THE SYSTEMS PROGRAMMING SERIES

AN INTRODUCTION TO
DATABASE SYSTEMS

C. J. DATE

SIXTH EDITION

C. J. Date

AN INTRODUCTION TO

Database Systems

SIXTH
EDITION

*This book is dedicated to my wife Lindy
and to the memory of my mother Rene*

1 | An Overview of Database Management

1.1 An Introductory Example

A **database system** is essentially nothing more than a *computerized record-keeping system*. The **database** itself can be regarded as a kind of electronic filing cabinet; in other words, it is a repository for a collection of computerized data files. The user of the system will be given facilities to perform a variety of operations on such files, including the following among others:

- Adding new, empty files to the database
- Inserting new data into existing files
- Retrieving data from existing files
- Updating data in existing files
- Deleting data from existing files
- Removing existing files, empty or otherwise, from the database

By way of illustration, Fig. 1.1 shows a very small database containing just a single file, called CELLAR, which in turn contains data concerning the contents of a wine cellar. Fig. 1.2 shows an example of a **retrieval** operation against that database, together with the data (more accurately, the *result*—but it is usual in database contexts to refer to results as data also) returned from that retrieval. *Note:* Throughout this book we show all database operations and suchlike material in upper case for clarity. In practice, it is often more convenient to enter such material in lower case. Most systems will accept both.

Fig. 1.3 gives examples, all more or less self-explanatory, of **insert, update,** and **delete** operations on the wine cellar database. Examples of adding and removing entire files will be given later, in Chapters 3 and 4.

To conclude this introductory section, a few final remarks:

- First, for obvious reasons, computerized files such as CELLAR in the example are frequently referred to as **tables** rather than files (in fact, they are **relational** tables—see Section 1.6).

BIN	WINE	PRODUCER	YEAR	BOTTLES	READY
2	Chardonnay	Buena Vista	92	1	94
3	Chardonnay	Geyser Peak	92	5	94
6	Chardonnay	Stonestreet	91	4	93
12	Jo. Riesling	Jekel	93	1	94
21	Fumé Blanc	Ch. St. Jean	92	4	94
22	Fumé Blanc	Robt. Mondavi	91	2	93
30	Gewurztraminer	Ch. St. Jean	93	3	94
43	Cab. Sauvignon	Windsor	86	12	95
45	Cab. Sauvignon	Geyser Peak	89	12	97
48	Cab. Sauvignon	Robt. Mondavi	88	12	99
50	Pinot Noir	Gary Farrell	91	3	94
51	Pinot Noir	Stemmler	88	3	95
52	Pinot Noir	Dehlinger	90	2	93
58	Merlot	Clos du Bois	89	9	95
64	Zinfandel	Lytton Spring	89	9	98
72	Zinfandel	Rafanelli	90	2	98

FIG. 1.1 The wine cellar database (CELLAR file)

- Second, the rows of such a table can be thought of as representing the **records** of the file (sometimes referred to explicitly as *logical* records, to distinguish them from other kinds of records to be discussed later). Likewise, the columns can be regarded as representing the **fields** of those logical records. In this book, we will tend to use the "record" and "field" terminology when we are talking about database systems in general, the "row" and "column" terminology when we are talking about relational systems specifically. (Actually, when we get to our more formal relational discussions in later parts of the book, we will switch to more formal terms anyway.)

```
Retrieval:

SELECT WINE, BIN, PRODUCER
FROM   CELLAR
WHERE  READY = 95 ;
```

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

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

FIG. 1.2 Sample retrieval against the wine cellar database

```
Inserting new data:

INSERT
INTO    CELLAR ( BIN, WINE, PRODUCER, YEAR, BOTTLES, READY )
        VALUES ( 53, 'Pinot Noir', 'Saintsbury', 92, 1, 96 ) ;
```

```
Updating existing data:

UPDATE CELLAR
SET    BOTTLES = 4
WHERE  BIN = 3 ;
```

```
Deleting existing data:

DELETE
FROM    CELLAR
WHERE   BIN = 2 ;
```

FIG. 1.3   INSERT, UPDATE, and DELETE examples

- Third, the SELECT, INSERT, UPDATE, and DELETE operations shown in Figs. 1.2 and 1.3 above are actually all examples of statements from a database language called **SQL**. SQL is the language currently supported by most commercial database products; in fact, it is the official standard language for dealing with relational systems (see further discussion in Section 1.6 below). The name "SQL" was originally an abbreviation for "Structured Query Language," and was pronounced "sequel." Now that the language has become a standard, however, the name is just a name—it is not officially an abbreviation for anything at all—and the pendulum has swung in favor of the pronunciation "ess-cue-ell." We will assume this latter pronunciation in this book.

1.2   What Is a Database System?

To repeat from Section 1.1, a database system is basically a computerized record-keeping system; that is, it is a computerized system whose overall purpose is to maintain information and to make that information available on demand. The information concerned can be anything that is deemed to be of significance to the individual or organization the system is intended to serve—anything, in other words, that is needed to assist in the general process of running the business of that individual or organization.

Note: The terms "data" and "information" are treated as synonymous in this book. Some writers prefer to distinguish between the two, using "data" to refer to the values actually stored in the database and "information" to refer to the *meaning* of those values as understood by some user. The distinction is clearly important—so important that it seems preferable to make it explicit, where relevant, instead of relying on a somewhat arbitrary differentiation between two essentially similar terms.
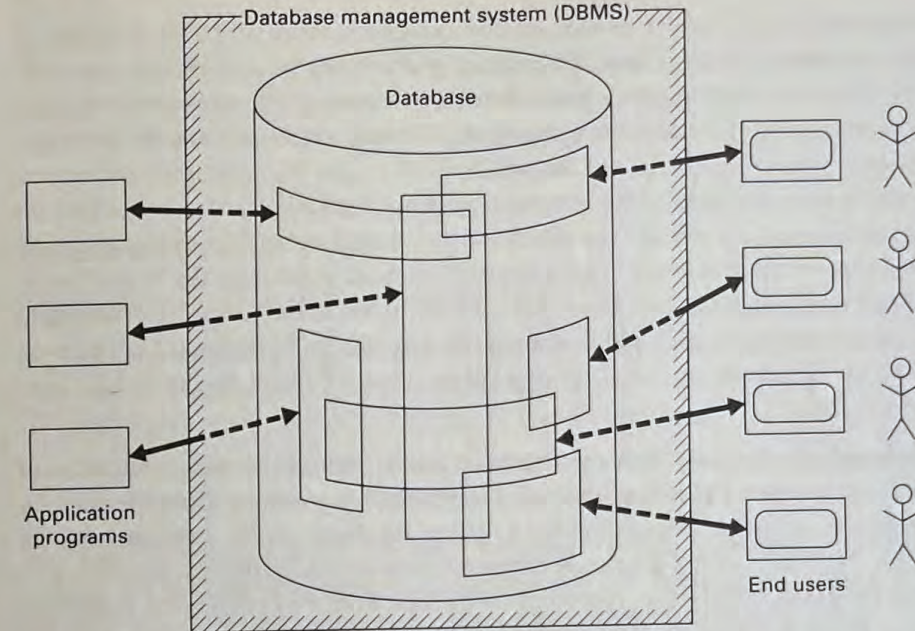
FIG. 1.4   Simplified picture of a database system

Fig. 1.4 shows a greatly simplified view of a database system. The figure is intended to illustrate the point that a database system involves four major components, namely, **data, hardware, software,** and **users**. We consider these four components briefly below. Later, of course, we will discuss each in much more detail (except for the hardware component, most details of which are beyond the scope of this book).

Data

Database systems are available on machines that range all the way from quite small micros (even portable PCs) to the largest mainframes. Needless to say, the facilities provided by any given system are to some extent determined 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 concurrently. As Fig. 1.4 suggests, we will normally assume the latter case in this book, for generality, but in fact the distinction is largely irrelevant so far as most users are concerned: A major objective of most multi-user systems is precisely to allow each individual user to behave as if he or she were working with a *single-*user system. 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 14).

Incidentally, it is usually convenient to assume for the sake of simplicity that the

totality of data stored in the system is all held in a single database, and we will normally make this assumption, 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 elsewhere in this book (e.g., in Chapter 2).

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 also. 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 that the database can be thought of as a unification of several otherwise distinct data files, with any redundancy among those files wholly or partly eliminated. For example, a given database might contain both an EMPLOYEE file, giving employee names, addresses, departments, salaries, etc., 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 that individual pieces of data in the database can be shared among several different users, in the sense that each of those users can have access to the same piece of data (and different users can use it for different purposes). As indicated earlier, different users can even be accessing the same piece of data *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 EMPLOYEE-ENROLLMENT example cited above, the department information in the EMPLOYEE file would typically be shared by users in the Personnel Department and users in the Education Department—and, as suggested above, those two classes of users would typically use that information for different purposes.

Another consequence of the fact that the database is integrated is that any given user will typically be concerned only with some small portion of the total database
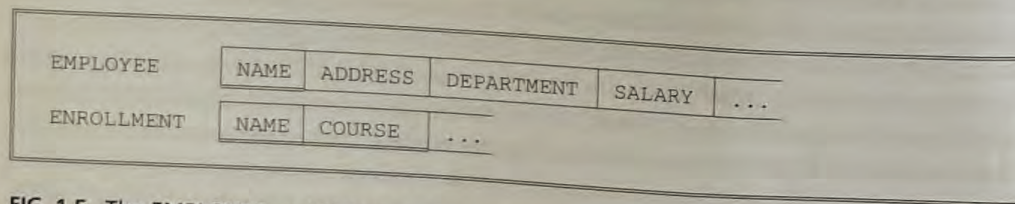
EMPLOYEE | NAME | ADDRESS | DEPARTMENT | SALARY | . . .

ENROLLMENT | NAME | COURSE | . . .

FIG. 1.5 The EMPLOYEE and ENROLLMENT files

(moreover, different users' portions will overlap in many different ways). In other words, a given database will be perceived by different users in a variety of 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 the next chapter.

Hardware

The hardware portions of the system consist of:

- The secondary storage volumes—typically moving-head 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; and

- The processor(s) and associated main memory that are used to support the execution of the database system software (see the next subsection below).

This book does not concern itself very greatly with the hardware portions of the system, for the following reasons among others: First, these 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 in numerous other places.

Software

Between the physical database itself (i.e., the data as actually stored) and the users of the system is a layer of software, the **database manager** (DB manager) or, more usually, **database management system** (DBMS). All requests from users 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 forth, 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 view of the database that is elevated somewhat above the hardware level, and supports user operations (such as the SQL operations discussed briefly in Section 1.1) that are expressed in terms of that higher-level view. We shall discuss this function, and other functions of the DBMS, in considerably more detail throughout the body of this book.

Note: 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 so on. See Chapter 2 for further discussion.

Users

We consider three broad classes of users:

- First, there are the **application programmers,** who are responsible for writing ap-

plication programs that use the database, typically in a language such as COBOL or PL/I or some more modern language such as C or Pascal. Those programs operate on the data in all the usual ways—retrieving existing information, inserting new information, deleting or changing existing information. All of these functions are of course performed by issuing the appropriate request to the DBMS. The programs themselves may be conventional batch applications, or they may be **online** applications, whose function is to support an end user (see the next paragraph) who is accessing the database from an online workstation or terminal. Most modern applications are of the online variety.

- The second class of user, then, is **end users,** who interact with the system from online workstations or terminals. 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 database system software. Such interfaces are also supported by means of online applications, of course, but those applications are **builtin,** not user-written. Most systems provide at least one such builtin application, namely an interactive **query language processor,** by which the user is able to issue high-level commands or statements (such as SELECT, INSERT, etc.) to the DBMS. The language SQL mentioned in Section 1.1 can be regarded as a typical example of a database query language.

    Note: The term "query language," common though it is, is really a misnomer, inasmuch as the English verb "query" suggests *retrieval* (only), whereas query languages typically provide UPDATE, INSERT, and DELETE operations (and probably other operations) as well.

    Most systems also provide additional builtin interfaces in which users do not issue explicit commands such as SELECT 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 = Information Systems is often used with much the same meaning). By contrast, **command-driven interfaces**—i.e., query languages—do tend to require a certain amount of professional IT expertise, though perhaps not a very great deal (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 provide certain functions that are not supported by those other interfaces.

- The third class of user (not shown in Fig. 1.4) is the **database administrator** or DBA. Discussion of the DBA function—and the associated (very important) **data** administrator function—is deferred to Sections 1.4 and 2.7.

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

1.3 What Is a Database?

Persistent Data

It is customary to refer to the data in a database as "persistent" (even though it might not actually persist for very long!). By "persistent," we mean to suggest that database data differs in kind from other, more ephemeral, data, such as input data, output data, control statements, work queues, software control blocks, intermediate results, and more generally any data that is transient in nature. Let us elaborate briefly on the terms "input data" and "output data":

- "Input data" refers to information entering the system for the very first time (typically from a terminal or workstation). Such information might cause a change to be made to the persistent data (it might *become* part of the persistent data), but it is not initially part of the database as such.

- Similarly, "output data" refers to messages and results emanating from the system (typically printed or displayed on a screen). Again, such information might be *derived from* the persistent data, but it is not itself considered to be part of the database.

Of course, the distinction between persistent and transient data is not a hard and fast one—it depends to some extent on context (i.e., how the data is being used). However, assuming that the distinction does at least make some intuitive sense, we can now give a slightly more precise definition of the term "database":

- A **database** consists of some 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 private database), or a complete corporation or similar large body (with a very 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. This is the "persistent data" referred to above. The enterprises just mentioned would typically include the following among their persistent data:

1. Product data
2. Account data
3. Patient data

4. Student data

5. Planning data

Note: The first few 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—i.e., 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). Now that databases are increasingly being used for other kinds of application as well—i.e., **decision support** applications—the term "operational data" is no longer entirely appropriate. Indeed, enterprises nowadays often maintain two distinct databases, one containing operational data and one containing decision support data. The decision support database frequently consists of *summary information* (e.g., totals, averages), where that summary information in turn is extracted from the operational database on a periodic basis—say once a day or once a week.

Entities and Relationships

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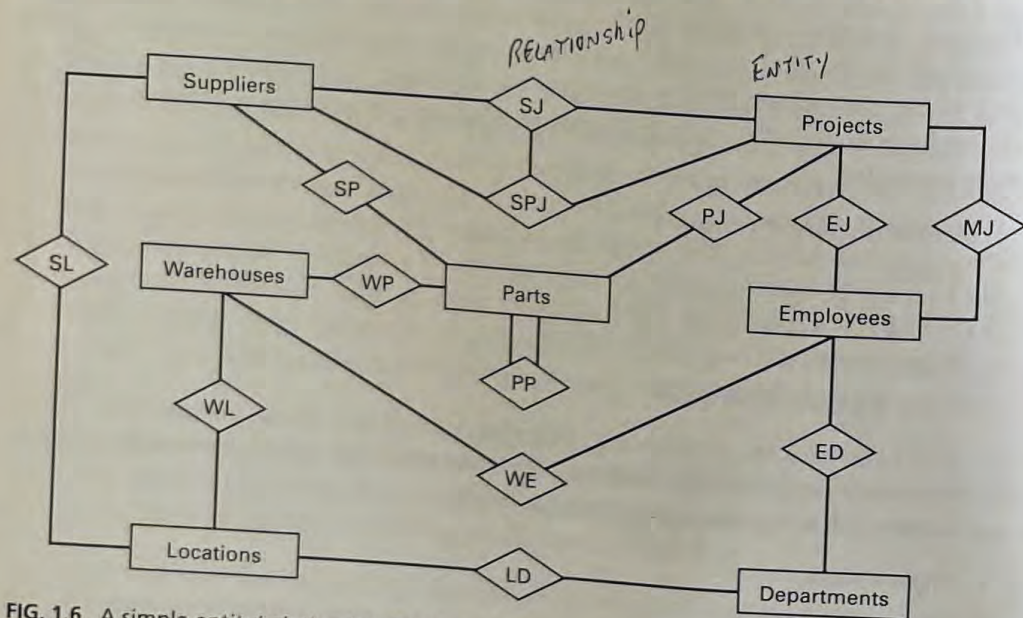

FIG. 1.6   A simple entity/relationship (E/R) diagram

It is important to understand that, in addition to the basic entities themselves, 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") 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 either of the following questions:

■ Given a supplier, find the parts supplied by that supplier

■ Given a part, find 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. Later in this book we will consider ways in which this can be done.

Incidentally, Fig. 1.6 is a simple example of what is called (for obvious reasons) an **entity/relationship diagram** (E/R diagram for short). In Chapter 12 we will consider such diagrams in some detail.

Fig. 1.6 also illustrates a number of other points:

1. Although most of the relationships in the diagram involve *two* types of entity—i.e., they are *binary* relationships—it is by no means the case that all relationships must necessarily be binary in this sense. In the example there is one relationship ("SPJ") involving three types of entity (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 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, and

(d) The Manhattan project is supplied by Smith

—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 that Py is monkey wrenches or that Jz is the Manhattan project. False

inferences such as these are examples of what is sometimes called **the connection trap**.

2. The diagram also includes one relationship (PP) involving just *one* type of entity (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 considered as 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 types of entity that are linked together, namely parts and parts, happen to be one and the same.

3. In general, a given set of entity types might be linked together in any number of distinct relationships. In the diagram, there are two relationships between projects and employees: One (EJ) represents the fact that employees are assigned to projects, the other (MJ) represents the fact that employees manage projects.

Note carefully 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—e.g., 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 generally treat relationships as just a special kind of entity.

Properties

As just indicated, we regard an entity as any object about which we wish to record information. In other words, entities (and hence relationships also) have **properties**. For example, suppliers have *locations;* parts have *weights;* projects have *priorities;* assignments have *start dates;* and so on. Such properties must therefore be represented in the database also. For example, the database might include a record type S representing the "suppliers" entity type, and that record type in turn might include a field type CITY representing the "location" property.

Properties in turn might be very simple in nature, or they might have an internal structure of arbitrary complexity. 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 plus associated descriptive text. Current database products are mostly not very good at dealing with complex properties such as drawings or text. We will return to this topic later in this book (especially in Chapter 19 and Chapters 22-25); until then, we will generally assume (where it makes any difference) that all properties are "simple" and can be represented by "simple" data types in the database. Examples of such "simple" data types include numbers, strings, dates, times, etc.

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). Let us consider the single-user case first.

Refer back to the wine cellar example once again (Fig. 1.1), which we can regard as typical of a single-user database. Now, that particular database is so small and so simple that the advantages might not be very immediately obvious. But imagine a similar database for a large restaurant, with a stock of perhaps thousands of bottles and with very frequent changes to that stock; or think of a liquor store, with again a very large stock and with high turnover on that stock. (These would typically still be single-user systems, incidentally, even though the database is larger.) The advantages of a database system over traditional, paper-based methods of record-keeping will perhaps be more readily apparent in these examples. Here are some of them:

- *Compactness:* No need for possibly voluminous paper files.
- *Speed:* The machine can retrieve and change 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.

The foregoing benefits apply with even more force in a multi-user environment, of course, where the database is likely to be much larger and much more complex than in the single-user case. However, there is one overriding additional advantage in such an environment, namely as follows: *The database system provides the enterprise with centralized control of its data* (which, as the reader should realize from Section 1.3, 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 might thus be difficult to control in any systematic way.

Data Administration and Database Administration

Let us elaborate a little on this concept of centralized control. The concept implies that (in an enterprise with a database system) there will be some identifiable person who has this central responsibility for the data. That person is the **data administrator** (sometimes abbreviated DA) mentioned briefly at the end of Section 1.2. Given that (as indicated above) the data is one of the enterprise's most valuable assets, it is imperative that there should be some person who understands the data, and the needs of the enterprise with respect to the 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 would be one that dictates who can perform what operations on what data in what circumstances—in other words, a *data security* policy (see further discussion below).

Note carefully that the data administrator is a manager, not a technician (although he or she certainly does need to have some appreciation of the capabilities of database systems at a technical level). The *technical* person responsible for implementing the data administrator's decisions is the **database administrator** (usually abbreviated DBA). Thus, the DBA, unlike the data administrator, is an *IT professional*. The job of the DBA is to create the actual database and to implement 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 related technical services. The DBA will typically have a staff of systems programmers and other technical assistants (i.e., the DBA function will typically be performed in practice by a team of several 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

We close this section by identifying some of the specific advantages that accrue from the notion of centralized control of the data.

- Redundancy can be reduced.

  In nondatabase systems each application has its own private files. This fact can often lead to considerable redundancy 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, those two files can be integrated, and the redundancy eliminated, *if* the data administrator is aware of the data requirements for both applications—i.e., *if* the enterprise has the necessary overall control.

  Incidentally, we do not mean to suggest 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 stored data. However, we do mean to suggest 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 next point below).

- Inconsistency can be avoided (to some extent).

  This is really a corollary of the previous point. Suppose that 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 stored 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 has not. At such times the

database is said to be *inconsistent*. Clearly, a database that is in an inconsistent state is capable of supplying incorrect or contradictory information to its users.

It should also be clear that 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 could 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 also. This process is known as **propagating updates**—where (as is usually the case) the term "update" is taken to include all of the operations of insertion, deletion, and modification. Note, however, that few commercially available systems today are capable of automatically propagating updates in this manner; that is, most current products do not support controlled redundancy at all, except in certain special situations.

- 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 stored data. In other words, it might be possible to satisfy the data requirements of new applications without having to create any additional stored data.

- Standards can be enforced.

  With central control of the database, the DBA (under the direction of the data administrator) can ensure that all applicable standards are observed in the representation of the data. Applicable standards might include any or all of the following: corporate, installation, departmental, industry, national, and international standards. Standardizing data representation is particularly desirable as an aid to *data interchange,* or migration of data between systems (this consideration is becoming particularly important with the advent of distributed processing technology—see Section 2.12). Likewise, data naming and documentation standards are also very desirable as an aid to data sharing and understandability.

- Security restrictions can be applied.

  Having complete jurisdiction over the database, the DBA (a) can ensure that the only means of access to the database is through the proper channels, and hence (b) can define security rules to be checked whenever access is attempted to sensitive data (again, under appropriate direction from the data administrator). Different rules 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 rules 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.

- Integrity can be maintained.

  The problem of integrity is the problem of ensuring that the data in the database is accurate. Inconsistency between two entries that purport to represent the same "fact" is an example of lack of integrity (see the discussion of this point

above); of course, that 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 D9 when no such department exists. 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 rules to be checked whenever any data update operation is attempted. (Again we are using the term "update" generically to cover all of the operations of insertion, deletion, and modification.)

It is worth pointing out that data integrity is even more important in a multiuser database system than it is in a "private files" environment, precisely because the database 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 tend to be somewhat weak in their support for integrity controls, although there have been some recent improvements in this area.

■ Conflicting requirements can be balanced.

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

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

1.5 Data Independence

Data independence can most easily be explained by first explaining its opposite. Applications implemented on older systems tend to be data-dependent. What this means is that the way in which the data is organized in secondary storage, and the technique for accessing it, are both dictated by the requirements of the application under consideration, and moreover that *knowledge of that data organization and that access technique is built into the application logic and code.*

■ *Example:* Suppose we have an application that processes the EMPLOYEE file, and suppose it is decided, for performance reasons, that the file is to be stored indexed on the "employee name" field. In an older system, the application in question will typically be aware of the fact that the index exists, and aware also of the file se-

quence 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 procedures 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 storage structure (how the data is physically stored) or access technique (how it is accessed) without affecting the application, probably drastically. For instance, it would not be possible to replace the indexed file in the example by a hash-addressed file without making major modifications to the application. What is more, the portions of the application requiring alteration in such a case 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—i.e., 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, for at least the following two reasons:

1. Different applications will need 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 records this field in decimal, whereas application B records 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. Many other possible differences will be considered later.

2. The DBA must have the freedom to change the storage structure or access technique in response to changing requirements, without having to modify existing applications. 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 types of storage device might become available; and so on. If applications are data-dependent, such changes will typically require corresponding changes to be made to programs, 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 that 25 percent or even more of the programming effort available in the installation is devoted to this kind of maintenance activity—clearly a waste of a scarce and valuable resource.

It follows that the provision of data independence is a major objective of database systems. Data independence can be defined as **the immunity of applications to change in storage structure and access technique**—which implies, of course, that the

applications concerned do not depend on any one particular storage structure or access technique. In Chapter 2, we describe an architecture for database systems that provides a basis for achieving the data independence objective. Before then, however, let us consider in more detail some examples of the types of change 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 the smallest unit of stored data. The database will, in general, contain many **occurrences** (or **instances**) of each of several **types** of stored field. For example, a database containing information about parts would probably include a stored field type called "part number," and there would be one occurrence of this stored field for each kind of part (screw, hinge, lid, etc.).
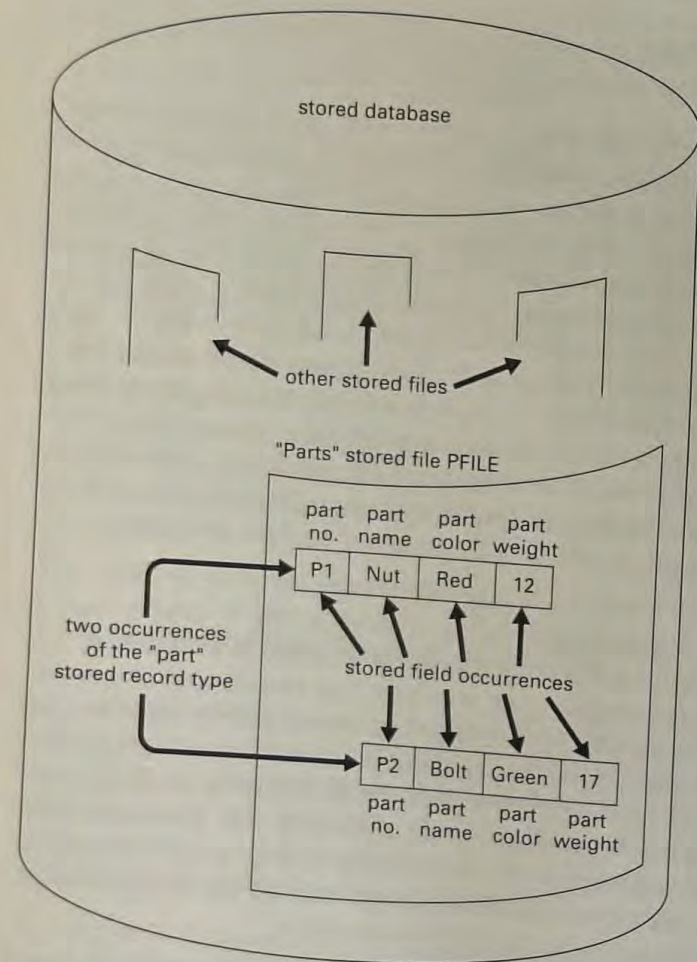

FIG. 1.7 Stored fields, records, and files

- 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 distinct kind of part).

  As an aside, we note that it is common to drop the qualifiers "type" and "occurrence" and to rely on context to indicate which is meant. Although there is a slight risk of confusion, the practice is convenient, and we will adopt it ourselves from time to time in this book.

- Finally, a **stored file** is the collection of all occurrences of one type of stored record. *Note:* We deliberately ignore the possibility of a stored file containing more than one type of stored record. This is another simplifying assumption that does not materially affect any of our subsequent discussions.

Now, in nondatabase systems it is usually the case that an application's *logical* record 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 storage structure—that is, to the stored fields, records, and files—while the corresponding logical structure does *not* change. For example, the "part weight" field mentioned above might be stored in binary to economize on storage space, whereas a given COBOL application might see it as a PICTURE item (i.e., 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, and yet still allow the application to see it in character form.

As stated earlier, a difference such as this one, involving data type conversion on a particular field on each access, is comparatively minor; in principle, however, the difference between what the application sees and what is actually stored might be quite considerable. To amplify this remark, we present below a list of aspects of the database storage structure that might be subject to variation. The reader should consider in each case what the DBMS would have to do to protect an application from such variation (and indeed whether such protection can always be achieved).

- Representation of numeric data

  A numeric field might be stored in internal arithmetic form (e.g., in packed decimal) or as a character string. Either way, the DBA must choose an appropriate base (e.g., binary or decimal), scale (fixed or floating point), mode (real or complex), and precision (number of digits). Any of these aspects might 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 or "forms-of-use" (e.g., ASCII, EBCDIC).

- Units for numeric data

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

- Data encoding

  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 will usually correspond to some specific stored field (although, as we have already seen, there might be differences in data type, units, and so on). In such a case, 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—can be 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 performed on a set of several stored field occurrences. For example, values of the logical field "total quantity" might be materialized by summing a number of individual stored quantity values. "Total quantity" here is an example of a **virtual** field, and the materialization process is said to be *indirect*. Note, however, that the user might see a difference between real and virtual fields, inasmuch as it might not be (directly) possible to insert or modify an occurrence of a virtual field.

- Structure of stored records

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

  | Part Number | Color | and | Part Number | Weight |
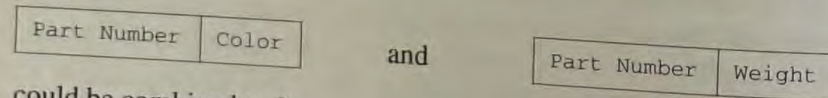
  could be combined to form

  | Part Number | Color | Weight |
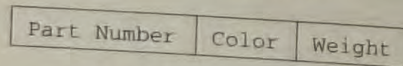
  Such a change might occur as predatabase applications are brought into the database system. It implies that an application's logical record might consist of a subset of the corresponding stored record—that is, certain fields in that stored record would be invisible to the application in question.

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

  | Part Number | Color | Weight |
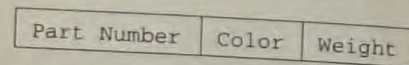
  could be broken down into

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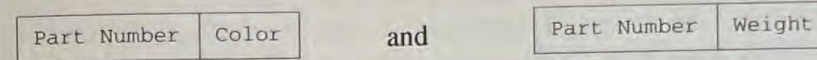
  Such a split would allow less frequently used portions of the original record to be stored on a slower device, for example. The implication is that an application's logical record might contain fields from several distinct stored records—that is, it would be a superset of any given one of those stored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

- Structure of stored files

  A given stored file can be physically implemented in storage in a wide variety of ways. 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 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 hash-addressing; the stored records might or might not be physically blocked (several per physical record); and so on. But none of these considerations should affect applications in any way (other than in performance, of course).

This concludes our list of aspects of the storage structure that are subject to possible change. The list implies (among other things) that the database should be able to **grow** without affecting existing applications; indeed, enabling the database to grow without logically impairing existing applications is probably the single most important reason 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 types of stored record (and hence new stored files), again without requiring any change to existing applications; such records would typically represent new types of entity (e.g., a "supplier" record type could be added to the "parts" database). Again, such additions should be invisible to existing applications.

We close this section by noting that data independence is not an absolute—different systems provide it in different degrees. To put this another way, few systems, if any, provide no data independence at all; it is just that some systems are less data-independent than others. Modern systems tend to be more data-independent than older systems, but they are still not perfect, as we will see in some of the chapters to come.

1.6 Relational Systems and Others

Almost all of the database products developed since the late 1970s have been based on what is called **the relational approach**; what is more, the vast majority of database research over the last 25 years has also been based—albeit a little indirectly, in some

cases—on that approach. In fact, it is undeniable that the relational approach represents the dominant trend in the marketplace today, and that the "relational model" (see Part II of this book) is the single most important development in the entire history of the database field. For these reasons, plus the additional reason that the relational model is solidly based on certain aspects of mathematics and therefore provides an ideal vehicle for teaching the concepts and principles of database systems, the emphasis in this book is very heavily on relational systems and the relational approach.

What then does it mean to say that a system is relational? It is unfortunately not possible to answer this question fully at this early point in our discussions; however, it is possible, and desirable, to give a rough-and-ready answer, which we can make more precise later. Briefly, a relational system is a system in which:

1. The data is perceived by the user as tables (and nothing but tables); and
2. The operators at the user's disposal (e.g., for data retrieval) are operators that generate new tables from old. For example, there will be one operator to extract a subset of the rows of a given table, and another to extract a subset of the columns—and of course a row subset and a column subset of a table can both in turn be regarded as tables themselves.

The reason such systems are called "relational" is that the term "relation" is essentially just a mathematical term for a table. For most practical purposes, indeed, the terms "relation" and "table" can be taken as synonymous. See Part II of this book for further discussion.

As indicated, we will make the foregoing definition considerably more precise later, but it 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 of the CELLAR table from Fig. 1.1, reduced in size to make it a little more manageable). Two sample retrievals—one involving a row-subsetting operation and the other a column-subsetting operation—are shown in part (b) of the figure. *Note:* Once again, the two retrievals are in fact examples of the SELECT statement of the language SQL first mentioned in Section 1.1.

We can now distinguish between relational and nonrelational systems, as follows. As already stated, the user of a relational system sees the data as tables. The user of a nonrelational system, by contrast, sees the data as tables, and nothing but tables. The user of a nonrelational system, by contrast, sees other data structures, either instead of or in addition to the tables of a relational system. Those other structures, in turn, require other operators to manipulate them. For example, in a **hierarchic** system, the data is represented to the user in the form of a set of tree structures (hierarchies), and the operators provided for manipulating such structures include operators for traversing hierarchic paths up and down the trees.

To pursue the point a little further: Database systems can in fact be conveniently categorized according to the data structures and operators they present to the user. First of all, older (prerelational) systems fall into three broad categories, namely **inverted**

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

list, hierarchic, and **network** systems. Examples of commercially available products in these three categories include:

Inverted list:	CA-DATACOM/DB, from Computer Associates International Inc. (previously known as DATACOM/DB, from Applied Data Research)
Hierarchic:	IMS, from IBM Corporation
Network:	CA-IDMS/DB, from Computer Associates International Inc. (previously known as IDMS, from Cullinet Software Inc.)

The first **relational** products began to appear in the late 1970s and early 1980s. At the time of writing (1993), there are well over 100—perhaps as many as 200—such products commercially available, and those products run on just about every kind of hardware and software platform imaginable. Examples of such products include DB2 from IBM Corporation; Rdb/VMS from Digital Equipment Corporation; ORACLE from Oracle Corporation; INGRES from the Ingres Division of The ASK Group Inc.; SYBASE from Sybase Inc.; and many, many more.

More recently, research has proceeded on a variety of what might be called "postrelational" systems, some of them based on upward-compatible extensions to the original relational approach, others consisting of attempts at doing something entirely different. We content ourselves for now with merely mentioning some of these more

recent approaches by name, without making any attempt at this point to explain what the names mean or what the researchers are trying to achieve:

- Deductive DBMSs
- Expert DBMSs
- Extendable DBMSs
- Object-oriented DBMSs
- Semantic DBMSs
- Universal relation DBMSs

In the case of **object-oriented** systems, in fact, some products have begun to appear, including GemStone from Servio Corporation, ObjectStore from Object Design Corporation, and OpenODB from Hewlett-Packard Corporation. We will examine some of these newer directions—object-oriented systems in particular—in later parts of the book.

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 system 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**.

Databases are **integrated** and (usually) **shared;** they are used to store **persistent** data. Such data can be usefully (albeit informally) 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/relationship diagrams**.

Database systems provide a number of benefits, of which one of the most important is **data independence** (the immunity of applications to changes in the way the data is stored and accessed).

Finally, database systems can be based on a number of different approaches, including in particular the **relational** approach. From both an economic and a theoretical perspective, the relational approach is easily the most important (and this state of affairs is not likely to change in the foreseeable future). In a relational system, the data is seen by the user as **tables,** and the operators available to the user for dealing with the data are operators that manipulate tables. We have seen a few simple examples of **SQL,** the standard language for dealing with relational systems. This book will be heavily based on the relational approach, although *not*—for reasons explained in the Preface—on SQL *per se.*

Exercises

1.1 Define the following terms:

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	

1.2 What are the advantages of using a database system?

1.3 What are the disadvantages of using a database system?

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

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 > 91 ;

(c) SELECT  BIN, WINE, YEAR
    FROM     CELLAR
    WHERE   READY < 94 ;

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

1.6 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', 89, 12, 94 ) ;

(b) DELETE
    FROM     CELLAR
    WHERE   READY > 95 ;

(c) UPDATE  CELLAR
    SET      BOTTLES = 5
    WHERE   BIN = 50 ;
```

(d) `UPDATE CELLAR`
```
    SET    BOTTLES = BOTTLES + 2
    WHERE  BIN = 50 ;
```

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

(a) Retrieve bin number, name of wine, and number of bottles for all Geyser Peak wines.

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

(c) Retrieve 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 91, ready in 96.

1.8 Suppose you have a classical music collection consisting of CDs and/or LPs and/or audio tapes, 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 NYPO) or kind of work (e.g., violin concerto) or chamber group (e.g., the Kronos Quartet). Draw an entity/relation-ship diagram like that of Fig. 1.6 for this database.

Answers to Selected Exercises

1.1 We make one remark here: The trade press, sales brochures, etc., very frequently use the term *database* when they really mean *DBMS* (e.g., "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. *Caveat lector.*

1.3 Some disadvantages are as follows:

- Security might be compromised (without good controls)
- Integrity might be compromised (without good controls)
- Additional hardware might be required
- Performance overhead might be significant
- Successful operation is crucial (the enterprise might be highly vulnerable to failure)
- The system is likely to be complex (though such complexity should be concealed from the user)

1.5 (a)

WINE	PRODUCER
Zinfandel	Rafanelli

(b)

WINE	PRODUCER
Chardonnay	Buena Vista
Chardonnay	Geyser Peak
Jo. Riesling	Jekel
Fumé Blanc	Ch. St. Jean
Gewurztraminer	Ch. St. Jean

(c)

BIN	WINE	YEAR
6	Chardonnay	91
22	Fumé Blanc	91
52	Pinot Noir	90

(d)

WINE	BIN	YEAR
Cab. Sauvignon	48	88

1.6 (a) Row for bin 80 added to the CELLAR table.

(b) Rows for bins 45, 48, 64, and 72 deleted from the CELLAR table.

(c) Row for bin 50 has number of bottles set to 5.

(d) Same as (c).

1.7 (a) `SELECT BIN, WINE, BOTTLES`
```
    FROM    CELLAR
    WHERE   PRODUCER = 'Geyser Peak' ;
```

(b) `SELECT BIN, WINE`
```
    FROM    CELLAR
    WHERE   BOTTLES > 5 ;
```

(c) `SELECT BIN`
```
    FROM    CELLAR
    WHERE   WINE = 'Cab. Sauvignon'
    OR      WINE = 'Pinot Noir'
    OR      WINE = 'Zinfandel'
    OR      WINE = 'Syrah'
    OR      ....... ;
```

There is no shortcut answer to this question, because "color of wine" is not explicitly recorded in the database.

(d) `UPDATE CELLAR`
```
    SET    BOTTLES = BOTTLES + 3
    WHERE  BIN = 30 ;
```

(e) `DELETE`
```
    FROM    CELLAR
    WHERE   WINE = 'Chardonnay' ;
```

(f) `INSERT`
```
    INTO    CELLAR ( BIN, WINE, PRODUCER, YEAR, BOTTLES, READY )
            VALUES ( 55, 'Merlot', 'Gary Farrell', 91, 12, 96 ) ;
```

2 | An Architecture for a Database System

2.1 Purpose

We are now in a position to introduce an architecture for a database system. Our aim in presenting this architecture is to provide a framework on which we can build in subsequent chapters. 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. "Small" systems, in particular, will probably not support all aspects of the architecture. However, the architecture in question does seem to fit most systems (relational or otherwise) reasonably well; moreover, it is in broad agreement with that proposed by the ANSI/SPARC Study Group on Data Base Management Systems (the so-called ANSI/SPARC architecture—see references [2.1–2.2]). We choose not to follow the ANSI/SPARC terminology in every detail, however.

One additional preliminary remark: The material of this chapter (and the preceding chapter) is of course fundamental to a full appreciation of the structure and capabilities of modern database systems. However, it is also somewhat abstract, and hence rather dry, and it does tend to involve a large number of concepts and terms that are probably new to the novice reader. In later parts of the book, you will find material that is much less abstract, and thus perhaps more immediately understandable. You might therefore prefer just to give the present chapter a "once over lightly" reading for now, and to reread individual sections more carefully later as they become 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, known as the internal, conceptual, and external levels (see Fig. 2.1). Broadly speaking:

- The **internal level** is the one closest to physical storage—i.e., it is the one concerned with the way the data is physically stored;
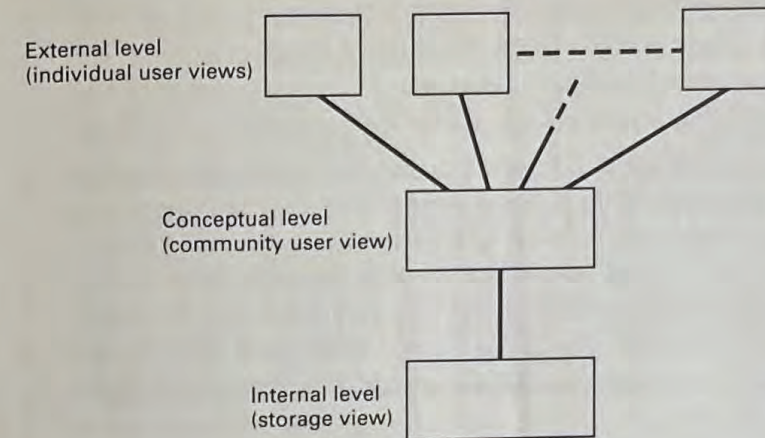
External level
(individual user views)

Conceptual level
(community user view)

Internal level
(storage view)

FIG. 2.1   The three levels of the architecture

- 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.

If the external level is concerned with *individual* user views, then the conceptual level is concerned with a *community* user view. In other words, 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. (Remember that most users will not be interested in the total database, but only in some restricted portion of it.) Likewise, there will be precisely one internal view, representing the total database as physically stored. *Note:* When we describe some representation as abstract here, we merely mean that it involves user-oriented constructs such as logical records and fields instead 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), all for a simple personnel database. Of course, the example is completely hypothetical—it is not intended to resemble any actual system—and many irrelevant details have deliberately been omitted.

We explain the example as follows.

- At the conceptual level, the database contains information concerning an entity type called EMPLOYEE. Each individual EMPLOYEE occurrence has an EMPLOYEE_NUMBER (six characters), a DEPARTMENT_NUMBER (four characters), and a SALARY (five decimal digits).
- At the internal level, employees are represented by a stored record type called STORED_EMP, twenty bytes long. STORED_EMP contains four stored fields: a six-byte prefix (presumably containing control information such as flags or pointers), and three data fields corresponding to the three properties of employees. In

```
┌─────────────────────────────┬───────────────────────────────┐
│ External (PL/I)             │ External (COBOL)              │
│                             │                               │
│ DCL 1 EMPP,                 │   01 EMPC.                     │
│     2 EMP# CHAR(6),         │      02 EMPNO PIC X(6).        │
│     2 SAL FIXED BIN(31);    │      02 DEPTNO PIC X(4).       │
└─────────────────────────────┴───────────────────────────────┘

┌─────────────────────────────────────────────────────────────┐
│ Conceptual                                                   │
│                                                              │
│        EMPLOYEE                                              │
│          EMPLOYEE_NUMBER    CHARACTER (6)                    │
│          DEPARTMENT_NUMBER  CHARACTER (4)                    │
│          SALARY             NUMERIC (5)                      │
├─────────────────────────────────────────────────────────────┤
│ Internal                                                     │
│                                                              │
│        STORED_EMP  LENGTH=20                                 │
│        PREFIX      TYPE=BYTE(6),OFFSET=0                     │
│        EMP#        TYPE=BYTE(6),OFFSET=6,INDEX=EMPX          │
│        DEPT#       TYPE=BYTE(4),OFFSET=12                    │
│        PAY         TYPE=FULLWORD,OFFSET=16                   │
└─────────────────────────────────────────────────────────────┘
```

FIG. 2.2 An example of the three levels

addition, STORED_EMP records are indexed on the EMP# field by an index called EMPX, whose definition is not shown.

- 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 from the view). The record type is defined by an ordinary PL/I structure declaration in accordance with the normal PL/I rules.

- 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 the normal COBOL rules.

Notice that corresponding objects can have different names at each point. For example, the employee number is referred to as EMP# in the PL/I view, as EMPLOYEE_NUMBER in the conceptual view, as EMPNO in the COBOL view, and as EMP# 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 represented at the internal level by the stored field EMP#. Such correspondences, or **mappings,** are not shown in Fig. 2.2. See Section 2.6.

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

- 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 (also, the operators will be relational operators, i.e., operators that work on such tables, such as the row- and column-subsetting operators discussed briefly in Section 1.6).

- Second, a given external view will typically either be relational also, or else something very close to it; for example, the PL/I and COBOL record declarations of Fig. 2.2 can be regarded as, respectively, the PL/I and COBOL equivalents of the declaration of a relational table in a relational system. *Note:* In passing we should mention the point 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 ascribed to it in this chapter. See Chapter 3 for an explanation 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 object found at the internal level of any other kind of system (stored records, pointers, indexes, hashes, etc.). In fact, relational theory 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 examine the three levels of the architecture in considerably more detail, starting with the external level. Fig. 2.3 (overleaf) shows the major components of the architecture and their interrelationships. That figure will be referenced repeatedly throughout the remainder of this chapter.

2.3 The External Level

The external level is the individual user level. As explained in Chapter 1, a given user can be either 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 one of the conventional programming languages such as C, COBOL, or PL/I, or else a proprietary language that is specific to the system in question. Such proprietary languages are often called "fourth generation" languages (4GLs), on the (very informal!) grounds that (a) machine code, assembler language, and languages such as COBOL can be regarded as three earlier language "generations," and (b) the proprietary languages represent the same kind of improvement over "third generation" languages as those languages did over assembler language.

- For the end user, the language will be either a query language or some special-purpose language, perhaps forms- or menu-driven, tailored to that user's requirements and supported by some online application program (see Section 1.2).
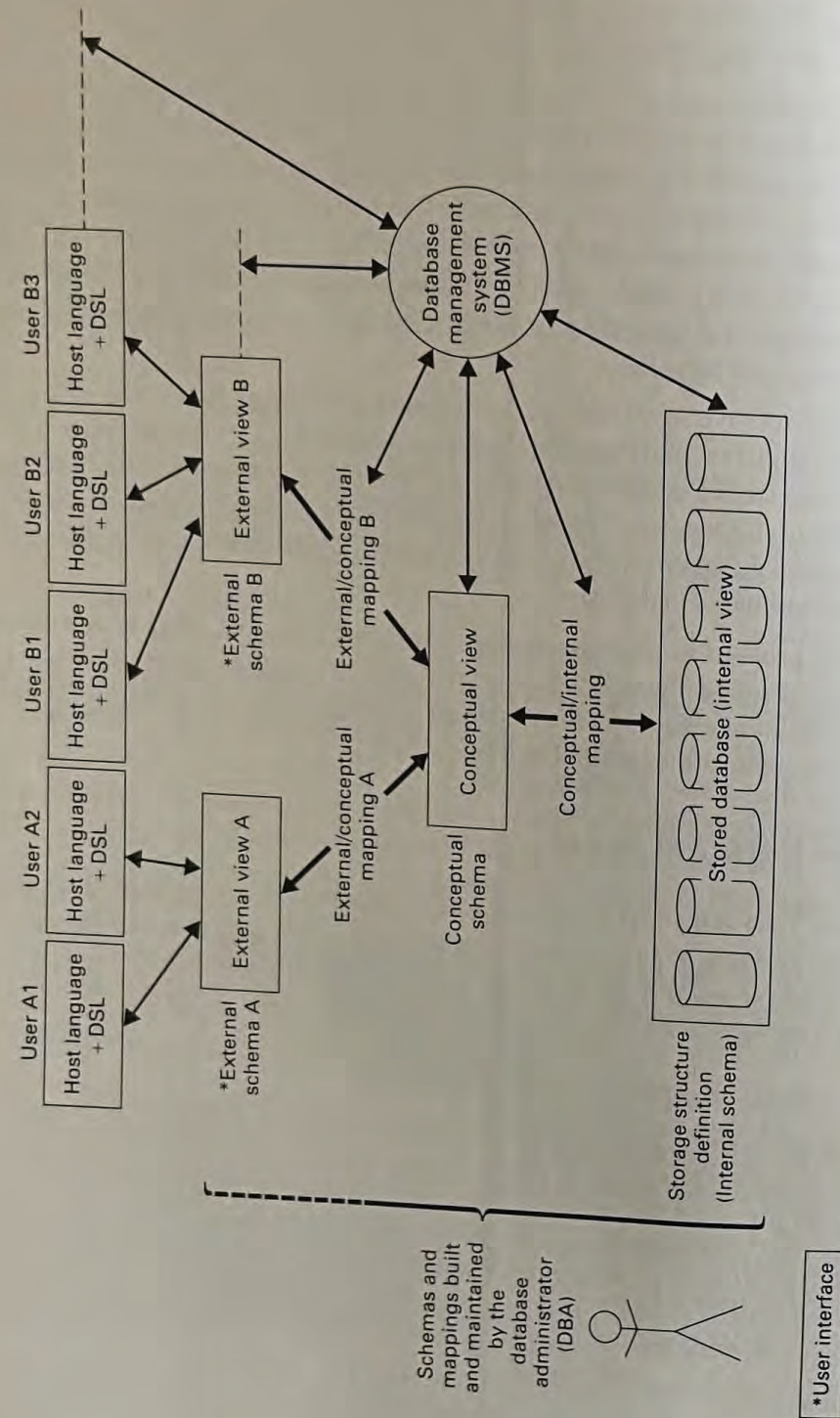
FIG. 2.3 Detailed system architecture

For our purposes, the important thing about all such languages is that they will include a **data sublanguage**—i.e., 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 (temporary) variables, computational operations, if-then-else 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 very briefly in Chapter 1. Most such systems allow SQL to be used both interactively (as a standalone query language) and also embedded in other languages such as C and COBOL. See Chapter 8 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 be *indistinguishable* so far as the user is concerned; indeed, it is preferable from the user's point of view if they *are* indistinguishable. If they are, or if they can be separated only with difficulty, we say the two are **tightly coupled**. If they are clearly and easily separable, then we say they are **loosely** coupled. Most systems today support loose coupling only. A tightly coupled system would provide a more uniform set of facilities for the user, but obviously involves more effort on the part of the system designers and developers (which presumably accounts for the status quo); however, there is evidence to suggest that there will or may be a gradual movement toward more tightly coupled systems over the next few years.

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 manipulation or processing of such objects. 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, possibly special extensions to PL/I to support new objects that are not handled by existing PL/I.

- The DML portion consists of those executable statements of PL/I that transfer information to and from the database—again, possibly including special new statements.

Note: Current PL/I does not in fact include any specific database features at all. The "DML" statements in particular are typically just calls to the DBMS (though those calls might be syntactically disguised in some manner to make them a little more user-friendly; see, e.g., the discussion of embedded SQL in Chapter 8). This is because PL/I systems, like most other systems today, currently provide only very loose coupling between the data sublanguage and its host.

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 (that is, to that user the external view *is* the database). For example, a user from the Personnel Department might regard the database as a collection of department record occurrences plus a collection of 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 each of many types of **external record** (*not* necessarily the same thing as a stored record). The user's data sublanguage is defined in terms of external records; for example, a DML *retrieve* operation will retrieve external record occurrences, not stored record occurrences. *Note:* For the time being we assume that all information is represented at the external level in the form of records. Some systems allow information to be represented in other ways as well, e.g., in the form of "links" or pointers. 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 (see Sections 2.4 and 2.5).

Incidentally, we can now see that the term "logical record" used at several points 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 (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 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 constraints of (for example) the particular language or hardware they might be using.

The conceptual view consists of many occurrences of each 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 (etc., etc.). A con-

ceptual record is not necessarily the same as either an external record, on the one hand, or a stored record, on the other.

Note: It should be pointed out that there might well be other ways of representing data at the conceptual level—ways, that is, that do not involve records as such at all, and hence might be preferable in some respects for that very reason [2.7]. For example, instead of dealing in terms of "conceptual records," it might be preferable to consider entities, and perhaps relationships too, in some more direct fashion. However, such considerations are beyond the scope of this early part of the book. See Chapters 12 and 22–25 for further discussion.

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 data independence is to be achieved, then those conceptual DDL definitions must not involve any considerations of storage structure or access technique—they must be definitions of information content *only*. Thus there must be no reference in the conceptual schema to stored field representations, stored record sequence, indexing, hash-addressing, 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 rules mentioned in Chapter 1. Some authorities would go so 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 level of anything approaching this degree of comprehensiveness; in most existing systems, the "conceptual schema" is really little more than a simple union of all individual external schemas, with the addition of certain security and integrity rules. But it seems clear that systems of the future will eventually be far 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 each of many types of **internal record**. "Internal record" is the ANSI/SPARC term for the construct that we have 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 infinite 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.

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). The internal schema is written using yet another data definition language—the *internal DDL. Note:* In this book we will normally use the more intuitive terms "stored database" in place of "internal view" and "storage structure definition" in place of "internal schema."

In closing, we remark that, in certain exceptional situations, application programs—in particular, applications 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 rules are bypassed) and an integrity risk (since the integrity rules are bypassed likewise), and the program will be data-dependent to boot; but sometimes it might be the only way to obtain the required function or performance—just as the user in a high-level programming language system might occasionally need to descend to assembler language in order to satisfy certain function or performance objectives today.

2.6   Mappings

Referring again to Fig. 2.3, the reader will observe two levels of **mapping** in the architecture, one from the conceptual level to the internal level and one from the external level to the conceptual level. 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—i.e., if a change is made to the storage structure 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 to manage such changes, of course.) In other words, the effects of such changes must be isolated below the conceptual level, in order that data independence might be preserved.

An *external/conceptual* mapping defines the correspondence between a particular external view and the conceptual view. In general, the differences that can exist between these two levels are similar to those that can exist between the conceptual view and the stored database. For example, fields can have different data types, field and record names can be changed, several conceptual fields can be combined into a single (virtual) external field, and so on. Any number of external views can exist at the same

---

* The block or page is *the unit of I/O*—i.e., it is the amount of data transferred between secondary storage and main memory in a single secondary storage access. Typical page sizes are 1K, 2K, or 4K bytes (K = 1024).

time; any number of users can share a given external view; different external views can overlap.

Incidentally, most systems permit the definition of one external view to be expressed in terms of others (in effect, via an *external/external* mapping), rather than always requiring an explicit definition of the mapping to the conceptual level—a useful feature if several external views are rather similar to one another. Relational systems in particular typically do provide such a capability.

2.7   The Database Administrator

As explained in Chapter 1, the *data* administrator is the person who makes the strategic and policy decisions regarding the data of the enterprise, and the data*base* administrator (DBA) is the person who provides the necessary technical support for implementing those decisions. Thus, the DBA is responsible for the overall control of the system at a technical level. We can now describe some of the functions of the DBA in a little more detail. In general, those functions will include the following.

■   Defining the conceptual schema

It is the *data* administrator's job to decide exactly what information is to be held in the database—in other words, to identify the entities of interest to the enterprise and to identify the information to be recorded about those entities. This process is usually referred to as **logical**—sometimes *conceptual*—**database design**. Once the data administrator has thus decided the content of the database at an abstract level, the DBA will then create the corresponding conceptual schema, using the conceptual DDL. The object (compiled) form of that schema will be used by the DBMS in responding to access requests. The source (uncompiled) form will act as a reference document for the users of the system.

(In practice, matters will rarely be as clearcut as the foregoing remarks suggest. In some cases, the data administrator will create the conceptual schema directly. In others, the DBA will do the logical design.)

■   Defining the internal schema

The DBA must also decide how the data is to be represented in the stored database. This process is usually referred to as **physical** database design. Having done the physical design, the DBA must then create the corresponding storage structure definition (i.e., the internal schema), using the internal DDL. In addition, he or she must also define the associated mapping between the internal and conceptual schemas. In practice, either the conceptual DDL or the internal DDL—most likely the former—will probably include the means for defining that mapping, but the two functions (creating the schema, defining the mapping) should be clearly separable. Like the conceptual schema, the internal schema and corresponding mapping will exist in both source and object form.

■   Liaising with users

It is the business of the DBA to liaise with users, to ensure that the data they

require is available, and to write (or help the users write) the necessary external schemas, using the applicable external DDL. (As already mentioned, a given system might support several distinct external DDLs.) In addition, the mapping between any given external schema and the conceptual schema must also be defined. In practice, the external DDL will probably include the means for specifying that mapping, but once again the schema and the mapping should be clearly separable. Each external schema and corresponding mapping will exist in both source and object form.

Other aspects of the user liaison function include consulting on application design, providing technical education, assisting with problem determination and resolution, and similar system-related professional services.

■   Defining security and integrity rules

As already discussed, security and integrity rules can be regarded as part of the conceptual schema. The conceptual DDL should include facilities for specifying such rules.

■   Defining backup and recovery procedures

Once an enterprise is committed to a database system, it becomes critically dependent on the successful operation of that system. In the event of damage to any portion of the database—caused by human error, say, or a failure in the hardware or supporting operating system—it is essential to be able to repair the data concerned with the minimum of delay and with as little effect as possible on the rest of the system. For example, the availability of data that has *not* been damaged should ideally not be affected. The DBA must define and implement an appropriate recovery scheme, involving, e.g., periodic unloading or "dumping" of the database to backup storage, and procedures for reloading the database when necessary from the most recent dump.

Incidentally, the foregoing discussion provides one reason why it might be a good idea to spread the total data collection across several databases, instead of keeping it all in one place; the individual database might very well form the unit for dump and reload purposes. Nevertheless, we will continue to talk as if there were in fact just a single database, for simplicity.

■   Monitoring performance and responding to changing requirements

As indicated in Section 1.4, the DBA is responsible for so organizing the system as to get the performance that is "best for the enterprise," and for making the appropriate adjustments as requirements change. For example, it might be necessary to **reorganize** the stored database on a periodic basis to ensure that performance levels remain acceptable. As already mentioned, any change to the physical storage (internal) level of the system must be accompanied by a corresponding change to the definition of the mapping from the conceptual level, 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.

1. A user issues an access request, using some particular data sublanguage (typically SQL).

2. The DBMS intercepts that request and analyzes it.

3. The DBMS inspects, in turn, the external schema for that user, the corresponding external/conceptual mapping, the conceptual schema, the conceptual/internal mapping, and the storage structure 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 occurrences, 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 implies that the entire process is interpretive, inasmuch as it suggests that the processes of analyzing the request, inspecting the various schemas, etc., are all done at execution time. Interpretation, in turn, usually implies poor performance, because of the execution-time overhead. In practice, however, it might be possible for access requests to be *compiled* in advance of execution time. A concrete example of a system that employs this latter approach, IBM's DB2, is briefly described in Appendix B.

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.

■   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 *language processor* 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; it must then be able to use this knowledge in interpreting and responding to user requests (e.g., a request for all employees with salary less than $50,000).

■   Data manipulation

The DBMS must be able to handle requests from the user 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 data manipulation language (DML) processor component.

In general, DML requests may be "planned" or "unplanned":

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

2. An **unplanned** request, by contrast, is an *ad hoc* query, i.e., 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 ideally suited for the specific request under consideration. In general, obtaining the best possible performance for unplanned requests represents a significant challenge for the DBMS. See Chapter 18 for an extensive discussion of this problem.

To use the terminology introduced in 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.

- Data security and integrity

The DBMS must monitor user requests and reject any attempts to violate the security and integrity rules defined by the DBA (see Section 2.7).

- Data recovery and concurrency

The DBMS—or else some other related software component, usually called the **transaction manager**—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 further information (Chapters 13 and 14).

- 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). The dictionary contains "data about the data" (sometimes called *metadata*)—that is, *definitions* of other objects in the system—rather than just "raw data." In particular, all the various schemas and mappings (external, conceptual, etc.) will physically be stored, in both source and object form, in the dictionary. A comprehensive dictionary will also include cross-reference information, showing, for instance, which programs use which pieces of the database, which users require which reports, what terminals are connected to the system, and so on. The dictionary might even—in fact, probably should—be integrated into the database it defines, and thus include its own definition. It should certainly 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 implication that directories and catalogs are some-

---

how inferior to a true dictionary—and would reserve the term "dictionary" to refer to a specific (important) kind of application development tool [2.6]. Other terms that are also sometimes used to refer to this latter kind of object are "data repository" and "data encyclopedia." See references [2.4–2.6].

- Performance

It goes without saying that the DBMS should perform all of the functions identified above as efficiently as possible.

We can summarize all of the foregoing by saying that the overall function 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. However, as we shall see in Chapter 17, there are some situations in which the external view is unlikely to differ very significantly from the relevant portion of the underlying conceptual view, at least in today's commercial products.

We conclude this section by briefly contrasting database management systems as discussed above with **file** management systems (file managers for short). Basically, the file manager is that component of the overall system that manages stored files; loosely speaking, therefore, it is "closer to the disk" than the DBMS is. (In fact, Appendix A explains how the DBMS is typically 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 (such as "Find all employees with salary less than $50,000").
- They typically provide little or no support for security and integrity rules.
- They typically provide little or no support for recovery and concurrency controls.
- There is no true data dictionary concept at the file manager level.
- They provide much less data independence than the DBMS does.

2.9 The Data Communications Manager

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