

part 1

Introduction to Databases

This page intentionally left blank

Databases and Database Users

Databases and database systems are an essential component of life in modern society: most of us encounter several activities every day that involve some interaction with a database. For example, if we go to the bank to deposit or withdraw funds, if we make a hotel or airline reservation, if we access a computerized library catalog to search for a bibliographic item, or if we purchase something online—such as a book, toy, or computer—chances are that our activities will involve someone or some computer program accessing a database. Even purchasing items at a supermarket often automatically updates the database that holds the inventory of grocery items.

These interactions are examples of what we may call **traditional database applications**, in which most of the information that is stored and accessed is either textual or numeric. In the past few years, advances in technology have led to exciting new applications of database systems. The proliferation of social media Web sites, such as Facebook, Twitter, and Flickr, among many others, has required the creation of huge databases that store nontraditional data, such as posts, tweets, images, and video clips. New types of database systems, often referred to as **big data** storage systems, or **NOSQL systems**, have been created to manage data for social media applications. These types of systems are also used by companies such as Google, Amazon, and Yahoo, to manage the data required in their Web search engines, as well as to provide **cloud storage**, whereby users are provided with storage capabilities on the Web for managing all types of data including documents, programs, images, videos and emails. We will give an overview of these new types of database systems in Chapter 24.

We now mention some other applications of databases. The wide availability of photo and video technology on cellphones and other devices has made it possible to

store images, audio clips, and video streams digitally. These types of files are becoming an important component of **multimedia databases**. **Geographic information systems (GISs)** can store and analyze maps, weather data, and satellite images. **Data warehouses** and **online analytical processing (OLAP)** systems are used in many companies to extract and analyze useful business information from very large databases to support decision making. **Real-time** and **active database technology** is used to control industrial and manufacturing processes. And database **search techniques** are being applied to the World Wide Web to improve the search for information that is needed by users browsing the Internet.

To understand the fundamentals of database technology, however, we must start from the basics of traditional database applications. In Section 1.1 we start by defining a database, and then we explain other basic terms. In Section 1.2, we provide a simple UNIVERSITY database example to illustrate our discussion. Section 1.3 describes some of the main characteristics of database systems, and Sections 1.4 and 1.5 categorize the types of personnel whose jobs involve using and interacting with database systems. Sections 1.6, 1.7, and 1.8 offer a more thorough discussion of the various capabilities provided by database systems and discuss some typical database applications. Section 1.9 summarizes the chapter.

The reader who desires a quick introduction to database systems can study Sections 1.1 through 1.5, then skip or browse through Sections 1.6 through 1.8 and go on to Chapter 2.

1.1 Introduction

Databases and database technology have had a major impact on the growing use of computers. It is fair to say that databases play a critical role in almost all areas where computers are used, including business, electronic commerce, social media, engineering, medicine, genetics, law, education, and library science. The word *database* is so commonly used that we must begin by defining what a database is. Our initial definition is quite general.

A **database** is a collection of related data.¹ By **data**, we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. Nowadays, this data is typically stored in mobile phones, which have their own simple database software. This data can also be recorded in an indexed address book or stored on a hard drive, using a personal computer and software such as Microsoft Access or Excel. This collection of related data with an implicit meaning is a database.

The preceding definition of database is quite general; for example, we may consider the collection of words that make up this page of text to be related data and hence to

¹We will use the word *data* as both singular and plural, as is common in database literature; the context will determine whether it is singular or plural. In standard English, *data* is used for plural and *datum* for singular.

constitute a database. However, the common use of the term *database* is usually more restricted. A database has the following implicit properties:

- A database represents some aspect of the real world, sometimes called the **miniworld** or the **universe of discourse (UoD)**. Changes to the miniworld are reflected in the database.
- A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.
- A database is designed, built, and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

In other words, a database has some source from which data is derived, some degree of interaction with events in the real world, and an audience that is actively interested in its contents. The end users of a database may perform business transactions (for example, a customer buys a camera) or events may happen (for example, an employee has a baby) that cause the information in the database to change. In order for a database to be accurate and reliable at all times, it must be a true reflection of the miniworld that it represents; therefore, changes must be reflected in the database as soon as possible.

A database can be of any size and complexity. For example, the list of names and addresses referred to earlier may consist of only a few hundred records, each with a simple structure. On the other hand, the computerized catalog of a large library may contain half a million entries organized under different categories—by primary author's last name, by subject, by book title—with each category organized alphabetically. A database of even greater size and complexity would be maintained by a social media company such as Facebook, which has more than a billion users. The database has to maintain information on which users are related to one another as *friends*, the postings of each user, which users are allowed to see each posting, and a vast amount of other types of information needed for the correct operation of their Web site. For such Web sites, a large number of databases are needed to keep track of the constantly changing information required by the social media Web site.

An example of a large commercial database is Amazon.com. It contains data for over 60 million active users, and millions of books, CDs, videos, DVDs, games, electronics, apparel, and other items. The database occupies over 42 terabytes (a terabyte is 10^{12} bytes worth of storage) and is stored on hundreds of computers (called servers). Millions of visitors access Amazon.com each day and use the database to make purchases. The database is continually updated as new books and other items are added to the inventory, and stock quantities are updated as purchases are transacted.

A database may be generated and maintained manually or it may be computerized. For example, a library card catalog is a database that may be created and maintained manually. A computerized database may be created and maintained either by a group of application programs written specifically for that task or by a

database management system. Of course, we are only concerned with computerized databases in this text.

A **database management system (DBMS)** is a computerized system that enables users to create and maintain a database. The DBMS is a *general-purpose software system* that facilitates the processes of *defining*, *constructing*, *manipulating*, and *sharing* databases among various users and applications. **Defining** a database involves specifying the data types, structures, and constraints of the data to be stored in the database. The database definition or descriptive information is also stored by the DBMS in the form of a database catalog or dictionary; it is called **meta-data**. **Constructing** the database is the process of storing the data on some storage medium that is controlled by the DBMS. **Manipulating** a database includes functions such as querying the database to retrieve specific data, updating the database to reflect changes in the miniworld, and generating reports from the data. **Sharing** a database allows multiple users and programs to access the database simultaneously.

An **application program** accesses the database by sending queries or requests for data to the DBMS. A **query**² typically causes some data to be retrieved; a **transaction** may cause some data to be read and some data to be written into the database.

Other important functions provided by the DBMS include *protecting* the database and *maintaining* it over a long period of time. **Protection** includes *system protection* against hardware or software malfunction (or crashes) and *security protection* against unauthorized or malicious access. A typical large database may have a life cycle of many years, so the DBMS must be able to **maintain** the database system by allowing the system to evolve as requirements change over time.

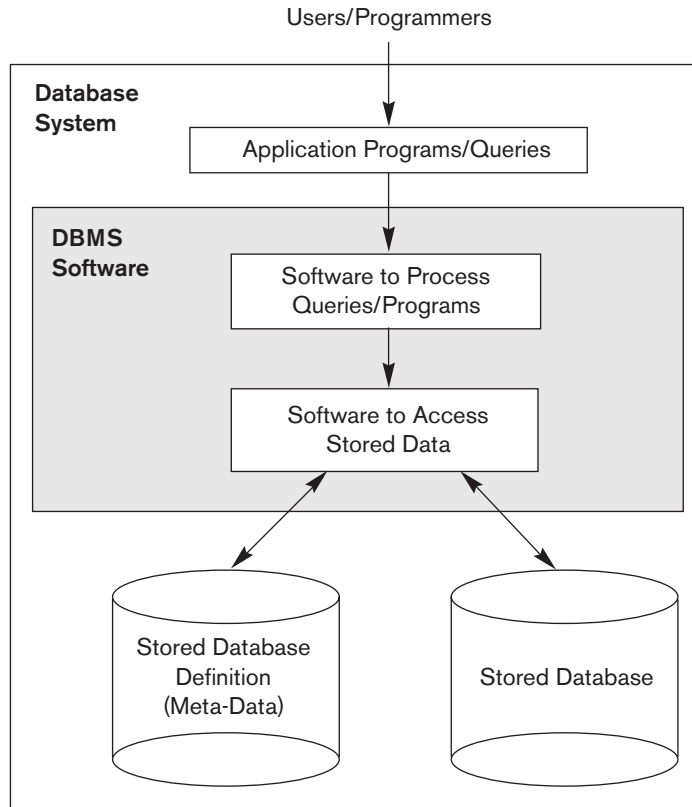
It is not absolutely necessary to use general-purpose DBMS software to implement a computerized database. It is possible to write a customized set of programs to create and maintain the database, in effect creating a *special-purpose* DBMS software for a specific application, such as airlines reservations. In either case—whether we use a general-purpose DBMS or not—a considerable amount of complex software is deployed. In fact, most DBMSs are very complex software systems.

To complete our initial definitions, we will call the database and DBMS software together a **database system**. Figure 1.1 illustrates some of the concepts we have discussed so far.

1.2 An Example

Let us consider a simple example that most readers may be familiar with: a UNIVERSITY database for maintaining information concerning students, courses, and grades in a university environment. Figure 1.2 shows the database structure and a few sample data records. The database is organized as five files, each of which

²The term *query*, originally meaning a question or an inquiry, is sometimes loosely used for all types of interactions with databases, including modifying the data.

**Figure 1.1**

A simplified database system environment.

stores **data records** of the same type.³ The STUDENT file stores data on each student, the COURSE file stores data on each course, the SECTION file stores data on each section of a course, the GRADE_REPORT file stores the grades that students receive in the various sections they have completed, and the PREREQUISITE file stores the prerequisites of each course.

To *define* this database, we must specify the structure of the records of each file by specifying the different types of **data elements** to be stored in each record. In Figure 1.2, each STUDENT record includes data to represent the student's Name, Student_number, Class (such as freshman or '1', sophomore or '2', and so forth), and Major (such as mathematics or 'MATH' and computer science or 'CS'); each COURSE record includes data to represent the Course_name, Course_number, Credit_hours, and Department (the department that offers the course), and so on. We must also specify a **data type** for each data element within a record. For example, we can specify that Name of STUDENT is a string of alphabetic characters, Student_number of STUDENT is an integer, and Grade of GRADE_REPORT is a

³We use the term *file* informally here. At a conceptual level, a *file* is a *collection* of records that may or may not be ordered.

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2

A database that stores student and course information.

single character from the set {'A', 'B', 'C', 'D', 'F', 'I'}. We may also use a coding scheme to represent the values of a data item. For example, in Figure 1.2 we represent the Class of a STUDENT as 1 for freshman, 2 for sophomore, 3 for junior, 4 for senior, and 5 for graduate student.

To *construct* the UNIVERSITY database, we store data to represent each student, course, section, grade report, and prerequisite as a record in the appropriate file. Notice that records in the various files may be related. For example, the record for Smith in the STUDENT file is related to two records in the GRADE_REPORT file that specify Smith's grades in two sections. Similarly, each record in the PREREQUISITE file relates two course records: one representing the course and the other representing the prerequisite. Most medium-size and large databases include many types of records and have *many relationships* among the records.

Database *manipulation* involves querying and updating. Examples of queries are as follows:

- Retrieve the transcript—a list of all courses and grades—of 'Smith'
- List the names of students who took the section of the 'Database' course offered in fall 2008 and their grades in that section
- List the prerequisites of the 'Database' course

Examples of updates include the following:

- Change the class of 'Smith' to sophomore
- Create a new section for the 'Database' course for this semester
- Enter a grade of 'A' for 'Smith' in the 'Database' section of last semester

These informal queries and updates must be specified precisely in the query language of the DBMS before they can be processed.

At this stage, it is useful to describe the database as part of a larger undertaking known as an information system within an organization. The Information Technology (IT) department within an organization designs and maintains an information system consisting of various computers, storage systems, application software, and databases. Design of a new application for an existing database or design of a brand new database starts off with a phase called **requirements specification and analysis**. These requirements are documented in detail and transformed into a **conceptual design** that can be represented and manipulated using some computerized tools so that it can be easily maintained, modified, and transformed into a database implementation. (We will introduce a model called the Entity-Relationship model in Chapter 3 that is used for this purpose.) The design is then translated to a **logical design** that can be expressed in a data model implemented in a commercial DBMS. (Various types of DBMSs are discussed throughout the text, with an emphasis on relational DBMSs in Chapters 5 through 9.)

The final stage is **physical design**, during which further specifications are provided for storing and accessing the database. The database design is implemented, populated with actual data, and continuously maintained to reflect the state of the miniworld.

1.3 Characteristics of the Database Approach

A number of characteristics distinguish the database approach from the much older approach of writing customized programs to access data stored in files. In traditional **file processing**, each user defines and implements the files needed for a specific software application as part of programming the application. For example, one user, the *grade reporting office*, may keep files on students and their grades. Programs to print a student's transcript and to enter new grades are implemented as part of the application. A second user, the *accounting office*, may keep track of students' fees and their payments. Although both users are interested in data about students, each user maintains separate files—and programs to manipulate these files—because each requires some data not available from the other user's files. This redundancy in defining and storing data results in wasted storage space and in redundant efforts to maintain common up-to-date data.

In the database approach, a single repository maintains data that is defined once and then accessed by various users repeatedly through queries, transactions, and application programs. The main characteristics of the database approach versus the file-processing approach are the following:

- Self-describing nature of a database system
- Insulation between programs and data, and data abstraction
- Support of multiple views of the data
- Sharing of data and multiuser transaction processing

We describe each of these characteristics in a separate section. We will discuss additional characteristics of database systems in Sections 1.6 through 1.8.

1.3.1 Self-Describing Nature of a Database System

A fundamental characteristic of the database approach is that the database system contains not only the database itself but also a complete definition or description of the database structure and constraints. This definition is stored in the DBMS catalog, which contains information such as the structure of each file, the type and storage format of each data item, and various constraints on the data. The information stored in the catalog is called **meta-data**, and it describes the structure of the primary database (Figure 1.1). It is important to note that some newer types of database systems, known as NOSQL systems, do not require meta-data. Rather the data is stored as **self-describing data** that includes the data item names and data values together in one structure (see Chapter 24).

The catalog is used by the DBMS software and also by database users who need information about the database structure. A general-purpose DBMS software package is not written for a specific database application. Therefore, it must refer to the catalog to know the structure of the files in a specific database, such as the type and format of data it will access. The DBMS software must work equally well with *any number of database applications*—for example, a university database, a

banking database, or a company database—as long as the database definition is stored in the catalog.

In traditional file processing, data definition is typically part of the application programs themselves. Hence, these programs are constrained to work with only *one specific database*, whose structure is declared in the application programs. For example, an application program written in C++ may have struct or class declarations. Whereas file-processing software can access only specific databases, DBMS software can access diverse databases by extracting the database definitions from the catalog and using these definitions.

For the example shown in Figure 1.2, the DBMS catalog will store the definitions of all the files shown. Figure 1.3 shows some entries in a database catalog. Whenever a request is made to access, say, the Name of a STUDENT record, the DBMS software refers to the catalog to determine the structure of the STUDENT file and the position and size of the Name data item within a STUDENT record. By contrast, in a typical file-processing application, the file structure and, in the extreme case, the exact location of Name within a STUDENT record are already coded within each program that accesses this data item.

RELATIONS

Relation_name	No_of_columns
STUDENT	4
COURSE	4
SECTION	5
GRADE_REPORT	3
PREREQUISITE	2

Figure 1.3

An example of a database catalog for the database in Figure 1.2.

COLUMNS

Column_name	Data_type	Belongs_to_relation
Name	Character (30)	STUDENT
Student_number	Character (4)	STUDENT
Class	Integer (1)	STUDENT
Major	Major_type	STUDENT
Course_name	Character (10)	COURSE
Course_number	XXXXNNNN	COURSE
....
....
....
Prerequisite_number	XXXXNNNN	PREREQUISITE

Note: Major_type is defined as an enumerated type with all known majors.

XXXXNNNN is used to define a type with four alphabetic characters followed by four numeric digits.

1.3.2 Insulation between Programs and Data, and Data Abstraction

In traditional file processing, the structure of data files is embedded in the application programs, so any changes to the structure of a file may require *changing all programs* that access that file. By contrast, DBMS access programs do not require such changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property **program-data independence**.

For example, a file access program may be written in such a way that it can access only STUDENT records of the structure shown in Figure 1.4. If we want to add another piece of data to each STUDENT record, say the Birth_date, such a program will no longer work and must be changed. By contrast, in a DBMS environment, we only need to change the *description* of STUDENT records in the catalog (Figure 1.3) to reflect the inclusion of the new data item Birth_date; no programs are changed. The next time a DBMS program refers to the catalog, the new structure of STUDENT records will be accessed and used.

In some types of database systems, such as object-oriented and object-relational systems (see Chapter 12), users can define operations on data as part of the database definitions. An **operation** (also called a *function* or *method*) is specified in two parts. The *interface* (or *signature*) of an operation includes the operation name and the data types of its arguments (or parameters). The *implementation* (or *method*) of the operation is specified separately and can be changed without affecting the interface. User application programs can operate on the data by invoking these operations through their names and arguments, regardless of how the operations are implemented. This may be termed **program-operation independence**.

The characteristic that allows program-data independence and program-operation independence is called **data abstraction**. A DBMS provides users with a **conceptual representation** of data that does not include many of the details of how the data is stored or how the operations are implemented. Informally, a **data model** is a type of data abstraction that is used to provide this conceptual representation. The data model uses logical concepts, such as objects, their properties, and their interrelationships, that may be easier for most users to understand than computer storage concepts. Hence, the data model *hides* storage and implementation details that are not of interest to most database users.

Looking at the example in Figures 1.2 and 1.3, the internal implementation of the STUDENT file may be defined by its record length—the number of characters (bytes) in each record—and each data item may be specified by its starting byte within a record and its length in bytes. The STUDENT record would thus be represented as shown in Figure 1.4. But a typical database user is not concerned with the location of each data item within a record or its length; rather, the user is concerned that when a reference is made to Name of STUDENT, the correct value is returned. A conceptual representation of the STUDENT records is shown in Figure 1.2. Many other details of file storage organization—such as the access paths specified on a

Data Item Name	Starting Position in Record	Length in Characters (bytes)
Name	1	30
Student_number	31	4
Class	35	1
Major	36	4

Figure 1.4

Internal storage format for a STUDENT record, based on the database catalog in Figure 1.3.

file—can be hidden from database users by the DBMS; we discuss storage details in Chapters 16 and 17.

In the database approach, the detailed structure and organization of each file are stored in the catalog. Database users and application programs refer to the conceptual representation of the files, and the DBMS extracts the details of file storage from the catalog when these are needed by the DBMS file access modules. Many data models can be used to provide this data abstraction to database users. A major part of this text is devoted to presenting various data models and the concepts they use to abstract the representation of data.

In object-oriented and object-relational databases, the abstraction process includes not only the data structure but also the operations on the data. These operations provide an abstraction of miniworld activities commonly understood by the users. For example, an operation `CALCULATE_GPA` can be applied to a `STUDENT` object to calculate the grade point average. Such operations can be invoked by the user queries or application programs without having to know the details of how the operations are implemented.

1.3.3 Support of Multiple Views of the Data

A database typically has many types of users, each of whom may require a different perspective or **view** of the database. A view may be a subset of the database or it may contain **virtual data** that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether the data they refer to is stored or derived. A multiuser DBMS whose users have a variety of distinct applications must provide facilities for defining multiple views. For example, one user of the database of Figure 1.2 may be interested only in accessing and printing the transcript of each student; the view for this user is shown in Figure 1.5(a). A second user, who is interested only in checking that students have taken all the prerequisites of each course for which the student registers, may require the view shown in Figure 1.5(b).

1.3.4 Sharing of Data and Multiuser Transaction Processing

A multiuser DBMS, as its name implies, must allow multiple users to access the database at the same time. This is essential if data for multiple applications is to be integrated and maintained in a single database. The DBMS must include **concurrency control** software to ensure that several users trying to update the same data

TRANSCRIPT

Student_name	Student_transcript				
	Course_number	Grade	Semester	Year	Section_id
Smith	CS1310	C	Fall	08	119
	MATH2410	B	Fall	08	112
Brown	MATH2410	A	Fall	07	85
	CS1310	A	Fall	07	92
	CS3320	B	Spring	08	102
	CS3380	A	Fall	08	135

(a)

COURSE_PREREQUISITES

Course_name	Course_number	Prerequisites
Database	CS3380	CS3320
		MATH2410
Data Structures	CS3320	CS1310

(b)

Figure 1.5

Two views derived from the database in Figure 1.2. (a) The TRANSCRIPT view.

(b) The COURSE_PREREQUISITES view.

do so in a controlled manner so that the result of the updates is correct. For example, when several reservation agents try to assign a seat on an airline flight, the DBMS should ensure that each seat can be accessed by only one agent at a time for assignment to a passenger. These types of applications are generally called **online transaction processing (OLTP)** applications. A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly and efficiently.

The concept of a **transaction** has become central to many database applications. A transaction is an *executing program* or *process* that includes one or more database accesses, such as reading or updating of database records. Each transaction is supposed to execute a logically correct database access if executed in its entirety without interference from other transactions. The DBMS must enforce several transaction properties. The **isolation** property ensures that each transaction appears to execute in isolation from other transactions, even though hundreds of transactions may be executing concurrently. The **atomicity** property ensures that either all the database operations in a transaction are executed or none are. We discuss transactions in detail in Part 9.

The preceding characteristics are important in distinguishing a DBMS from traditional file-processing software. In Section 1.6 we discuss additional features that characterize a DBMS. First, however, we categorize the different types of people who work in a database system environment.

1.4 Actors on the Scene

For a small personal database, such as the list of addresses discussed in Section 1.1, one person typically defines, constructs, and manipulates the database, and there is no sharing. However, in large organizations, many people are involved in the design, use, and maintenance of a large database with hundreds or thousands of users. In this section we identify the people whose jobs involve the day-to-day use of a large database; we call them the *actors on the scene*. In Section 1.5 we consider people who may be called *workers behind the scene*—those who work to maintain the database system environment but who are not actively interested in the database contents as part of their daily job.

1.4.1 Database Administrators

In any organization where many people use the same resources, there is a need for a chief administrator to oversee and manage these resources. In a database environment, the primary resource is the database itself, and the secondary resource is the DBMS and related software. Administering these resources is the responsibility of the **database administrator (DBA)**. The DBA is responsible for authorizing access to the database, coordinating and monitoring its use, and acquiring software and hardware resources as needed. The DBA is accountable for problems such as security breaches and poor system response time. In large organizations, the DBA is assisted by a staff that carries out these functions.

1.4.2 Database Designers

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. These tasks are mostly undertaken before the database is actually implemented and populated with data. It is the responsibility of database designers to communicate with all prospective database users in order to understand their requirements and to create a design that meets these requirements. In many cases, the designers are on the staff of the DBA and may be assigned other staff responsibilities after the database design is completed. Database designers typically interact with each potential group of users and develop **views** of the database that meet the data and processing requirements of these groups. Each view is then analyzed and *integrated* with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

1.4.3 End Users

End users are the people whose jobs require access to the database for querying, updating, and generating reports; the database primarily exists for their use. There are several categories of end users:

- **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query interface

to specify their requests and are typically middle- or high-level managers or other occasional browsers.

- **Naive or parametric end users** make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called **canned transactions**—that have been carefully programmed and tested. Many of these tasks are now available as **mobile apps** for use with mobile devices. The tasks that such users perform are varied. A few examples are:
 - Bank customers and tellers check account balances and post withdrawals and deposits.
 - Reservation agents or customers for airlines, hotels, and car rental companies check availability for a given request and make reservations.
 - Employees at receiving stations for shipping companies enter package identifications via bar codes and descriptive information through buttons to update a central database of received and in-transit packages.
 - Social media users post and read items on social media Web sites.
- **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS in order to implement their own applications to meet their complex requirements.
- **Standalone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu-based or graphics-based interfaces. An example is the user of a financial software package that stores a variety of personal financial data.

A typical DBMS provides multiple facilities to access a database. Naive end users need to learn very little about the facilities provided by the DBMS; they simply have to understand the user interfaces of the mobile apps or standard transactions designed and implemented for their use. Casual users learn only a few facilities that they may use repeatedly. Sophisticated users try to learn most of the DBMS facilities in order to achieve their complex requirements. Standalone users typically become very proficient in using a specific software package.

1.4.4 System Analysts and Application Programmers (Software Engineers)

System analysts determine the requirements of end users, especially naive and parametric end users, and develop specifications for standard canned transactions that meet these requirements. **Application programmers** implement these specifications as programs; then they test, debug, document, and maintain these canned transactions. Such analysts and programmers—commonly referred to as **software developers** or **software engineers**—should be familiar with the full range of capabilities provided by the DBMS to accomplish their tasks.

1.5 Workers behind the Scene

In addition to those who design, use, and administer a database, others are associated with the design, development, and operation of the DBMS *software and system environment*. These persons are typically not interested in the database content itself. We call them the *workers behind the scene*, and they include the following categories:

- **DBMS system designers and implementers** design and implement the DBMS modules and interfaces as a software package. A DBMS is a very complex software system that consists of many components, or **modules**, including modules for implementing the catalog, query language processing, interface processing, accessing and buffering data, controlling concurrency, and handling data recovery and security. The DBMS must interface with other system software, such as the operating system and compilers for various programming languages.
- **Tool developers** design and implement **tools**—the software packages that facilitate database modeling and design, database system design, and improved performance. Tools are optional packages that are often purchased separately. They include packages for database design, performance monitoring, natural language or graphical interfaces, prototyping, simulation, and test data generation. In many cases, independent software vendors develop and market these tools.
- **Operators and maintenance personnel** (system administration personnel) are responsible for the actual running and maintenance of the hardware and software environment for the database system.

Although these categories of workers behind the scene are instrumental in making the database system available to end users, they typically do not use the database contents for their own purposes.

1.6 Advantages of Using the DBMS Approach

In this section we discuss some additional advantages of using a DBMS and the capabilities that a good DBMS should possess. These capabilities are in addition to the four main characteristics discussed in Section 1.3. The DBA must utilize these capabilities to accomplish a variety of objectives related to the design, administration, and use of a large multiuser database.

1.6.1 Controlling Redundancy

In traditional software development utilizing file processing, every user group maintains its own files for handling its data-processing applications. For example, consider the UNIVERSITY database example of Section 1.2; here, two groups of users might be the course registration personnel and the accounting office. In the traditional approach, each group independently keeps files on students. The

accounting office keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Other groups may further duplicate some or all of the same data in their own files.

This **redundancy** in storing the same data multiple times leads to several problems. First, there is the need to perform a single logical update—such as entering data on a new student—multiple times: once for each file where student data is recorded. This leads to *duplication of effort*. Second, *storage space is wasted* when the same data is stored repeatedly, and this problem may be serious for large databases. Third, files that represent the same data may become *inconsistent*. This may happen because an update is applied to some of the files but not to others. Even if an update—such as adding a new student—is applied to all the appropriate files, the data concerning the student may still be *inconsistent* because the updates are applied independently by each user group. For example, one user group may enter a student’s birth date erroneously as ‘JAN-19-1988’, whereas the other user groups may enter the correct value of ‘JAN-29-1988’.

In the database approach, the views of different user groups are integrated during database design. Ideally, we should have a database design that stores each logical data item—such as a student’s name or birth date—in *only one place* in the database. This is known as **data normalization**, and it ensures consistency and saves storage space (data normalization is described in Part 6 of the text).

However, in practice, it is sometimes necessary to use **controlled redundancy** to improve the performance of queries. For example, we may store Student_name and Course_number redundantly in a GRADE_REPORT file (Figure 1.6(a)) because whenever we retrieve a GRADE_REPORT record, we want to retrieve the student name and course number along with the grade, student number, and section identifier. By placing all the data together, we do not have to search multiple files to collect this data. This is known as **denormalization**. In such cases, the DBMS should

Figure 1.6
Redundant storage of Student_name and Course_name in GRADE_REPORT.
(a) Consistent data.
(b) Inconsistent record.

(a)	GRADE_REPORT				
	Student_number	Student_name	Section_identifier	Course_number	Grade
	17	Smith	112	MATH2410	B
	17	Smith	119	CS1310	C
	8	Brown	85	MATH2410	A
	8	Brown	92	CS1310	A
(b)	GRADE_REPORT				
	Student_number	Student_name	Section_identifier	Course_number	Grade
	17	Brown	112	MATH2410	B

have the capability to *control* this redundancy in order to prohibit inconsistencies among the files. This may be done by automatically checking that the Student_name–Student_number values in any GRADE_REPORT record in Figure 1.6(a) match one of the Name–Student_number values of a STUDENT record (Figure 1.2). Similarly, the Section_identifier–Course_number values in GRADE_REPORT can be checked against SECTION records. Such checks can be specified to the DBMS during database design and automatically enforced by the DBMS whenever the GRADE_REPORT file is updated. Figure 1.6(b) shows a GRADE_REPORT record that is inconsistent with the STUDENT file in Figure 1.2; this kind of error may be entered if the redundancy is *not controlled*. Can you tell which part is inconsistent?

1.6.2 Restricting Unauthorized Access

When multiple users share a large database, it is likely that most users will not be authorized to access all information in the database. For example, financial data such as salaries and bonuses is often considered confidential, and only authorized persons are allowed to access such data. In addition, some users may only be permitted to retrieve data, whereas others are allowed to retrieve and update. Hence, the type of access operation—retrieval or update—must also be controlled. Typically, users or user groups are given account numbers protected by passwords, which they can use to gain access to the database. A DBMS should provide a **security and authorization subsystem**, which the DBA uses to create accounts and to specify account restrictions. Then, the DBMS should enforce these restrictions automatically. Notice that we can apply similar controls to the DBMS software. For example, only the DBA's staff may be allowed to use certain **privileged software**, such as the software for creating new accounts. Similarly, parametric users may be allowed to access the database only through the pre-defined apps or canned transactions developed for their use. We discuss database security and authorization in Chapter 30.

1.6.3 Providing Persistent Storage for Program Objects

Databases can be used to provide **persistent storage** for program objects and data structures. This is one of the main reasons for **object-oriented database systems** (see Chapter 12). Programming languages typically have complex data structures, such as structs or class definitions in C++ or Java. The values of program variables or objects are discarded once a program terminates, unless the programmer explicitly stores them in permanent files, which often involves converting these complex structures into a format suitable for file storage. When the need arises to read this data once more, the programmer must convert from the file format to the program variable or object structure. Object-oriented database systems are compatible with programming languages such as C++ and Java, and the DBMS software automatically performs any necessary conversions. Hence, a complex object in C++ can be stored permanently in an object-oriented DBMS. Such an object is said to be **persistent**, since it survives the termination of program execution and can later be directly retrieved by another program.

The persistent storage of program objects and data structures is an important function of database systems. Traditional database systems often suffered from the so-called **impedance mismatch problem**, since the data structures provided by the DBMS were incompatible with the programming language's data structures. Object-oriented database systems typically offer data structure **compatibility** with one or more object-oriented programming languages.

1.6.4 Providing Storage Structures and Search Techniques for Efficient Query Processing

Database systems must provide capabilities for *efficiently executing queries and updates*. Because the database is typically stored on disk, the DBMS must provide specialized data structures and search techniques to speed up disk search for the desired records. Auxiliary files called **indexes** are often used for this purpose. Indexes are typically based on tree data structures or hash data structures that are suitably modified for disk search. In order to process the database records needed by a particular query, those records must be copied from disk to main memory. Therefore, the DBMS often has a **buffering** or **caching** module that maintains parts of the database in main memory buffers. In general, the operating system is responsible for disk-to-memory buffering. However, because data buffering is crucial to the DBMS performance, most DBMSs do their own data buffering.

The **query processing and optimization** module of the DBMS is responsible for choosing an efficient query execution plan for each query based on the existing storage structures. The choice of which indexes to create and maintain is part of *physical database design and tuning*, which is one of the responsibilities of the DBA staff. We discuss query processing and optimization in Part 8 of the text.

1.6.5 Providing Backup and Recovery

A DBMS must provide facilities for recovering from hardware or software failures. The **backup and recovery subsystem** of the DBMS is responsible for recovery. For example, if the computer system fails in the middle of a complex update transaction, the recovery subsystem is responsible for making sure that the database is restored to the state it was in before the transaction started executing. Disk backup is also necessary in case of a catastrophic disk failure. We discuss recovery and backup in Chapter 22.

1.6.6 Providing Multiple User Interfaces

Because many types of users with varying levels of technical knowledge use a database, a DBMS should provide a variety of user interfaces. These include apps for mobile users, query languages for casual users, programming language interfaces for application programmers, forms and command codes for parametric users, and menu-driven interfaces and natural language interfaces for standalone users. Both forms-style interfaces and menu-driven interfaces are commonly known as

graphical user interfaces (GUIs). Many specialized languages and environments exist for specifying GUIs. Capabilities for providing Web GUI interfaces to a database—or Web-enabling a database—are also quite common.

1.6.7 Representing Complex Relationships among Data

A database may include numerous varieties of data that are interrelated in many ways. Consider the example shown in Figure 1.2. The record for ‘Brown’ in the STUDENT file is related to four records in the GRADE_REPORT file. Similarly, each section record is related to one course record and to a number of GRADE_REPORT records—one for each student who completed that section. A DBMS must have the capability to represent a variety of complex relationships among the data, to define new relationships as they arise, and to retrieve and update related data easily and efficiently.

1.6.8 Enforcing Integrity Constraints

Most database applications have certain **integrity constraints** that must hold for the data. A DBMS should provide capabilities for defining and enforcing these constraints. The simplest type of integrity constraint involves specifying a data type for each data item. For example, in Figure 1.3, we specified that the value of the Class data item within each STUDENT record must be a one-digit integer and that the value of Name must be a string of no more than 30 alphabetic characters. To restrict the value of Class between 1 and 5 would be an additional constraint that is not shown in the current catalog. A more complex type of constraint that frequently occurs involves specifying that a record in one file must be related to records in other files. For example, in Figure 1.2, we can specify that *every section record must be related to a course record*. This is known as a **referential integrity** constraint. Another type of constraint specifies uniqueness on data item values, such as *every course record must have a unique value for Course_number*. This is known as a **key** or **uniqueness** constraint. These constraints are derived from the meaning or **semantics** of the data and of the miniworld it represents. It is the responsibility of the database designers to identify integrity constraints during database design. Some constraints can be specified to the DBMS and automatically enforced. Other constraints may have to be checked by update programs or at the time of data entry. For typical large applications, it is customary to call such constraints **business rules**.

A data item may be entered erroneously and still satisfy the specified integrity constraints. For example, if a student receives a grade of ‘A’ but a grade of ‘C’ is entered in the database, the DBMS *cannot* discover this error automatically because ‘C’ is a valid value for the Grade data type. Such data entry errors can only be discovered manually (when the student receives the grade and complains) and corrected later by updating the database. However, a grade of ‘Z’ would be rejected automatically by the DBMS because ‘Z’ is not a valid value for the Grade data type. When we discuss each data model in subsequent chapters, we will introduce rules that pertain to

that model implicitly. For example, in the Entity-Relationship model in Chapter 3, a relationship must involve at least two entities. Rules that pertain to a specific data model are called **inherent rules** of the data model.

1.6.9 Permitting Inferencing and Actions Using Rules and Triggers

Some database systems provide capabilities for defining *deduction rules* for *inferencing* new information from the stored database facts. Such systems are called **deductive database systems**. For example, there may be complex rules in the miniworld application for determining when a student is on probation. These can be specified *declaratively* as **rules**, which when compiled and maintained by the DBMS can determine all students on probation. In a traditional DBMS, an explicit *procedural program code* would have to be written to support such applications. But if the miniworld rules change, it is generally more convenient to change the declared deduction rules than to recode procedural programs. In today's relational database systems, it is possible to associate **triggers** with tables. A trigger is a form of a rule activated by updates to the table, which results in performing some additional operations to some other tables, sending messages, and so on. More involved procedures to enforce rules are popularly called **stored procedures**; they become a part of the overall database definition and are invoked appropriately when certain conditions are met. More powerful functionality is provided by **active database systems**, which provide active rules that can automatically initiate actions when certain events and conditions occur (see Chapter 26 for introductions to active databases in Section 26.1 and deductive databases in Section 26.5).

1.6.10 Additional Implications of Using the Database Approach

This section discusses a few additional implications of using the database approach that can benefit most organizations.

Potential for Enforcing Standards. The database approach permits the DBA to define and enforce standards among database users in a large organization. This facilitates communication and cooperation among various departments, projects, and users within the organization. Standards can be defined for names and formats of data elements, display formats, report structures, terminology, and so on. The DBA can enforce standards in a centralized database environment more easily than in an environment where each user group has control of its own data files and software.

Reduced Application Development Time. A prime selling feature of the database approach is that developing a new application—such as the retrieval of certain data from the database for printing a new report—takes very little time. Designing and implementing a large multiuser database from scratch may take more time than writing a single specialized file application. However, once a database is up and running, substantially less time is generally required to create new applications

using DBMS facilities. Development time using a DBMS is estimated to be one-sixth to one-fourth of that for a file system.

Flexibility. It may be necessary to change the structure of a database as requirements change. For example, a new user group may emerge that needs information not currently in the database. In response, it may be necessary to add a file to the database or to extend the data elements in an existing file. Modern DBMSs allow certain types of evolutionary changes to the structure of the database without affecting the stored data and the existing application programs.

Availability of Up-to-Date Information. A DBMS makes the database available to all users. As soon as one user's update is applied to the database, all other users can immediately see this update. This availability of up-to-date information is essential for many transaction-processing applications, such as reservation systems or banking databases, and it is made possible by the concurrency control and recovery subsystems of a DBMS.

Economies of Scale. The DBMS approach permits consolidation of data and applications, thus reducing the amount of wasteful overlap between activities of data-processing personnel in different projects or departments as well as redundancies among applications. This enables the whole organization to invest in more powerful processors, storage devices, or networking gear, rather than having each department purchase its own (lower performance) equipment. This reduces overall costs of operation and management.

1.7 A Brief History of Database Applications

We now give a brief historical overview of the applications that use DBMSs and how these applications provided the impetus for new types of database systems.

1.7.1 Early Database Applications Using Hierarchical and Network Systems

Many early database applications maintained records in large organizations such as corporations, universities, hospitals, and banks. In many of these applications, there were large numbers of records of similar structure. For example, in a university application, similar information would be kept for each student, each course, each grade record, and so on. There were also many types of records and many interrelationships among them.

One of the main problems with early database systems was the intermixing of conceptual relationships with the physical storage and placement of records on disk. Hence, these systems did not provide sufficient *data abstraction* and *program-data independence* capabilities. For example, the grade records of a particular student could be physically stored next to the student record. Although this provided very

efficient access for the original queries and transactions that the database was designed to handle, it did not provide enough flexibility to access records efficiently when new queries and transactions were identified. In particular, new queries that required a different storage organization for efficient processing were quite difficult to implement efficiently. It was also laborious to reorganize the database when changes were made to the application's requirements.

Another shortcoming of early systems was that they provided only programming language interfaces. This made it time-consuming and expensive to implement new queries and transactions, since new programs had to be written, tested, and debugged. Most of these database systems were implemented on large and expensive mainframe computers starting in the mid-1960s and continuing through the 1970s and 1980s. The main types of early systems were based on three main paradigms: hierarchical systems, network model-based systems, and inverted file systems.

1.7.2 Providing Data Abstraction and Application Flexibility with Relational Databases

Relational databases were originally proposed to separate the physical storage of data from its conceptual representation and to provide a mathematical foundation for data representation and querying. The relational data model also introduced high-level query languages that provided an alternative to programming language interfaces, making it much faster to write new queries. Relational representation of data somewhat resembles the example we presented in Figure 1.2. Relational systems were initially targeted to the same applications as earlier systems, and provided flexibility to develop new queries quickly and to reorganize the database as requirements changed. Hence, *data abstraction* and *program-data independence* were much improved when compared to earlier systems.

Early experimental relational systems developed in the late 1970s and the commercial relational database management systems (RDBMS) introduced in the early 1980s were quite slow, since they did not use physical storage pointers or record placement to access related data records. With the development of new storage and indexing techniques and better query processing and optimization, their performance improved. Eventually, relational databases became the dominant type of database system for traditional database applications. Relational databases now exist on almost all types of computers, from small personal computers to large servers.

1.7.3 Object-Oriented Applications and the Need for More Complex Databases

The emergence of object-oriented programming languages in the 1980s and the need to store and share complex, structured objects led to the development of object-oriented databases (OODBs). Initially, OODBs were considered a competitor

to relational databases, since they provided more general data structures. They also incorporated many of the useful object-oriented paradigms, such as abstract data types, encapsulation of operations, inheritance, and object identity. However, the complexity of the model and the lack of an early standard contributed to their limited use. They are now mainly used in specialized applications, such as engineering design, multimedia publishing, and manufacturing systems. Despite expectations that they will make a big impact, their overall penetration into the database products market remains low. In addition, many object-oriented concepts were incorporated into the newer versions of relational DBMSs, leading to object-relational database management systems, known as ORDBMSs.

1.7.4 Interchanging Data on the Web for E-Commerce Using XML

The World Wide Web provides a large network of interconnected computers. Users can create static Web pages using a Web publishing language, such as Hyper-Text Markup Language (HTML), and store these documents on Web servers where other users (clients) can access them and view them through Web browsers. Documents can be linked through **hyperlinks**, which are pointers to other documents. Starting in the 1990s, electronic commerce (e-commerce) emerged as a major application on the Web. Much of the critical information on e-commerce Web pages is dynamically extracted data from DBMSs, such as flight information, product prices, and product availability. A variety of techniques were developed to allow the interchange of dynamically extracted data on the Web for display on Web pages. The eXtended Markup Language (XML) is one standard for interchanging data among various types of databases and Web pages. XML combines concepts from the models used in document systems with database modeling concepts. Chapter 13 is devoted to an overview of XML.

1.7.5 Extending Database Capabilities for New Applications

The success of database systems in traditional applications encouraged developers of other types of applications to attempt to use them. Such applications traditionally used their own specialized software and file and data structures. Database systems now offer extensions to better support the specialized requirements for some of these applications. The following are some examples of these applications:

- **Scientific** applications that store large amounts of data resulting from scientific experiments in areas such as high-energy physics, the mapping of the human genome, and the discovery of protein structures
- Storage and retrieval of **images**, including scanned news or personal photographs, satellite photographic images, and images from medical procedures such as x-rays and MRI (magnetic resonance imaging) tests

- Storage and retrieval of **videos**, such as movies, and **video clips** from news or personal digital cameras
- **Data mining** applications that analyze large amounts of data to search for the occurrences of specific patterns or relationships, and for identifying unusual patterns in areas such as credit card fraud detection
- **Spatial** applications that store and analyze spatial locations of data, such as weather information, maps used in geographical information systems, and automobile navigational systems
- **Time series** applications that store information such as economic data at regular points in time, such as daily sales and monthly gross national product figures

It was quickly apparent that basic relational systems were not very suitable for many of these applications, usually for one or more of the following reasons:

- More complex data structures were needed for modeling the application than the simple relational representation.
- New data types were needed in addition to the basic numeric and character string types.
- New operations and query language constructs were necessary to manipulate the new data types.
- New storage and indexing structures were needed for efficient searching on the new data types.

This led DBMS developers to add functionality to their systems. Some functionality was general purpose, such as incorporating concepts from object-oriented databases into relational systems. Other functionality was special purpose, in the form of optional modules that could be used for specific applications. For example, users could buy a time series module to use with their relational DBMS for their time series application.

1.7.6 Emergence of Big Data Storage Systems and NOSQL Databases

In the first decade of the twenty-first century, the proliferation of applications and platforms such as social media Web sites, large e-commerce companies, Web search indexes, and cloud storage/backup led to a surge in the amount of data stored on large databases and massive servers. New types of database systems were necessary to manage these huge databases—systems that would provide fast search and retrieval as well as reliable and safe storage of nontraditional types of data, such as social media posts and tweets. Some of the requirements of these new systems were not compatible with SQL relational DBMSs (SQL is the standard data model and language for relational databases). The term *NOSQL* is generally interpreted as Not Only SQL, meaning that in systems that manage large amounts of data, some of the data is stored using SQL systems, whereas other data would be stored using NOSQL, depending on the application requirements.

1.8 When Not to Use a DBMS

In spite of the advantages of using a DBMS, there are a few situations in which a DBMS may involve unnecessary overhead costs that would not be incurred in traditional file processing. The overhead costs of using a DBMS are due to the following:

- High initial investment in hardware, software, and training
- The generality that a DBMS provides for defining and processing data
- Overhead for providing security, concurrency control, recovery, and integrity functions

Therefore, it may be more desirable to develop customized database applications under the following circumstances:

- Simple, well-defined database applications that are not expected to change at all
- Stringent, real-time requirements for some application programs that may not be met because of DBMS overhead
- Embedded systems with limited storage capacity, where a general-purpose DBMS would not fit
- No multiple-user access to data

Certain industries and applications have elected not to use general-purpose DBMSs. For example, many computer-aided design (CAD) tools used by mechanical and civil engineers have proprietary file and data management software that is geared for the internal manipulations of drawings and 3D objects. Similarly, communication and switching systems designed by companies like AT&T were early manifestations of database software that was made to run very fast with hierarchically organized data for quick access and routing of calls. GIS implementations often implement their own data organization schemes for efficiently implementing functions related to processing maps, physical contours, lines, polygons, and so on.

1.9 Summary

In this chapter we defined a database as a collection of related data, where *data* means recorded facts. A typical database represents some aspect of the real world and is used for specific purposes by one or more groups of users. A DBMS is a generalized software package for implementing and maintaining a computerized database. The database and software together form a database system. We identified several characteristics that distinguish the database approach from traditional file-processing applications, and we discussed the main categories of database users, or the *actors on the scene*. We noted that in addition to database users, there are several categories of support personnel, or *workers behind the scene*, in a database environment.

We presented a list of capabilities that should be provided by the DBMS software to the DBA, database designers, and end users to help them design, administer, and use a database. Then we gave a brief historical perspective on the evolution of database applications. We pointed out the recent rapid growth of the amounts and types of data that must be stored in databases, and we discussed the emergence of new systems for handling “big data” applications. Finally, we discussed the overhead costs of using a DBMS and discussed some situations in which it may not be advantageous to use one.

Review Questions

- 1.1. Define the following terms: *data*, *database*, *DBMS*, *database system*, *database catalog*, *program-data independence*, *user view*, *DBA*, *end user*, *canned transaction*, *deductive database system*, *persistent object*, *meta-data*, and *transaction-processing application*.
- 1.2. What four main types of actions involve databases? Briefly discuss each.
- 1.3. Discuss the main characteristics of the database approach and how it differs from traditional file systems.
- 1.4. What are the responsibilities of the DBA and the database designers?
- 1.5. What are the different types of database end users? Discuss the main activities of each.
- 1.6. Discuss the capabilities that should be provided by a DBMS.
- 1.7. Discuss the differences between database systems and information retrieval systems.

Exercises

- 1.8. Identify some informal queries and update operations that you would expect to apply to the database shown in Figure 1.2.
- 1.9. What is the difference between controlled and uncontrolled redundancy? Illustrate with examples.
- 1.10. Specify all the relationships among the records of the database shown in Figure 1.2.
- 1.11. Give some additional views that may be needed by other user groups for the database shown in Figure 1.2.
- 1.12. Cite some examples of integrity constraints that you think can apply to the database shown in Figure 1.2.
- 1.13. Give examples of systems in which it may make sense to use traditional file processing instead of a database approach.

1.14. Consider Figure 1.2.

- a. If the name of the 'CS' (Computer Science) Department changes to 'CSSE' (Computer Science and Software Engineering) Department and the corresponding prefix for the course number also changes, identify the columns in the database that would need to be updated.
- b. Can you restructure the columns in the COURSE, SECTION, and PREREQUISITE tables so that only one column will need to be updated?

Selected Bibliography

The October 1991 issue of *Communications of the ACM* and Kim (1995) include several articles describing next-generation DBMSs; many of the database features discussed in the former are now commercially available. The March 1976 issue of *ACM Computing Surveys* offers an early introduction to database systems and may provide a historical perspective for the interested reader. We will include references to other concepts, systems, and applications introduced in this chapter in the later text chapters that discuss each topic in more detail.

This page intentionally left blank

Database System Concepts and Architecture

The architecture of DBMS packages has evolved from the early monolithic systems, where the whole DBMS software package was one tightly integrated system, to the modern DBMS packages that are modular in design, with a client/server system architecture. The recent growth in the amount of data requiring storage has led to database systems with distributed architectures comprised of thousands of computers that manage the data stores. This evolution mirrors the trends in computing, where large centralized mainframe computers are replaced by hundreds of distributed workstations and personal computers connected via communications networks to various types of server machines—Web servers, database servers, file servers, application servers, and so on. The current **cloud computing** environments consist of thousands of large servers managing so-called **big data** for users on the Web.

In a basic client/server DBMS architecture, the system functionality is distributed between two types of modules.¹ A **client module** is typically designed so that it will run on a mobile device, user workstation, or personal computer (PC). Typically, application programs and user interfaces that access the database run in the client module. Hence, the client module handles user interaction and provides the user-friendly interfaces such as apps for mobile devices, or forms- or menu-based GUIs (graphical user interfaces) for PCs. The other kind of module, called a **server module**, typically handles data storage, access, search, and other functions. We discuss client/server architectures in more detail in Section 2.5. First, we must study more basic concepts that will give us a better understanding of modern database architectures.

¹As we shall see in Section 2.5, there are variations on this simple *two-tier* client/server architecture.

In this chapter we present the terminology and basic concepts that will be used throughout the text. Section 2.1 discusses data models and defines the concepts of schemas and instances, which are fundamental to the study of database systems. We discuss the three-schema DBMS architecture and data independence in Section 2.2; this provides a user's perspective on what a DBMS is supposed to do. In Section 2.3 we describe the types of interfaces and languages that are typically provided by a DBMS. Section 2.4 discusses the database system software environment. Section 2.5 gives an overview of various types of client/server architectures. Finally, Section 2.6 presents a classification of the types of DBMS packages. Section 2.7 summarizes the chapter.

The material in Sections 2.4 through 2.6 provides detailed concepts that may be considered as supplementary to the basic introductory material.

2.1 Data Models, Schemas, and Instances

One fundamental characteristic of the database approach is that it provides some level of data abstraction. **Data abstraction** generally refers to the suppression of details of data organization and storage, and the highlighting of the essential features for an improved understanding of data. One of the main characteristics of the database approach is to support data abstraction so that different users can perceive data at their preferred level of detail. A **data model**—a collection of concepts that can be used to describe the structure of a database—provides the necessary means to achieve this abstraction.² By *structure of a database* we mean the data types, relationships, and constraints that apply to the data. Most data models also include a set of **basic operations** for specifying retrievals and updates on the database.

In addition to the basic operations provided by the data model, it is becoming more common to include concepts in the data model to specify the **dynamic aspect** or **behavior** of a database application. This allows the database designer to specify a set of valid user-defined operations that are allowed on the database objects.³ An example of a user-defined operation could be `COMPUTE_GPA`, which can be applied to a `STUDENT` object. On the other hand, generic operations to insert, delete, modify, or retrieve any kind of object are often included in the *basic data model operations*. Concepts to specify behavior are fundamental to object-oriented data models (see Chapter 12) but are also being incorporated in more traditional data models. For example, object-relational models (see Chapter 12) extend the basic relational model to include such concepts, among others. In the basic relational data model, there is a provision to attach behavior to the relations in the form of persistent stored modules, popularly known as stored procedures (see Chapter 10).

²Sometimes the word *model* is used to denote a specific database description, or schema—for example, *the marketing data model*. We will not use this interpretation.

³The inclusion of concepts to describe behavior reflects a trend whereby database design and software design activities are increasingly being combined into a single activity. Traditionally, specifying behavior is associated with software design.

2.1.1 Categories of Data Models

Many data models have been proposed, which we can categorize according to the types of concepts they use to describe the database structure. **High-level** or **conceptual data models** provide concepts that are close to the way many users perceive data, whereas **low-level** or **physical data models** provide concepts that describe the details of how data is stored on the computer storage media, typically magnetic disks. Concepts provided by physical data models are generally meant for computer specialists, not for end users. Between these two extremes is a class of **representational** (or **implementation**) **data models**,⁴ which provide concepts that may be easily understood by end users but that are not too far removed from the way data is organized in computer storage. Representational data models hide many details of data storage on disk but can be implemented on a computer system directly.

Conceptual data models use concepts such as entities, attributes, and relationships. An **entity** represents a real-world object or concept, such as an employee or a project from the miniworld that is described in the database. An **attribute** represents some property of interest that further describes an entity, such as the employee's name or salary. A **relationship** among two or more entities represents an association among the entities, for example, a works-on relationship between an employee and a project. Chapter 3 presents the **entity-relationship model**—a popular high-level conceptual data model. Chapter 4 describes additional abstractions used for advanced modeling, such as generalization, specialization, and categories (union types).

Representational or implementation data models are the models used most frequently in traditional commercial DBMSs. These include the widely used **relational data model**, as well as the so-called legacy data models—the **network** and **hierarchical models**—that have been widely used in the past. Part 3 of the text is devoted to the relational data model, and its constraints, operations, and languages.⁵ The SQL standard for relational databases is described in Chapters 6 and 7. Representational data models represent data by using record structures and hence are sometimes called **record-based data models**.

We can regard the **object data model** as an example of a new family of higher-level implementation data models that are closer to conceptual data models. A standard for object databases called the ODMG object model has been proposed by the Object Data Management Group (ODMG). We describe the general characteristics of object databases and the object model proposed standard in Chapter 12. Object data models are also frequently utilized as high-level conceptual models, particularly in the software engineering domain.

Physical data models describe how data is stored as files in the computer by representing information such as record formats, record orderings, and access paths. An

⁴The term *implementation data model* is not a standard term; we have introduced it to refer to the available data models in commercial database systems.

⁵A summary of the hierarchical and network data models is included in Appendices D and E. They are accessible from the book's Web site.

access path is a search structure that makes the search for particular database records efficient, such as indexing or hashing. We discuss physical storage techniques and access structures in Chapters 16 and 17. An **index** is an example of an access path that allows direct access to data using an index term or a keyword. It is similar to the index at the end of this text, except that it may be organized in a linear, hierarchical (tree-structured), or some other fashion.

Another class of data models is known as **self-describing data models**. The data storage in systems based on these models combines the description of the data with the data values themselves. In traditional DBMSs, the description (schema) is separated from the data. These models include **XML** (see Chapter 12) as well as many of the **key-value stores** and **NOSQL systems** (see Chapter 24) that were recently created for managing big data.

2.1.2 Schemas, Instances, and Database State

In a data model, it is important to distinguish between the *description* of the database and the *database itself*. The description of a database is called the **database schema**, which is specified during database design and is not expected to change frequently.⁶ Most data models have certain conventions for displaying schemas as diagrams.⁷ A displayed schema is called a **schema diagram**. Figure 2.1 shows a schema diagram for the database shown in Figure 1.2; the diagram displays the structure of each record type but not the actual instances of records.

Figure 2.1

Schema diagram for the database in Figure 1.2.

STUDENT

Name	Student_number	Class	Major
------	----------------	-------	-------

COURSE

Course_name	Course_number	Credit_hours	Department
-------------	---------------	--------------	------------

PREREQUISITE

Course_number	Prerequisite_number
---------------	---------------------

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
--------------------	---------------	----------	------	------------

GRADE_REPORT

Student_number	Section_identifier	Grade
----------------	--------------------	-------

⁶Schema changes are usually needed as the requirements of the database applications change. Most database systems include operations for allowing schema changes.

⁷It is customary in database parlance to use *schemas* as the plural for *schema*, even though *schemata* is the proper plural form. The word *scheme* is also sometimes used to refer to a schema.

We call each object in the schema—such as STUDENT or COURSE—a **schema construct**.

A schema diagram displays only *some aspects* of a schema, such as the names of record types and data items, and some types of constraints. Other aspects are not specified in the schema diagram; for example, Figure 2.1 shows neither the data type of each data item nor the relationships among the various files. Many types of constraints are not represented in schema diagrams. A constraint such as *students majoring in computer science must take CS1310 before the end of their sophomore year* is quite difficult to represent diagrammatically.

The actual data in a database may change quite frequently. For example, the database shown in Figure 1.2 changes every time we add a new student or enter a new grade. The data in the database at a particular moment in time is called a **database state** or **snapshot**. It is also called the *current* set of **occurrences** or **instances** in the database. In a given database state, each schema construct has its own *current set* of instances; for example, the STUDENT construct will contain the set of individual student entities (records) as its instances. Many database states can be constructed to correspond to a particular database schema. Every time we insert or delete a record or change the value of a data item in a record, we change one state of the database into another state.

The distinction between database schema and database state is very important. When we **define** a new database, we specify its database schema only to the DBMS. At this point, the corresponding database state is the *empty state* with no data. We get the *initial state* of the database when the database is first **populated** or **loaded** with the initial data. From then on, every time an update operation is applied to the database, we get another database state. At any point in time, the database has a *current state*.⁸ The DBMS is partly responsible for ensuring that every state of the database is a **valid state**—that is, a state that satisfies the structure and constraints specified in the schema. Hence, specifying a correct schema to the DBMS is extremely important and the schema must be designed with utmost care. The DBMS stores the descriptions of the schema constructs and constraints—also called the **meta-data**—in the DBMS catalog so that DBMS software can refer to the schema whenever it needs to. The schema is sometimes called the **intension**, and a database state is called an **extension** of the schema.

Although, as mentioned earlier, the schema is not supposed to change frequently, it is not uncommon that changes occasionally need to be applied to the schema as the application requirements change. For example, we may decide that another data item needs to be stored for each record in a file, such as adding the *Date_of_birth* to the STUDENT schema in Figure 2.1. This is known as **schema evolution**. Most modern DBMSs include some operations for schema evolution that can be applied while the database is operational.

⁸The current state is also called the *current snapshot* of the database. It has also been called a *database instance*, but we prefer to use the term *instance* to refer to individual records.

2.2 Three-Schema Architecture and Data Independence

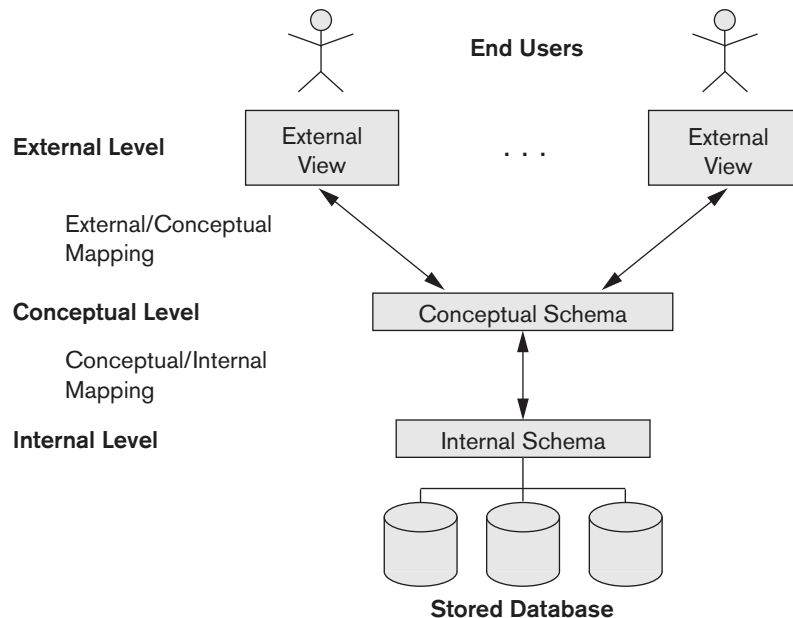
Three of the four important characteristics of the database approach, listed in Section 1.3, are (1) use of a catalog to store the database description (schema) so as to make it self-describing, (2) insulation of programs and data (program-data and program-operation independence), and (3) support of multiple user views. In this section we specify an architecture for database systems, called the **three-schema architecture**,⁹ that was proposed to help achieve and visualize these characteristics. Then we discuss further the concept of data independence.

2.2.1 The Three-Schema Architecture

The goal of the three-schema architecture, illustrated in Figure 2.2, is to separate the user applications from the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema**, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.

Figure 2.2
The three-schema architecture.



⁹This is also known as the ANSI/SPARC (American National Standards Institute/ Standards Planning And Requirements Committee) architecture, after the committee that proposed it (Tsichritzis & Klug, 1978).

2. The **conceptual level** has a **conceptual schema**, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. Usually, a representational data model is used to describe the conceptual schema when a database system is implemented. This *implementation conceptual schema* is often based on a *conceptual schema design* in a high-level data model.
3. The **external or view level** includes a number of **external schemas** or **user views**. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. As in the previous level, each external schema is typically implemented using a representational data model, possibly based on an external schema design in a high-level conceptual data model.

The three-schema architecture is a convenient tool with which the user can visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely and explicitly, but they support the three-schema architecture to some extent. Some older DBMSs may include physical-level details in the conceptual schema. The three-level ANSI architecture has an important place in database technology development because it clearly separates the users' external level, the database's conceptual level, and the internal storage level for designing a database. It is very much applicable in the design of DBMSs, even today. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information (for example, a relational DBMS like Oracle or SQLServer uses SQL for this).

Notice that the three schemas are only *descriptions* of data; the actual data is stored at the physical level only. In the three-schema architecture, each user group refers to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called **mappings**. These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, it is necessary to transform requests between the conceptual and internal levels.

2.2.2 Data Independence

The three-schema architecture can be used to further explain the concept of **data independence**, which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We

may change the conceptual schema to expand the database (by adding a record type or data item), to change constraints, or to reduce the database (by removing a record type or data item). In the last case, external schemas that refer only to the remaining data should not be affected. For example, the external schema of Figure 1.5(a) should not be affected by changing the `GRADE_REPORT` file (or record type) shown in Figure 1.2 into the one shown in Figure 1.6(a). Only the view definition and the mappings need to be changed in a DBMS that supports logical data independence. After the conceptual schema undergoes a logical reorganization, application programs that reference the external schema constructs must work as before. Changes to constraints can be applied to the conceptual schema without affecting the external schemas or application programs.

2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual schema. Hence, the external schemas need not be changed as well. Changes to the internal schema may be needed because some physical files were reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema. For example, providing an access path to improve retrieval speed of `SECTION` records (Figure 1.2) by semester and year should not require a query such as *list all sections offered in fall 2008* to be changed, although the query would be executed more efficiently by the DBMS by utilizing the new access path.

Generally, physical data independence exists in most databases and file environments where physical details, such as the exact location of data on disk, and hardware details of storage encoding, placement, compression, splitting, merging of records, and so on are hidden from the user. Applications remain unaware of these details. On the other hand, logical data independence is harder to achieve because it allows structural and constraint changes without affecting application programs—a much stricter requirement.

Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalog. Data independence occurs because when the schema is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed.

2.3 Database Languages and Interfaces

In Section 1.4 we discussed the variety of users supported by a DBMS. The DBMS must provide appropriate languages and interfaces for each category of users. In this section we discuss the types of languages and interfaces provided by a DBMS and the user categories targeted by each interface.

2.3.1 DBMS Languages

Once the design of a database is completed and a DBMS is chosen to implement the database, the first step is to specify conceptual and internal schemas for the database and any mappings between the two. In many DBMSs where no strict separation of levels is maintained, one language, called the **data definition language (DDL)**, is used by the DBA and by database designers to define both schemas. The DBMS will have a DDL compiler whose function is to process DDL statements in order to identify descriptions of the schema constructs and to store the schema description in the DBMS catalog.

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only. Another language, the **storage definition language (SDL)**, is used to specify the internal schema. The mappings between the two schemas may be specified in either one of these languages. In most relational DBMSs today, there *is no specific language* that performs the role of SDL. Instead, the internal schema is specified by a combination of functions, parameters, and specifications related to storage of files. These permit the DBA staff to control indexing choices and mapping of data to storage. For a true three-schema architecture, we would need a third language, the **view definition language (VDL)**, to specify user views and their mappings to the conceptual schema, but in most DBMSs *the DDL is used to define both conceptual and external schemas*. In relational DBMSs, SQL is used in the role of VDL to define user or application **views** as results of predefined queries (see Chapters 6 and 7).

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database. Typical manipulations include retrieval, insertion, deletion, and modification of the data. The DBMS provides a set of operations or a language called the **data manipulation language (DML)** for these purposes.

In current DBMSs, the preceding types of languages are usually *not considered distinct languages*; rather, a comprehensive integrated language is used that includes constructs for conceptual schema definition, view definition, and data manipulation. Storage definition is typically kept separate, since it is used for defining physical storage structures to fine-tune the performance of the database system, which is usually done by the DBA staff. A typical example of a comprehensive database language is the SQL relational database language (see Chapters 6 and 7), which represents a combination of DDL, VDL, and DML, as well as statements for constraint specification, schema evolution, and many other features. The SDL was a component in early versions of SQL but has been removed from the language to keep it at the conceptual and external levels only.

There are two main types of DMLs. A **high-level or nonprocedural** DML can be used on its own to specify complex database operations concisely. Many DBMSs allow high-level DML statements either to be entered interactively from a display monitor or terminal or to be embedded in a general-purpose programming language. In the latter case, DML statements must be identified within the program so

that they can be extracted by a precompiler and processed by the DBMS. A **low-level** or **procedural** DML *must* be embedded in a general-purpose programming language. This type of DML typically retrieves individual records or objects from the database and processes each separately. Therefore, it needs to use programming language constructs, such as looping, to retrieve and process each record from a set of records. Low-level DMLs are also called **record-at-a-time** DMLs because of this property. High-level DMLs, such as SQL, can specify and retrieve many records in a single DML statement; therefore, they are called **set-at-a-time** or **set-oriented** DMLs. A query in a high-level DML often specifies *which* data to retrieve rather than *how* to retrieve it; therefore, such languages are also called **declarative**.

Whenever DML commands, whether high level or low level, are embedded in a general-purpose programming language, that language is called the **host language** and the DML is called the **data sublanguage**.¹⁰ On the other hand, a high-level DML used in a standalone interactive manner is called a **query language**. In general, both retrieval and update commands of a high-level DML may be used interactively and are hence considered part of the query language.¹¹

Casual end users typically use a high-level query language to specify their requests, whereas programmers use the DML in its embedded form. For naive and parametric users, there usually are **user-friendly interfaces** for interacting with the database; these can also be used by casual users or others who do not want to learn the details of a high-level query language. We discuss these types of interfaces next.

2.3.2 DBMS Interfaces

User-friendly interfaces provided by a DBMS may include the following:

Menu-based Interfaces for Web Clients or Browsing. These interfaces present the user with lists of options (called **menus**) that lead the user through the formulation of a request. Menus do away with the need to memorize the specific commands and syntax of a query language; rather, the query is composed step-by-step by picking options from a menu that is displayed by the system. Pull-down menus are a very popular technique in **Web-based user interfaces**. They are also often used in **browsing interfaces**, which allow a user to look through the contents of a database in an exploratory and unstructured manner.

Apps for Mobile Devices. These interfaces present mobile users with access to their data. For example, banking, reservations, and insurance companies, among many others, provide apps that allow users to access their data through a mobile phone or mobile device. The apps have built-in programmed interfaces that typically

¹⁰In object databases, the host and data sublanguages typically form one integrated language—for example, C++ with some extensions to support database functionality. Some relational systems also provide integrated languages—for example, Oracle's PL/SQL.

¹¹According to the English meaning of the word *query*, it should really be used to describe retrievals only, not updates.

allow users to login using their account name and password; the apps then provide a limited menu of options for mobile access to the user data, as well as options such as paying bills (for banks) or making reservations (for reservation Web sites).

Forms-based Interfaces. A forms-based interface displays a form to each user. Users can fill out all of the **form** entries to insert new data, or they can fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Many DBMSs have **forms specification languages**, which are special languages that help programmers specify such forms. SQL*Forms is a form-based language that specifies queries using a form designed in conjunction with the relational database schema. Oracle Forms is a component of the Oracle product suite that provides an extensive set of features to design and build applications using forms. Some systems have utilities that define a form by letting the end user interactively construct a sample form on the screen.

Graphical User Interfaces. A GUI typically displays a schema to the user in diagrammatic form. The user then can specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms.

Natural Language Interfaces. These interfaces accept requests written in English or some other language and attempt to *understand* them. A natural language interface usually has its own *schema*, which is similar to the database conceptual schema, as well as a dictionary of important words. The natural language interface refers to the words in its schema, as well as to the set of standard words in its dictionary, that are used to interpret the request. If the interpretation is successful, the interface generates a high-level query corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request.

Keyword-based Database Search. These are somewhat similar to Web search engines, which accept strings of natural language (like English or Spanish) words and match them with documents at specific sites (for local search engines) or Web pages on the Web at large (for engines like Google or Ask). They use predefined indexes on words and use ranking functions to retrieve and present resulting documents in a decreasing degree of match. Such “free form” textual query interfaces are not yet common in structured relational databases, although a research area called **keyword-based querying** has emerged recently for relational databases.

Speech Input and Output. Limited use of speech as an input query and speech as an answer to a question or result of a request is becoming commonplace. Applications with limited vocabularies, such as inquiries for telephone directory, flight arrival/departure, and credit card account information, are allowing speech for input and output to enable customers to access this information. The speech input is detected using a library of predefined words and used to set up the parameters that are supplied to the queries. For output, a similar conversion from text or numbers into speech takes place.

Interfaces for Parametric Users. Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. For example, a teller is able to use single function keys to invoke routine and repetitive transactions such as account deposits or withdrawals, or balance inquiries. Systems analysts and programmers design and implement a special interface for each known class of naive users. Usually a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request.

Interfaces for the DBA. Most database systems contain privileged commands that can be used only by the DBA staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

2.4 The Database System Environment

A DBMS is a complex software system. In this section we discuss the types of software components that constitute a DBMS and the types of computer system software with which the DBMS interacts.

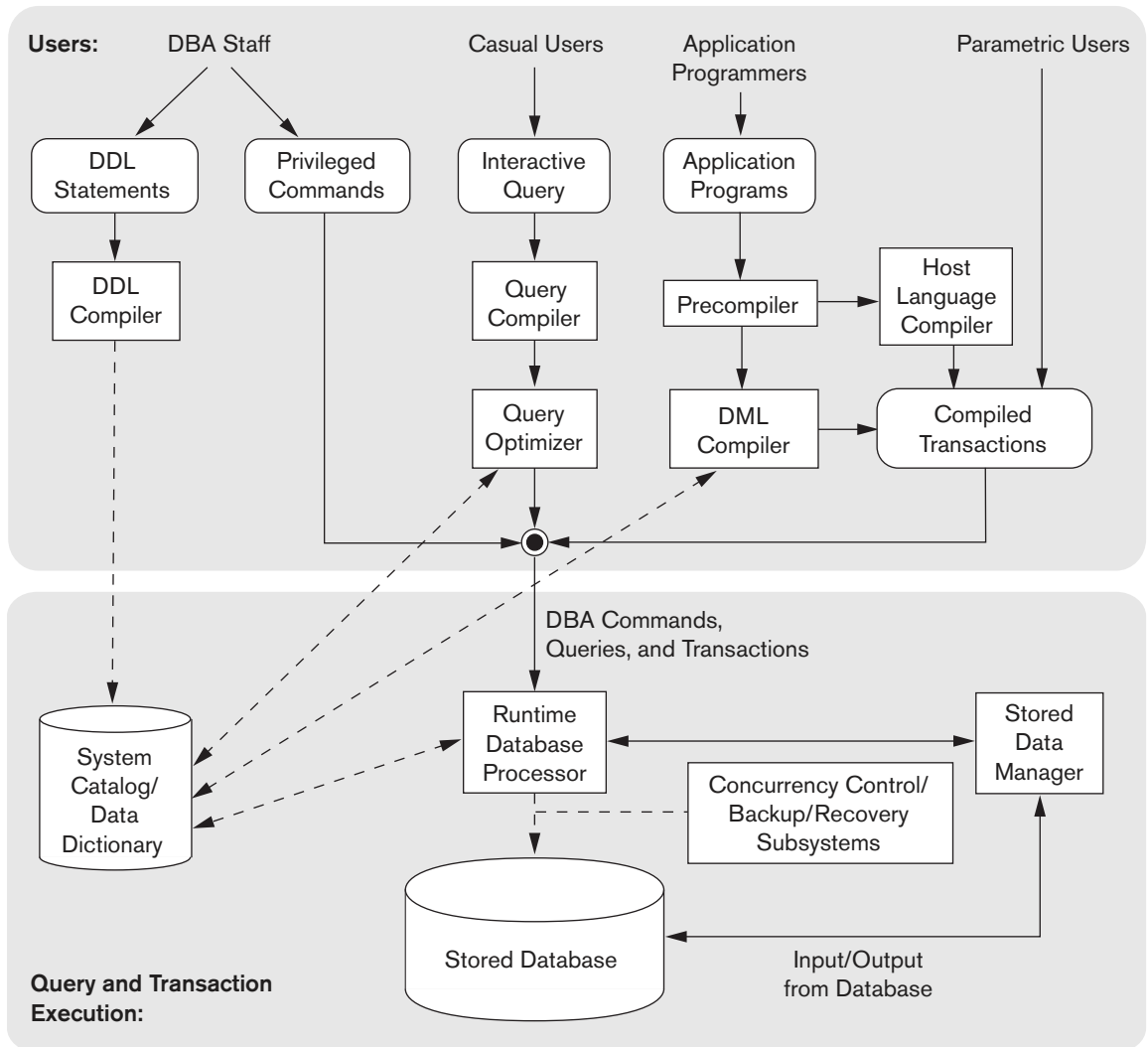
2.4.1 DBMS Component Modules

Figure 2.3 illustrates, in a simplified form, the typical DBMS components. The figure is divided into two parts. The top part of the figure refers to the various users of the database environment and their interfaces. The lower part shows the internal modules of the DBMS responsible for storage of data and processing of transactions.

The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **operating system (OS)**, which schedules disk read/write. Many DBMSs have their own **buffer management** module to schedule disk read/write, because management of buffer storage has a considerable effect on performance. Reducing disk read/write improves performance considerably. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.

Let us consider the top part of Figure 2.3 first. It shows interfaces for the DBA staff, casual users who work with interactive interfaces to formulate queries, application programmers who create programs using some host programming languages, and parametric users who do data entry work by supplying parameters to predefined transactions. The DBA staff works on defining the database and tuning it by making changes to its definition using the DDL and other privileged commands.

The DDL compiler processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names and sizes of files, names and data types of data items, storage details of each file, mapping information among schemas, and constraints.

**Figure 2.3**

Component modules of a DBMS and their interactions.

In addition, the catalog stores many other types of information that are needed by the DBMS modules, which can then look up the catalog information as needed.

Casual users and persons with occasional need for information from the database interact using the **interactive query** interface in Figure 2.3. We have *not explicitly shown* any menu-based or form-based or mobile interactions that are typically used to generate the interactive query automatically or to access canned transactions. These queries are parsed and validated for correctness of the query syntax, the names of files and data elements, and so on by a **query compiler** that compiles

them into an internal form. This internal query is subjected to query optimization (discussed in Chapters 18 and 19). Among other things, the **query optimizer** is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of efficient search algorithms during execution. It consults the system catalog for statistical and other physical information about the stored data and generates executable code that performs the necessary operations for the query and makes calls on the runtime processor.

Application programmers write programs in host languages such as Java, C, or C++ that are submitted to a precompiler. The **precompiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the DML compiler for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor. It is also becoming increasingly common to use scripting languages such as PHP and Python to write database programs. Canned transactions are executed repeatedly by parametric users via PCs or mobile apps; these users simply supply the parameters to the transactions. Each execution is considered to be a separate transaction. An example is a bank payment transaction where the account number, payee, and amount may be supplied as parameters.

In the lower part of Figure 2.3, the **runtime database processor** executes (1) the privileged commands, (2) the executable query plans, and (3) the canned transactions with runtime parameters. It works with the **system catalog** and may update it with statistics. It also works with the **stored data manager**, which in turn uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory. The runtime database processor handles other aspects of data transfer, such as management of buffers in the main memory. Some DBMSs have their own buffer management module whereas others depend on the OS for buffer management. We have shown **concurrency control** and **backup and recovery systems** separately as a module in this figure. They are integrated into the working of the runtime database processor for purposes of transaction management.

It is common to have the **client program** that accesses the DBMS running on a separate computer or device from the computer on which the database resides. The former is called the **client computer** running DBMS client software and the latter is called the **database server**. In many cases, the client accesses a middle computer, called the **application server**, which in turn accesses the database server. We elaborate on this topic in Section 2.5.

Figure 2.3 is not meant to describe a specific DBMS; rather, it illustrates typical DBMS modules. The DBMS interacts with the operating system when disk accesses—to the database or to the catalog—are needed. If the computer system is shared by many users, the OS will schedule DBMS disk access requests and DBMS processing along with other processes. On the other hand, if the computer system is mainly dedicated to running the database server, the DBMS will control main memory

buffering of disk pages. The DBMS also interfaces with compilers for general-purpose host programming languages, and with application servers and client programs running on separate machines through the system network interface.

2.4.2 Database System Utilities

In addition to possessing the software modules just described, most DBMSs have **database utilities** that help the DBA manage the database system. Common utilities have the following types of functions:

- **Loading.** A loading utility is used to load existing data files—such as text files or sequential files—into the database. Usually, the current (source) format of the data file and the desired (target) database file structure are specified to the utility, which then automatically reformats the data and stores it in the database. With the proliferation of DBMSs, transferring data from one DBMS to another is becoming common in many organizations. Some vendors offer **conversion tools** that generate the appropriate loading programs, given the existing source and target database storage descriptions (internal schemas).
- **Backup.** A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape or other mass storage medium. The backup copy can be used to restore the database in case of catastrophic disk failure. Incremental backups are also often used, where only changes since the previous backup are recorded. Incremental backup is more complex, but saves storage space.
- **Database storage reorganization.** This utility can be used to reorganize a set of database files into different file organizations and create new access paths to improve performance.
- **Performance monitoring.** Such a utility monitors database usage and provides statistics to the DBA. The DBA uses the statistics in making decisions such as whether or not to reorganize files or whether to add or drop indexes to improve performance.

Other utilities may be available for sorting files, handling data compression, monitoring access by users, interfacing with the network, and performing other functions.

2.4.3 Tools, Application Environments, and Communications Facilities

Other tools are often available to database designers, users, and the DBMS. CASE tools¹² are used in the design phase of database systems. Another tool that can be quite useful in large organizations is an expanded **data dictionary** (or **data repository**)

¹²Although CASE stands for computer-aided software engineering, many CASE tools are used primarily for database design.

system. In addition to storing catalog information about schemas and constraints, the data dictionary stores other information, such as design decisions, usage standards, application program descriptions, and user information. Such a system is also called an **information repository**. This information can be accessed *directly* by users or the DBA when needed. A data dictionary utility is similar to the DBMS catalog, but it includes a wider variety of information and is accessed mainly by users rather than by the DBMS software.

Application development environments, such as PowerBuilder (Sybase) or JBuilder (Borland), have been quite popular. These systems provide an environment for developing database applications and include facilities that help in many facets of database systems, including database design, GUI development, querying and updating, and application program development.

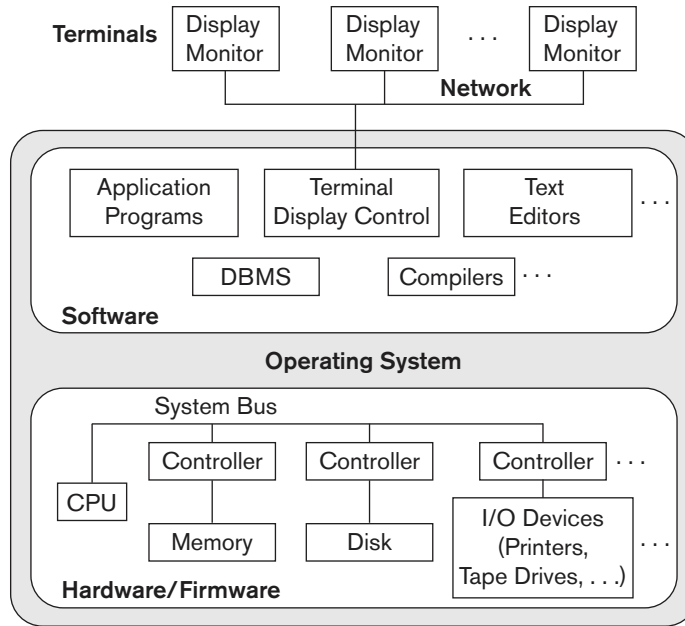
The DBMS also needs to interface with **communications software**, whose function is to allow users at locations remote from the database system site to access the database through computer terminals, workstations, or personal computers. These are connected to the database site through data communications hardware such as Internet routers, phone lines, long-haul networks, local networks, or satellite communication devices. Many commercial database systems have communication packages that work with the DBMS. The integrated DBMS and data communications system is called a **DB/DC** system. In addition, some distributed DBMSs are physically distributed over multiple machines. In this case, communications networks are needed to connect the machines. These are often **local area networks (LANs)**, but they can also be other types of networks.

2.5 Centralized and Client/Server Architectures for DBMSs

2.5.1 Centralized DBMSs Architecture

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Older architectures used mainframe computers to provide the main processing for all system functions, including user application programs and user interface programs, as well as all the DBMS functionality. The reason was that in older systems, most users accessed the DBMS via computer terminals that did not have processing power and only provided display capabilities. Therefore, all processing was performed remotely on the computer system housing the DBMS, and only display information and controls were sent from the computer to the display terminals, which were connected to the central computer via various types of communications networks.

As prices of hardware declined, most users replaced their terminals with PCs and workstations, and more recently with mobile devices. At first, database systems used these computers similarly to how they had used display terminals, so that the DBMS itself was still a **centralized** DBMS in which all the DBMS functionality,

**Figure 2.4**

A physical centralized architecture.

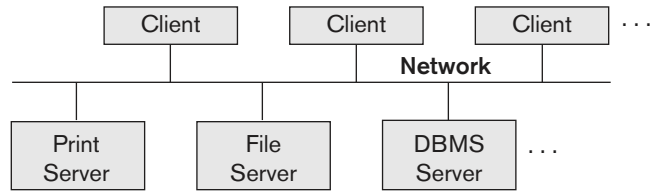
application program execution, and user interface processing were carried out on one machine. Figure 2.4 illustrates the physical components in a centralized architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

2.5.2 Basic Client/Server Architectures

First, we discuss client/server architecture in general; then we discuss how it is applied to DBMSs. The **client/server architecture** was developed to deal with computing environments in which a large number of PCs, workstations, file servers, printers, database servers, Web servers, e-mail servers, and other software and equipment are connected via a network. The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines. Another machine can be designated as a **printer server** by being connected to various printers; all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** also fall into the specialized server category. The resources provided by specialized servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. This concept can be carried over to other software packages, with specialized programs—such as a CAD (computer-aided design) package—being stored on specific server machines and being made accessible to multiple clients. Figure 2.5 illustrates

Figure 2.5

Logical two-tier client/server architecture.

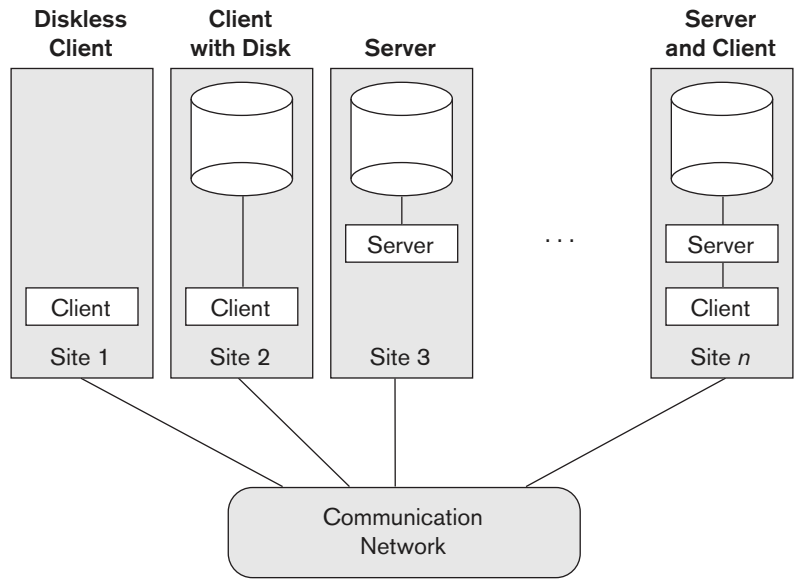


client/server architecture at the logical level; Figure 2.6 is a simplified diagram that shows the physical architecture. Some machines would be client sites only (for example, mobile devices or workstations/PCs that have only client software installed). Other machines would be dedicated servers, and others would have both client and server functionality.

The concept of client/server architecture assumes an underlying framework that consists of many PCs/workstations and mobile devices as well as a smaller number of server machines, connected via wireless networks or LANs and other types of computer networks. A **client** in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality—such as database access—that does not exist at the client, it connects to a server that provides the needed functionality. A **server** is a system containing both hardware and software that can provide services to the client machines, such as file access, printing, archiving, or database access. In general, some machines install only client software, others only server software, and still others may include both client and server software, as illustrated in Figure 2.6. However, it is more common that client and server software usually run on separate

Figure 2.6

Physical two-tier client/server architecture.



machines. Two main types of basic DBMS architectures were created on this underlying client/server framework: **two-tier** and **three-tier**.¹³ We discuss them next.

2.5.3 Two-Tier Client/Server Architectures for DBMSs

In relational database management systems (RDBMSs), many of which started as centralized systems, the system components that were first moved to the client side were the user interface and application programs. Because SQL (see Chapters 6 and 7) provided a standard language for RDBMSs, this created a logical dividing point between client and server. Hence, the query and transaction functionality related to SQL processing remained on the server side. In such an architecture, the server is often called a **query server** or **transaction server** because it provides these two functionalities. In an RDBMS, the server is also often called an **SQL server**.

The user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS. A standard called **Open Database Connectivity (ODBC)** provides an **application programming interface (API)**, which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed. Most DBMS vendors provide ODBC drivers for their systems. A client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are then processed at the server sites. Any query results are sent back to the client program, which can process and display the results as needed. A related standard for the Java programming language, called **JDBC**, has also been defined. This allows Java client programs to access one or more DBMSs through a standard interface.

The architectures described here are called **two-tier architectures** because the software components are distributed over two systems: client and server. The advantages of this architecture are its simplicity and seamless compatibility with existing systems. The emergence of the Web changed the roles of clients and servers, leading to the three-tier architecture.

2.5.4 Three-Tier and n -Tier Architectures for Web Applications

Many Web applications use an architecture called the **three-tier architecture**, which adds an intermediate layer between the client and the database server, as illustrated in Figure 2.7(a).

¹³There are many other variations of client/server architectures. We discuss the two most basic ones here.

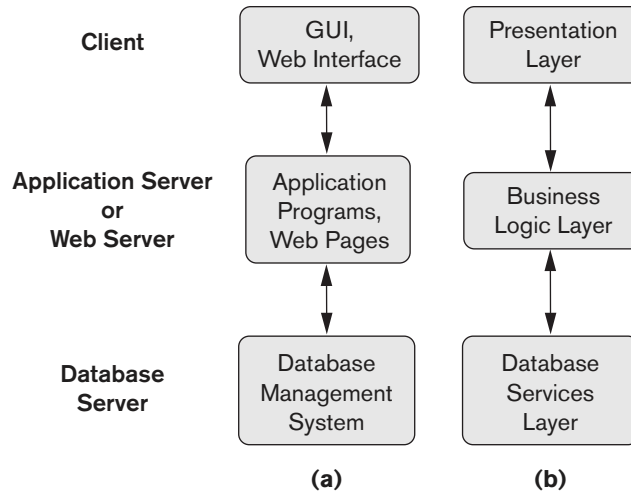


Figure 2.7
Logical three-tier
client/server
architecture, with a
couple of commonly
used nomenclatures.

This intermediate layer or **middle tier** is called the **application server** or the **Web server**, depending on the application. This server plays an intermediary role by running application programs and storing business rules (procedures or constraints) that are used to access data from the database server. It can also improve database security by checking a client's credentials before forwarding a request to the database server. Clients contain user interfaces and Web browsers. The intermediate server accepts requests from the client, processes the request and sends database queries and commands to the database server, and then acts as a conduit for passing (partially) processed data from the database server to the clients, where it may be processed further and filtered to be presented to the users. Thus, the *user interface*, *application rules*, and *data access* act as the three tiers. Figure 2.7(b) shows another view of the three-tier architecture used by database and other application package vendors. The presentation layer displays information to the user and allows data entry. The business logic layer handles intermediate rules and constraints before data is passed up to the user or down to the DBMS. The bottom layer includes all data management services. The middle layer can also act as a Web server, which retrieves query results from the database server and formats them into dynamic Web pages that are viewed by the Web browser at the client side. The client machine is typically a PC or mobile device connected to the Web.

Other architectures have also been proposed. It is possible to divide the layers between the user and the stored data further into finer components, thereby giving rise to n -tier architectures, where n may be four or five tiers. Typically, the business logic layer is divided into multiple layers. Besides distributing programming and data throughout a network, n -tier applications afford the advantage that any one tier can run on an appropriate processor or operating system platform and can be handled independently. Vendors of ERP (enterprise resource planning) and CRM (customer relationship management) packages often use a *middleware layer*, which

accounts for the front-end modules (clients) communicating with a number of back-end databases (servers).

Advances in encryption and decryption technology make it safer to transfer sensitive data from server to client in encrypted form, where it will be decrypted. The latter can be done by the hardware or by advanced software. This technology gives higher levels of data security, but the network security issues remain a major concern. Various technologies for data compression also help to transfer large amounts of data from servers to clients over wired and wireless networks.

2.6 Classification of Database Management Systems

Several criteria can be used to classify DBMSs. The first is the **data model** on which the DBMS is based. The main data model used in many current commercial DBMSs is the **relational data model**, and the systems based on this model are known as **SQL systems**. The **object data model** has been implemented in some commercial systems but has not had widespread use. Recently, so-called **big data systems**, also known as **key-value storage systems** and **NOSQL systems**, use various data models: **document-based**, **graph-based**, **column-based**, and **key-value data models**. Many legacy applications still run on database systems based on the **hierarchical** and **network data models**.

The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs called **object-relational DBMSs**. We can categorize DBMSs based on the data model: relational, object, object-relational, NOSQL, key-value, hierarchical, network, and other.

Some experimental DBMSs are based on the XML (eXtended Markup Language) model, which is a **tree-structured data model**. These have been called **native XML DBMSs**. Several commercial relational DBMSs have added XML interfaces and storage to their products.

The second criterion used to classify DBMSs is the **number of users** supported by the system. **Single-user systems** support only one user at a time and are mostly used with PCs. **Multiuser systems**, which include the majority of DBMSs, support concurrent multiple users.

The third criterion is the **number of sites** over which the database is distributed. A DBMS is **centralized** if the data is stored at a single computer site. A centralized DBMS can support multiple users, but the DBMS and the database reside totally at a single computer site. A **distributed** DBMS (DDBMS) can have the actual database and DBMS software distributed over many sites connected by a computer network. Big data systems are often massively distributed, with hundreds of sites. The data is often replicated on multiple sites so that failure of a site will not make some data unavailable.

Homogeneous DDBMSs use the same DBMS software at all the sites, whereas **heterogeneous** DDBMSs can use different DBMS software at each site. It is also possible to develop **middleware software** to access several autonomous preexisting databases stored under heterogeneous DBMSs. This leads to a **federated** DBMS (or **multidatabase system**), in which the participating DBMSs are loosely coupled and have a degree of local autonomy. Many DDBMSs use client-server architecture, as we described in Section 2.5.

The fourth criterion is cost. It is difficult to propose a classification of DBMSs based on cost. Today we have open source (free) DBMS products like MySQL and PostgreSQL that are supported by third-party vendors with additional services. The main RDBMS products are available as free examination 30-day copy versions as well as personal versions, which may cost under \$100 and allow a fair amount of functionality. The giant systems are being sold in modular form with components to handle distribution, replication, parallel processing, mobile capability, and so on, and with a large number of parameters that must be defined for the configuration. Furthermore, they are sold in the form of licenses—site licenses allow unlimited use of the database system with any number of copies running at the customer site. Another type of license limits the number of concurrent users or the number of user seats at a location. Standalone single-user versions of some systems like Microsoft Access are sold per copy or included in the overall configuration of a desktop or laptop. In addition, data warehousing and mining features, as well as support for additional data types, are made available at extra cost. It is possible to pay millions of dollars for the installation and maintenance of large database systems annually.

We can also classify a DBMS on the basis of the **types of access path** options for storing files. One well-known family of DBMSs is based on inverted file structures. Finally, a DBMS can be **general purpose** or **special purpose**. When performance is a primary consideration, a special-purpose DBMS can be designed and built for a specific application; such a system cannot be used for other applications without major changes. Many airline reservations and telephone directory systems developed in the past are special-purpose DBMSs. These fall into the category of **online transaction processing (OLTP)** systems, which must support a large number of concurrent transactions without imposing excessive delays.

Let us briefly elaborate on the main criterion for classifying DBMSs: the data model. The **relational data model** represents a database as a collection of tables, where each table can be stored as a separate file. The database in Figure 1.2 resembles a basic relational representation. Most relational databases use the high-level query language called SQL and support a limited form of user views. We discuss the relational model and its languages and operations in Chapters 5 through 8, and techniques for programming relational applications in Chapters 10 and 11.

The **object data model** defines a database in terms of objects, their properties, and their operations. Objects with the same structure and behavior belong to a **class**, and classes are organized into **hierarchies** (or **acyclic graphs**). The operations of

each class are specified in terms of predefined procedures called **methods**. Relational DBMSs have been extending their models to incorporate object database concepts and other capabilities; these systems are referred to as **object-relational** or **extended relational systems**. We discuss object databases and object-relational systems in Chapter 12.

Big data systems are based on various data models, with the following four data models most common. The **key-value data model** associates a unique key with each value (which can be a record or object) and provides very fast access to a value given its key. The **document data model** is based on JSON (Java Script Object Notation) and stores the data as documents, which somewhat resemble complex objects. The **graph data model** stores objects as graph nodes and relationships among objects as directed graph edges. Finally, the **column-based data models** store the columns of rows clustered on disk pages for fast access and allow multiple versions of the data. We will discuss some of these in more detail in Chapter 24.

The **XML model** has emerged as a standard for exchanging data over the Web and has been used as a basis for implementing several prototype native XML systems. XML uses hierarchical tree structures. It combines database concepts with concepts from document representation models. Data is represented as elements; with the use of tags, data can be nested to create complex tree structures. This model conceptually resembles the object model but uses different terminology. XML capabilities have been added to many commercial DBMS products. We present an overview of XML in Chapter 13.

Two older, historically important data models, now known as **legacy data models**, are the network and hierarchical models. The **network model** represents data as record types and also represents a limited type of 1:N relationship, called a **set type**. A 1:N, or one-to-many, relationship relates one instance of a record to many record instances using some pointer linking mechanism in these models. The network model, also known as the CODASYL DBTG model,¹⁴ has an associated record-at-a-time language that must be embedded in a host programming language. The network DML was proposed in the 1971 Database Task Group (DBTG) Report as an extension of the COBOL language.

The **hierarchical model** represents data as hierarchical tree structures. Each hierarchy represents a number of related records. There is no standard language for the hierarchical model. A popular hierarchical DML is DL/1 of the IMS system. It dominated the DBMS market for over 20 years between 1965 and 1985. Its DML, called DL/1, was a de facto industry standard for a long time.¹⁵

¹⁴CODASYL DBTG stands for Conference on Data Systems Languages Database Task Group, which is the committee that specified the network model and its language.

¹⁵The full chapters on the network and hierarchical models from the second edition of this book are available from this book's Companion Web site at <http://www.aw.com/elmasri>.

2.7 Summary

In this chapter we introduced the main concepts used in database systems. We defined a data model and we distinguished three main categories:

- High-level or conceptual data models (based on entities and relationships)
- Low-level or physical data models
- Representational or implementation data models (record-based, object-oriented)

We distinguished the schema, or description of a database, from the database itself. The schema does not change very often, whereas the database state changes every time data is inserted, deleted, or modified. Then we described the three-schema DBMS architecture, which allows three schema levels:

- An internal schema describes the physical storage structure of the database.
- A conceptual schema is a high-level description of the whole database.
- External schemas describe the views of different user groups.

A DBMS that cleanly separates the three levels must have mappings among the schemas to transform requests and query results from one level to the next. Most DBMSs do not separate the three levels completely. We used the three-schema architecture to define the concepts of logical and physical data independence.

Then we discussed the main types of languages and interfaces that DBMSs support. A data definition language (DDL) is used to define the database conceptual schema. In most DBMSs, the DDL also defines user views and, sometimes, storage structures; in other DBMSs, separate languages or functions exist for specifying storage structures. This distinction is fading away in today's relational implementations, with SQL serving as a catchall language to perform multiple roles, including view definition. The storage definition part (SDL) was included in SQL's early versions, but is now typically implemented as special commands for the DBA in relational DBMSs. The DBMS compiles all schema definitions and stores their descriptions in the DBMS catalog.

A data manipulation language (DML) is used for specifying database retrievals and updates. DMLs can be high level (set-oriented, nonprocedural) or low level (record-oriented, procedural). A high-level DML can be embedded in a host programming language, or it can be used as a standalone language; in the latter case it is often called a query language.

We discussed different types of interfaces provided by DBMSs and the types of DBMS users with which each interface is associated. Then we discussed the database system environment, typical DBMS software modules, and DBMS utilities for helping users and the DBA staff perform their tasks. We continued with an overview of the two-tier and three-tier architectures for database applications.

Finally, we classified DBMSs according to several criteria: data model, number of users, number of sites, types of access paths, and cost. We discussed the availability of DBMSs and additional modules—from no cost in the form of open source software to configurations that annually cost millions to maintain. We also pointed out the variety of licensing arrangements for DBMS and related products. The main classification of DBMSs is based on the data model. We briefly discussed the main data models used in current commercial DBMSs.

Review Questions

- 2.1. Define the following terms: *data model*, *database schema*, *database state*, *internal schema*, *conceptual schema*, *external schema*, *data independence*, *DDL*, *DML*, *SDL*, *VDL*, *query language*, *host language*, *data sublanguage*, *database utility*, *catalog*, *client/server architecture*, *three-tier architecture*, and *n-tier architecture*.
- 2.2. Discuss the main categories of data models. What are the basic differences among the relational model, the object model, and the XML model?
- 2.3. What is the difference between a database schema and a database state?
- 2.4. Describe the three-schema architecture. Why do we need mappings among schema levels? How do different schema definition languages support this architecture?
- 2.5. What is the difference between logical data independence and physical data independence? Which one is harder to achieve? Why?
- 2.6. What is the difference between procedural and nonprocedural DMLs?
- 2.7. Discuss the different types of user-friendly interfaces and the types of users who typically use each.
- 2.8. With what other computer system software does a DBMS interact?
- 2.9. What is the difference between the two-tier and three-tier client/server architectures?
- 2.10. Discuss some types of database utilities and tools and their functions.
- 2.11. What is the additional functionality incorporated in n -tier architecture ($n > 3$)?

Exercises

- 2.12. Think of different users for the database shown in Figure 1.2. What types of applications would each user need? To which user category would each belong, and what type of interface would each need?

- 2.13.** Choose a database application with which you are familiar. Design a schema and show a sample database for that application, using the notation of Figures 1.2 and 2.1. What types of additional information and constraints would you like to represent in the schema? Think of several users of your database, and design a view for each.
- 2.14.** If you were designing a Web-based system to make airline reservations and sell airline tickets, which DBMS architecture would you choose from Section 2.5? Why? Why would the other architectures not be a good choice?
- 2.15.** Consider Figure 2.1. In addition to constraints relating the values of columns in one table to columns in another table, there are also constraints that impose restrictions on values in a column or a combination of columns within a table. One such constraint dictates that a column or a group of columns must be unique across all rows in the table. For example, in the STUDENT table, the Student_number column must be unique (to prevent two different students from having the same Student_number). Identify the column or the group of columns in the other tables that must be unique across all rows in the table.

Selected Bibliography

Many database textbooks, including Date (2004), Silberschatz et al. (2011), Ramakrishnan and Gehrke (2003), Garcia-Molina et al. (2002, 2009), and Abiteboul et al. (1995), provide a discussion of the various database concepts presented here. Tsichritzis and Lochovsky (1982) is an early textbook on data models. Tsichritzis and Klug (1978) and Jardine (1977) present the three-schema architecture, which was first suggested in the DBTG CODASYL report (1971) and later in an American National Standards Institute (ANSI) report (1975). An in-depth analysis of the relational data model and some of its possible extensions is given in Codd (1990). The proposed standard for object-oriented databases is described in Cattell et al. (2000). Many documents describing XML are available on the Web, such as XML (2005).

Examples of database utilities are the ETI Connect, Analyze and Transform tools (<http://www.eti.com>) and the database administration tool, DBArtisan, from Embarcadero Technologies (<http://www.embarcadero.com>).