema that is directly tied to physical structures," and hence are rather weak on physical data independence. In their paper, therefore, they propose a conceptual/internal mapping language (to use the terminology of the present chapter) that can be used to specify far more kinds of mappings than are typically supported in products today. Given a particular "logical schema," the language (which is based on relational algebra—see Chapter 7—and is therefore declarative, not procedural, in nature) allows the specification of numerous different "physical" or internal schemas, all of them formally derived from that logical schema. Among other things, those physical schemas can include vertical and horizontal partitioning (or "fragmentation"—see Chapter 21), any number of physical access paths, clustering, and controlled redundancy.

The paper also gives an algorithm for transforming user operations against the logical schema into equivalent operations against the physical schema. A prototype implementation shows that the DBA can tune the physical schema to "achieve significantly better performance than is possible with conventional techniques."