chapter 1

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. New media technology 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** (**GIS**) 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 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, 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. You may have recorded this data in an indexed address book or you may have stored it 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 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 inter-

[1]We 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.

ested 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 is maintained by the Internal Revenue Service (IRS) to monitor tax forms filed by U.S. taxpayers. If we assume that there are 100 million taxpayers and each taxpayer files an average of five forms with approximately 400 characters of information per form, we would have a database of $100 \times 10^6 \times 400 \times 5$ characters (bytes) of information. If the IRS keeps the past three returns of each taxpayer in addition to the current return, we would have a database of 8×10^{11} bytes (800 gigabytes). This huge amount of information must be organized and managed so that users can search for, retrieve, and update the data as needed.

An example of a large commercial database is Amazon.com. It contains data for over 20 million books, CDs, videos, DVDs, games, electronics, apparel, and other items. The database occupies over 2 terabytes (a terabyte is 10^{12} bytes worth of storage) and is stored on 200 different computers (called servers). About 15 million 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. About 100 people are responsible for keeping the Amazon database up-to-date.

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. We are only concerned with computerized databases in this book.

A **database management system** (**DBMS**) is a collection of programs 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. We could write our own set of programs to create and maintain the database, in effect creating our own *special-purpose* DBMS software. In either case—whether we use a general-purpose DBMS or not—we usually have to deploy a considerable amount of complex software. 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 for such a database. The database is organized as five files, each of which 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

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

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

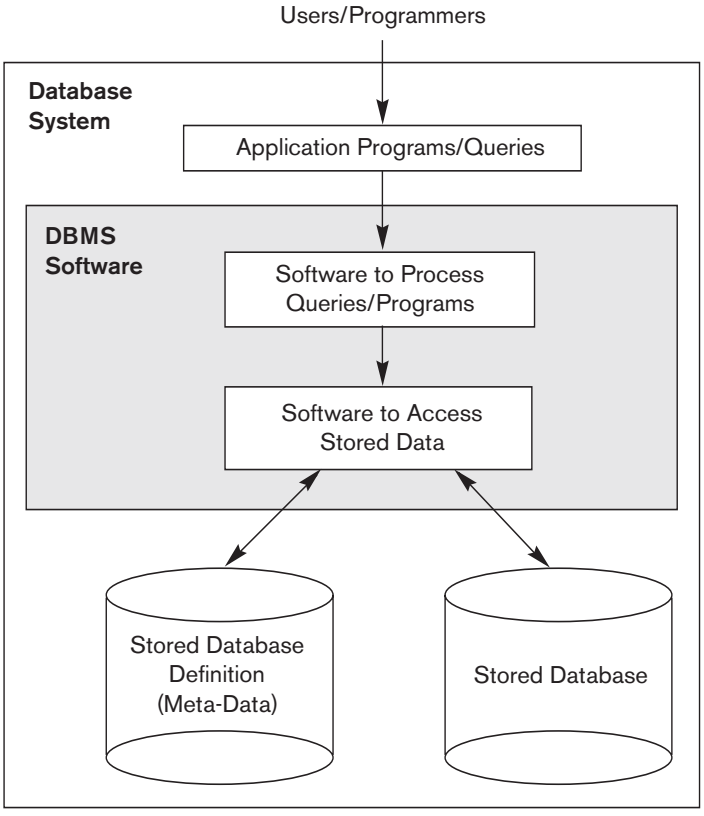


Figure 1.1
A simplified database system environment.

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 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.

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.

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 a part of a larger undertaking known as an information system within any organization. The Information Technology (IT) department within a company 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 7 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. (In this book we will emphasize a data model known as the Relational Data Model from Chapter 3 onward. This is currently the most popular approach for designing and implementing databases using relational DBMSs.) 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 programming with 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 avail-

able 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. In file systems, each application is free to name data elements independently. In contrast, in a database, the names or labels of data are defined once, and used repeatedly by queries, transactions, and applications. 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).

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, and a COBOL program has data division statements to define its files. 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 sample entries in a database catalog.

These definitions are specified by the database designer prior to creating the actual database and are stored in the 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.

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**.

RELATIONS

Relation_name	No_of_columns
STUDENT	4
COURSE	4
SECTION	5
GRADE_REPORT	3
PREREQUISITE	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 alpha characters followed by four digits.

Figure 1.3
An example of a database catalog for the database in Figure 1.2.

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 11), 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.

For example, reconsider Figures 1.2 and 1.3. The internal implementation of a 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 file—can be hidden from database users by the DBMS; we discuss storage details in Chapters 17 and 18.

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.

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 book 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. In that sense, an abstraction of the miniworld activity is made available to the user as an **abstract operation**.

1.3.3 Support of Multiple Views of the Data

A database typically has many 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 they register, 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 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

(a)

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

(b)

COURSE_PREREQUISITES

Course_name	Course_number	Prerequisites
Database	CS3380	CS3320
		MATH2410
Data Structures	CS3320	CS1310

Figure 1.5
Two views derived from the database in Figure 1.2. (a) The TRANSCRIPT view. (b) The COURSE_PREREQUISITES view.

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 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 language 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. The tasks that such users perform are varied:
  - Bank tellers check account balances and post withdrawals and deposits.
  - Reservation agents 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.

- **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 tax package that stores a variety of personal financial data for tax purposes.

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 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 of the 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'.

- 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. Similarly, GIS implementations often implement their own data organization schemes for efficiently implementing functions related to processing maps, physical contours, lines, polygons, and so on. General-purpose DBMSs are inadequate for their purpose.

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 marriage of database technology with information retrieval technology, which will play an important role due to the popularity of the Web. 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.

chapter 2

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. This evolution mirrors the trends in computing, where large centralized mainframe computers are being 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.

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 user workstation or personal computer. 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 forms- or menu-based GUIs (graphical user interfaces). 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.

In this chapter we present the terminology and basic concepts that will be used throughout the book. Section 2.1 discusses data models and defines the concepts of schemas and instances, which are fundamental to the study of database systems. Then, 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.

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

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 more 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 11) but are also being incorporated in more traditional data models. For example, object-relational models (see Chapter 11) 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 13).

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

[2]Sometimes 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.

[3]The 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.

magnetic disks. Concepts provided by low-level 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 7 presents the **Entity-Relationship model**—a popular high-level conceptual data model. Chapter 8 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 2 is devoted to the relational data model, and its constraints, operations and languages.[5] The SQL standard for relational databases is described in Chapters 4 and 5. 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 11. 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 **access path** is a structure that makes the search for particular database records efficient. We discuss physical storage techniques and access structures in Chapters 17 and 18. 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 book, except that it may be organized in a linear, hierarchical (tree-structured), or some other fashion.

---

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

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

2.1.2 Schemas, Instances, and Database State

In any 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. 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

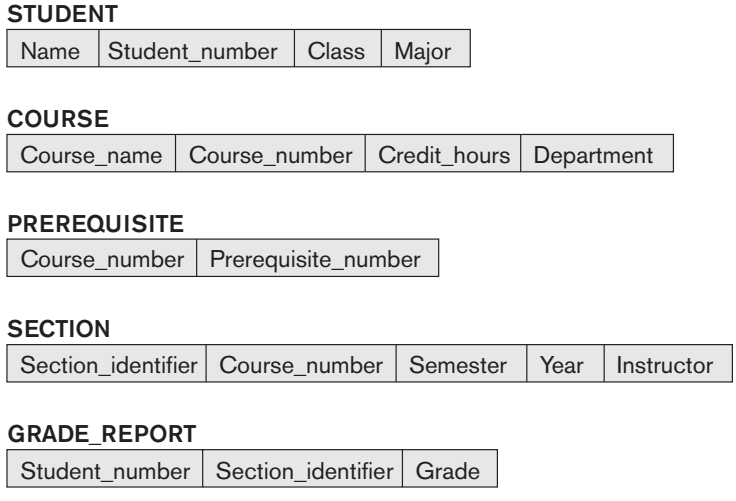


Figure 2.1
Schema diagram for the database in Figure 1.2.

[6]Schema changes are usually needed as the requirements of the database applications change. Newer database systems include operations for allowing schema changes, although the schema change process is more involved than simple database updates.

[7]It 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.

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.

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 the concept of data independence further.

---

[8]The 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.

[9]This is also known as the ANSI/SPARC architecture, after the committee that proposed it (Tsichritzis and Klug 1978).

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.
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 data model.

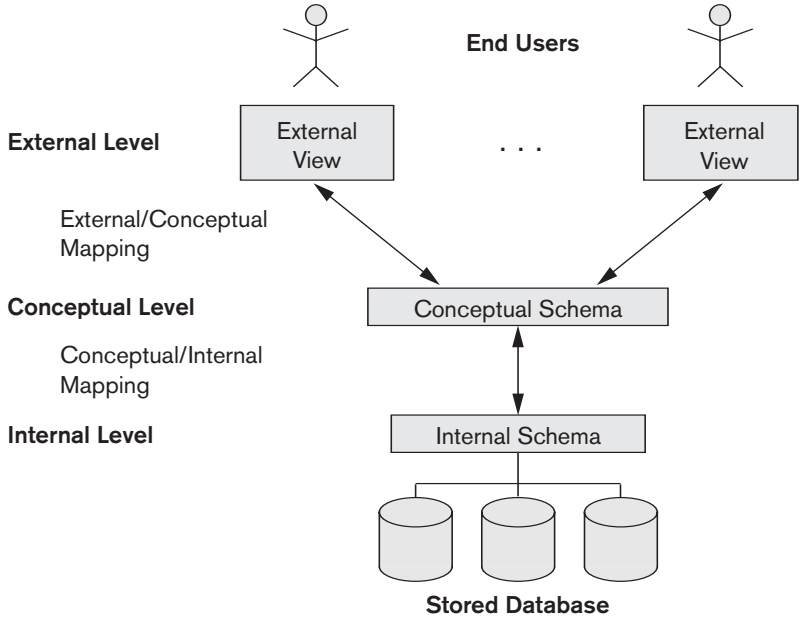


Figure 2.2
The three-schema architecture.

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 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 uses SQL for this). Some DBMSs allow different data models to be used at the conceptual and external levels. An example is Universal Data Base (UDB), a DBMS from IBM, which uses the relational model to describe the conceptual schema, but may use an object-oriented model to describe an external schema.

Notice that the three schemas are only *descriptions* of data; the stored data that *actually* exists is at the physical level only. In a DBMS based on 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, a certain amount of mapping 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.

The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

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

function keys in a terminal can be programmed to initiate various commands. This allows the parametric user to proceed with a minimal number of keystrokes.

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 internals 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 this 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. 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 some form of interface, which we call the **interactive query** interface in Figure 2.3. We have not explicitly shown any menu-based or form-based interaction that may be used to generate the interactive query automatically. These queries are parsed and validated for correctness of the query syntax, the names of files and

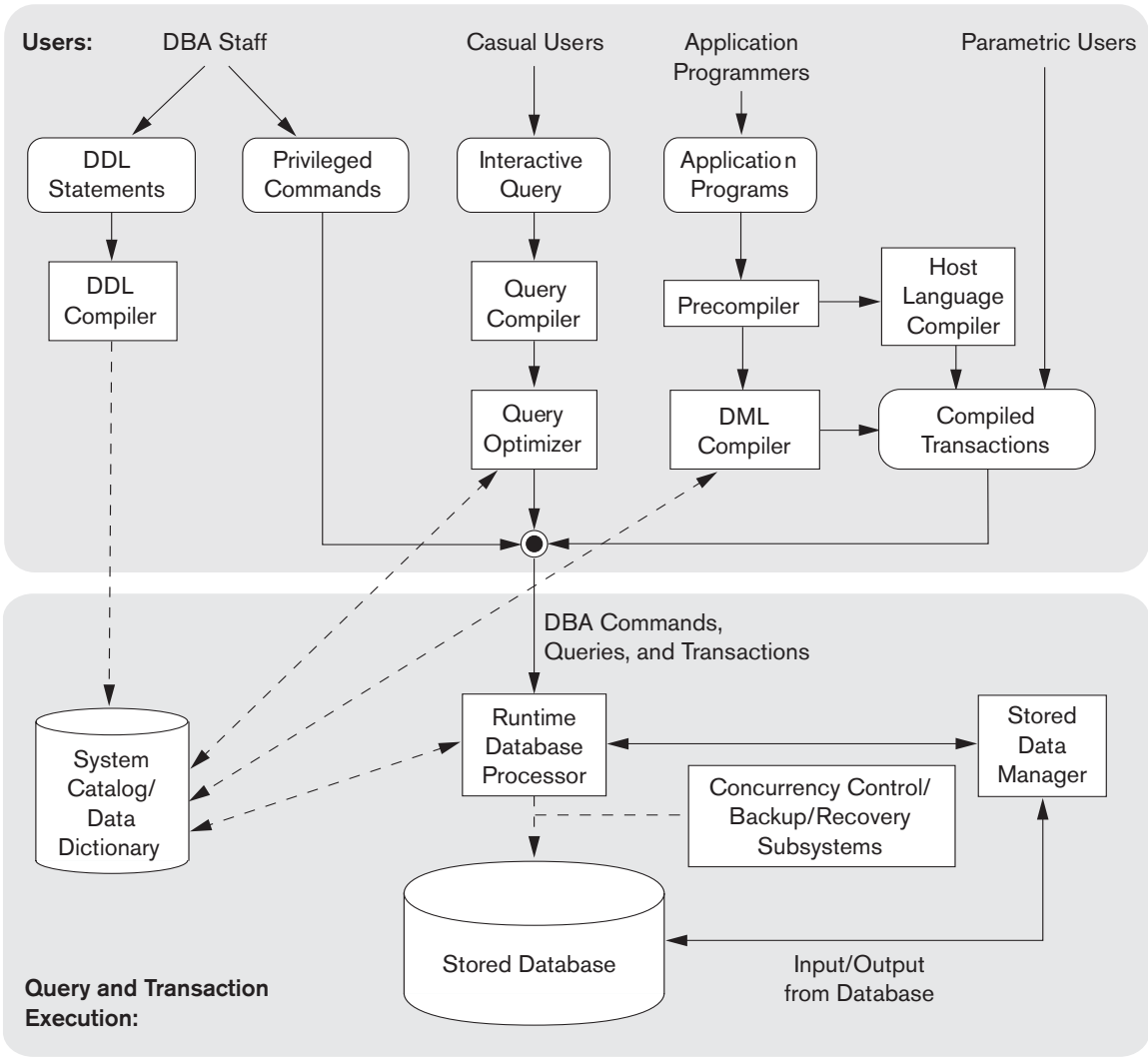


Figure 2.3
Component modules of a DBMS and their interactions.

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 19 and 20). Among other things, the **query optimizer** is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of correct algorithms and indexes 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. Canned transactions are executed repeatedly by parametric users, who simply supply the parameters to the transactions. Each execution is considered to be a separate transaction. An example is a bank withdrawal transaction where the account number and the 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 while 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 now common to have the **client program** that accesses the DBMS running on a separate computer from the computer on which the database resides. The former is called the **client computer** running a DBMS client software and the latter is called the **database server**. In some 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) for-

mat 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 are offering products that generate the appropriate loading programs, given the existing source and target database storage descriptions (internal schemas). Such tools are also called **conversion tools**. For the hierarchical DBMS called IMS (IBM) and for many network DBMSs including IDMS (Computer Associates), SUPRA (Cincom), and IMAGE (HP), the vendors or third-party companies are making a variety of conversion tools available (e.g., Cincom's SUPRA Server SQL) to transform data into the relational model.

- **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 tools[12] 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**) **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.

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

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. Earlier 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 most users accessed such systems via computer terminals that did not have processing power and only provided display capabilities. Therefore, all processing was performed remotely on the computer system, 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. 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, 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 see 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, data-

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, progressively moving toward *n*-tier, which are now common in many applications, particularly Web 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 between 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 between 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.

chapter 7

Data Modeling Using the Entity-Relationship (ER) Model

Conceptual modeling is a very important phase in designing a successful database application. Generally, the term **database application** refers to a particular database and the associated programs that implement the database queries and updates. For example, a BANK database application that keeps track of customer accounts would include programs that implement database updates corresponding to customer deposits and withdrawals. These programs provide user-friendly graphical user interfaces (GUIs) utilizing forms and menus for the end users of the application—the bank tellers, in this example. Hence, a major part of the database application will require the design, implementation, and testing of these application programs. Traditionally, the design and testing of **application programs** has been considered to be part of *software engineering* rather than *database design*. In many software design tools, the database design methodologies and software engineering methodologies are intertwined since these activities are strongly related.

In this chapter, we follow the traditional approach of concentrating on the database structures and constraints during conceptual database design. The design of application programs is typically covered in software engineering courses. We present the modeling concepts of the **Entity-Relationship** (**ER**) **model**, which is a popular high-level conceptual data model. This model and its variations are frequently used for the conceptual design of database applications, and many database design tools employ its concepts. We describe the basic data-structuring concepts and constraints of the ER model and discuss their use in the design of conceptual schemas for database applications. We also present the diagrammatic notation associated with the ER model, known as **ER diagrams**.

Object modeling methodologies such as the **Unified Modeling Language** (**UML**) are becoming increasingly popular in both database and software design. These methodologies go beyond database design to specify detailed design of software modules and their interactions using various types of diagrams. An important part of these methodologies—namely, *class diagrams*[1]—are similar in many ways to the ER diagrams. In class diagrams, *operations* on objects are specified, in addition to specifying the database schema structure. Operations can be used to specify the *functional requirements* during database design, as we will discuss in Section 7.1. We present some of the UML notation and concepts for class diagrams that are particularly relevant to database design in Section 7.8, and briefly compare these to ER notation and concepts. Additional UML notation and concepts are presented in Section 8.6 and in Chapter 10.

This chapter is organized as follows: Section 7.1 discusses the role of high-level conceptual data models in database design. We introduce the requirements for a sample database application in Section 7.2 to illustrate the use of concepts from the ER model. This sample database is also used throughout the book. In Section 7.3 we present the concepts of entities and attributes, and we gradually introduce the diagrammatic technique for displaying an ER schema. In Section 7.4 we introduce the concepts of binary relationships and their roles and structural constraints. Section 7.5 introduces weak entity types. Section 7.6 shows how a schema design is refined to include relationships. Section 7.7 reviews the notation for ER diagrams, summarizes the issues and common pitfalls that occur in schema design, and discusses how to choose the names for database schema constructs. Section 7.8 introduces some UML class diagram concepts, compares them to ER model concepts, and applies them to the same database example. Section 7.9 discusses more complex types of relationships. Section 7.10 summarizes the chapter.

The material in Sections 7.8 and 7.9 may be excluded from an introductory course. If a more thorough coverage of data modeling concepts and conceptual database design is desired, the reader should continue to Chapter 8, where we describe extensions to the ER model that lead to the Enhanced-ER (EER) model, which includes concepts such as specialization, generalization, inheritance, and union types (categories). We also introduce some additional UML concepts and notation in Chapter 8.

7.1 Using High-Level Conceptual Data Models for Database Design

Figure 7.1 shows a simplified overview of the database design process. The first step shown is **requirements collection and analysis**. During this step, the database designers interview prospective database users to understand and document their **data requirements**. The result of this step is a concisely written set of users' requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify

[1]A **class** is similar to an *entity type* in many ways.

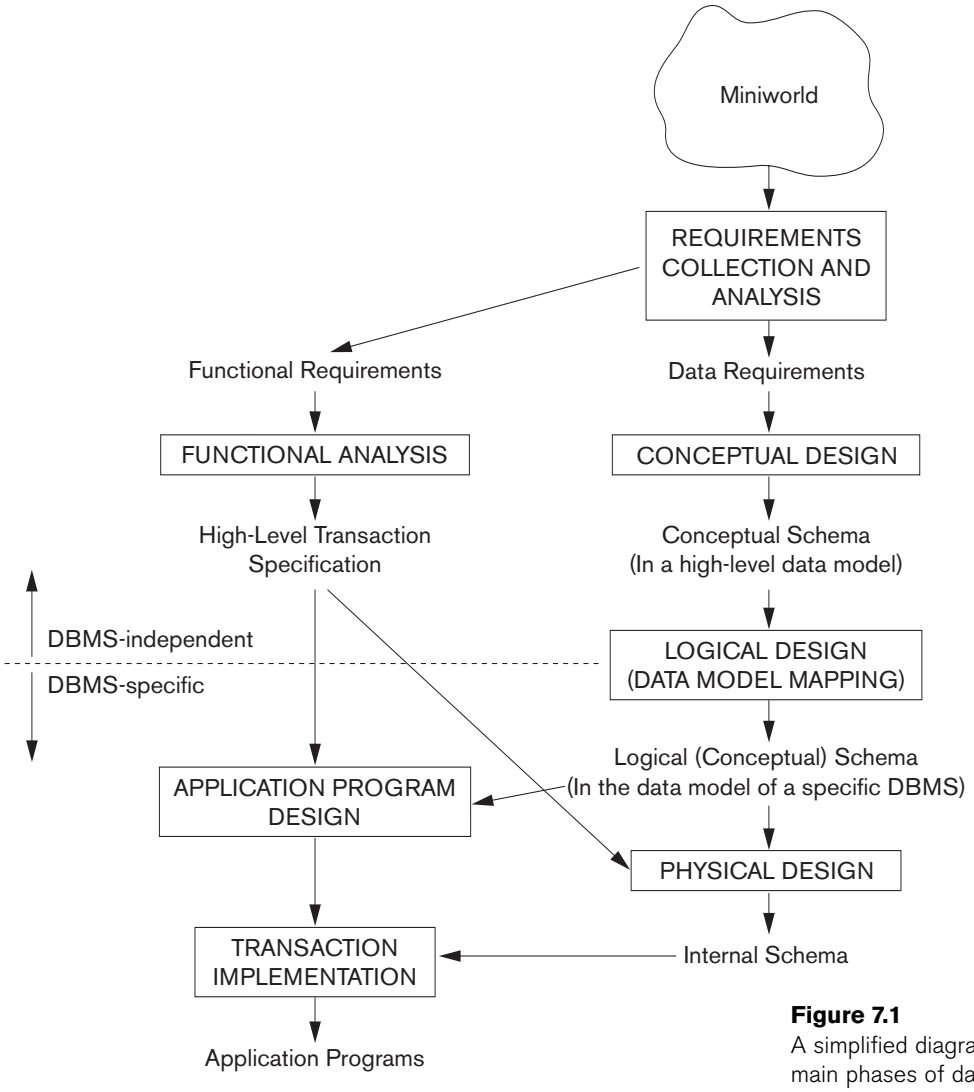


Figure 7.1
A simplified diagram to illustrate the main phases of database design.

the known **functional requirements** of the application. These consist of the user-defined **operations** (or **transactions**) that will be applied to the database, including both retrievals and updates. In software design, it is common to use *data flow diagrams*, *sequence diagrams*, *scenarios*, and other techniques to specify functional requirements. We will not discuss any of these techniques here; they are usually described in detail in software engineering texts. We give an overview of some of these techniques in Chapter 10.

Once the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design**. The conceptual schema is a concise description of

the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with nontechnical users. The high-level conceptual schema can also be used as a reference to ensure that all users' data requirements are met and that the requirements do not conflict. This approach enables database designers to concentrate on specifying the properties of the data, without being concerned with storage and implementation details. This makes it is easier to create a good conceptual database design.

During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user queries and operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified using the initial schema.

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object-relational database model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping**; its result is a database schema in the implementation data model of the DBMS. Data model mapping is often automated or semiautomated within the database design tools.

The last step is the **physical design** phase, during which the internal storage structures, file organizations, indexes, access paths, and physical design parameters for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications. We discuss the database design process in more detail in Chapter 10.

We present only the basic ER model concepts for conceptual schema design in this chapter. Additional modeling concepts are discussed in Chapter 8, when we introduce the EER model.

7.2 A Sample Database Application

In this section we describe a sample database application, called COMPANY, which serves to illustrate the basic ER model concepts and their use in schema design. We list the data requirements for the database here, and then create its conceptual schema step-by-step as we introduce the modeling concepts of the ER model. The COMPANY database keeps track of a company's employees, departments, and projects. Suppose that after the requirements collection and analysis phase, the database designers provide the following description of the *miniworld*—the part of the company that will be represented in the database.

- The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.
- A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
- We store each employee's name, Social Security number,[2] address, salary, sex (gender), and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. We keep track of the current number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee (who is another employee).
- We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date, and relationship to the employee.

Figure 7.2 shows how the schema for this database application can be displayed by means of the graphical notation known as **ER diagrams**. This figure will be explained gradually as the ER model concepts are presented. We describe the step-by-step process of deriving this schema from the stated requirements—and explain the ER diagrammatic notation—as we introduce the ER model concepts.

7.3 Entity Types, Entity Sets, Attributes, and Keys

The ER model describes data as *entities*, *relationships*, and *attributes*. In Section 7.3.1 we introduce the concepts of entities and their attributes. We discuss entity types and key attributes in Section 7.3.2. Then, in Section 7.3.3, we specify the initial conceptual design of the entity types for the COMPANY database. Relationships are described in Section 7.4.

7.3.1 Entities and Attributes

Entities and Their Attributes. The basic object that the ER model represents is an **entity**, which is a *thing* in the real world with an independent existence. An entity may be an object with a physical existence (for example, a particular person, car, house, or employee) or it may be an object with a conceptual existence (for instance, a company, a job, or a university course). Each entity has **attributes**—the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee's name, age, address, salary, and job. A particular entity will have a

[2]The Social Security number, or SSN, is a unique nine-digit identifier assigned to each individual in the United States to keep track of his or her employment, benefits, and taxes. Other countries may have similar identification schemes, such as personal identification card numbers.

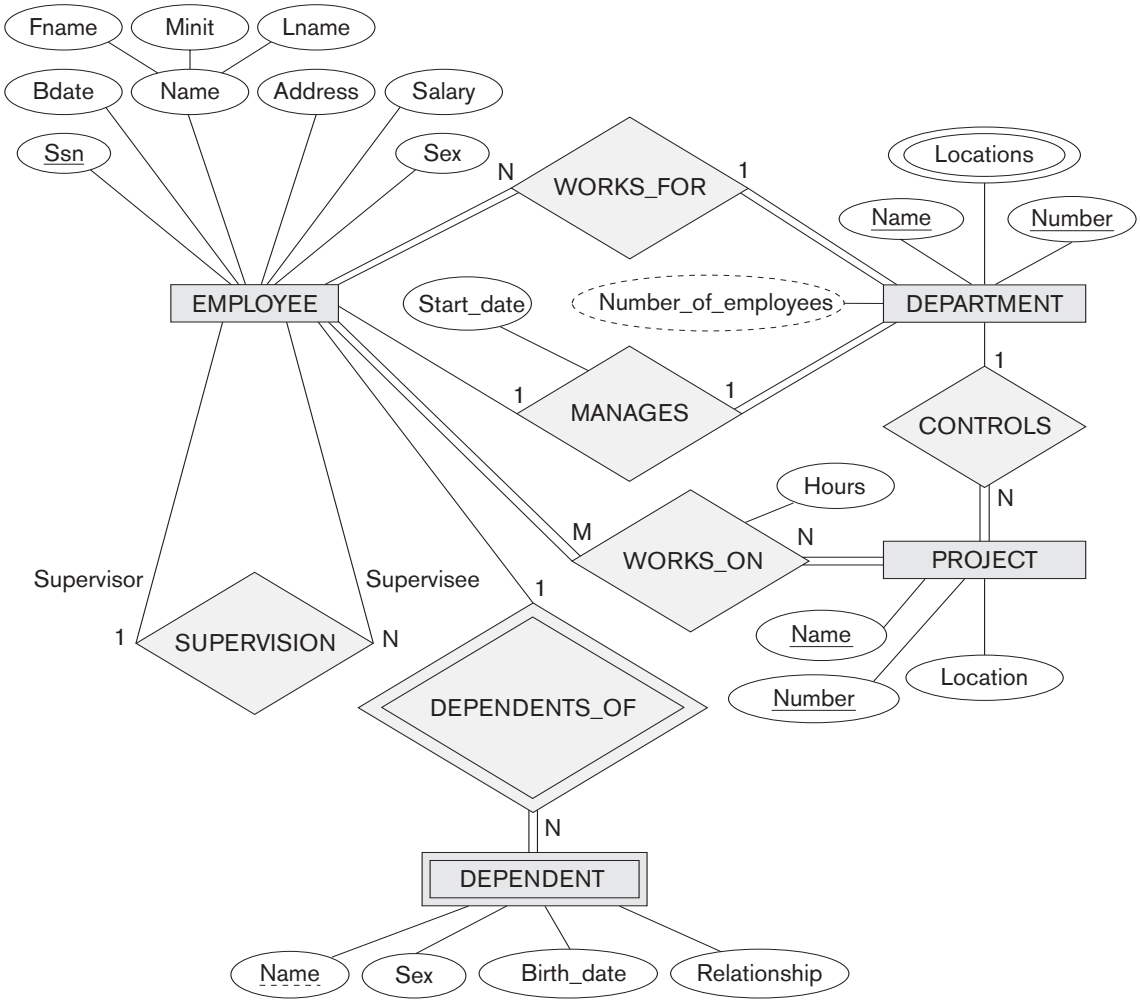


Figure 7.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter and is summarized in Figure 7.14.

value for each of its attributes. The attribute values that describe each entity become a major part of the data stored in the database.

Figure 7.3 shows two entities and the values of their attributes. The EMPLOYEE entity e_1 has four attributes: Name, Address, Age, and Home_phone; their values are 'John Smith,' '2311 Kirby, Houston, Texas 77001', '55', and '713-749-2630', respectively. The COMPANY entity c_1 has three attributes: Name, Headquarters, and President; their values are 'Sunco Oil', 'Houston', and 'John Smith', respectively.

Several types of attributes occur in the ER model: *simple* versus *composite*, *single-valued* versus *multivalued*, and *stored* versus *derived*. First we define these attribute

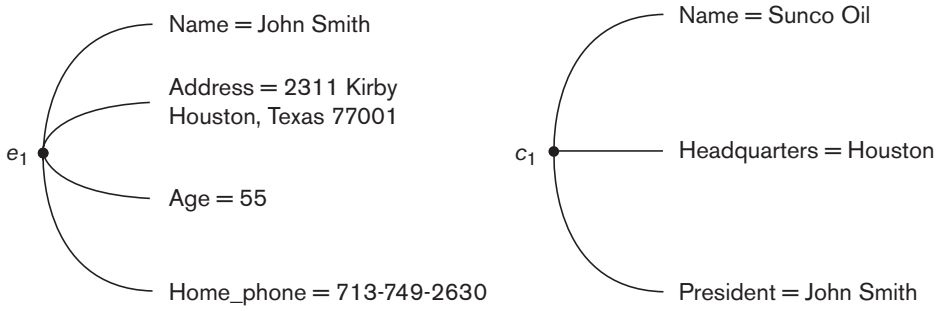


Figure 7.3
Two entities, EMPLOYEE e_1, and COMPANY c_1, and their attributes.

types and illustrate their use via examples. Then we discuss the concept of a *NULL value* for an attribute.

Composite versus Simple (Atomic) Attributes. **Composite attributes** can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the EMPLOYEE entity shown in Figure 7.3 can be subdivided into Street_address, City, State, and Zip,[3] with the values '2311 Kirby', 'Houston', 'Texas', and '77001.' Attributes that are not divisible are called **simple** or **atomic attributes**. Composite attributes can form a hierarchy; for example, Street_address can be further subdivided into three simple component attributes: Number, Street, and Apartment_number, as shown in Figure 7.4. The value of a composite attribute is the concatenation of the values of its component simple attributes.

Composite attributes are useful to model situations in which a user sometimes refers to the composite attribute as a unit but at other times refers specifically to its components. If the composite attribute is referenced only as a whole, there is no

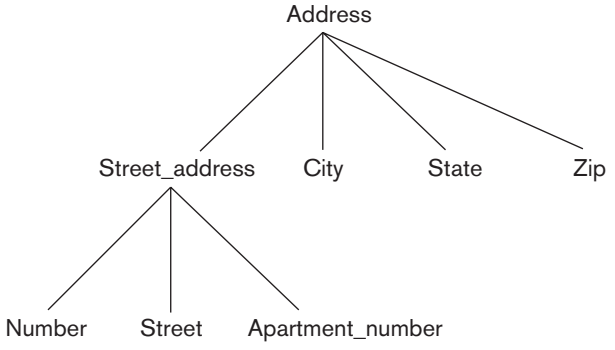


Figure 7.4
A hierarchy of composite attributes.

[3]Zip Code is the name used in the United States for a five-digit postal code, such as 76019, which can be extended to nine digits, such as 76019-0015. We use the five-digit Zip in our examples.

need to subdivide it into component attributes. For example, if there is no need to refer to the individual components of an address (Zip Code, street, and so on), then the whole address can be designated as a simple attribute.

Single-Valued versus Multivalued Attributes. Most attributes have a single value for a particular entity; such attributes are called **single-valued**. For example, Age is a single-valued attribute of a person. In some cases an attribute can have a set of values for the same entity—for instance, a Colors attribute for a car, or a College_degrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two color values. Similarly, one person may not have a college degree, another person may have one, and a third person may have two or more degrees; therefore, different people can have different *numbers* of *values* for the College_degrees attribute. Such attributes are called **multivalued**. A multivalued attribute may have lower and upper bounds to constrain the *number of values* allowed for each individual entity. For example, the Colors attribute of a car may be restricted to have between one and three values, if we assume that a car can have three colors at most.

Stored versus Derived Attributes. In some cases, two (or more) attribute values are related—for example, the Age and Birth_date attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's Birth_date. The Age attribute is hence called a **derived attribute** and is said to be **derivable from** the Birth_date attribute, which is called a **stored attribute**. Some attribute values can be derived from *related entities*; for example, an attribute Number_of_employees of a DEPARTMENT entity can be derived by counting the number of employees related to (working for) that department.

NULL Values. In some cases, a particular entity may not have an applicable value for an attribute. For example, the Apartment_number attribute of an address applies only to addresses that are in apartment buildings and not to other types of residences, such as single-family homes. Similarly, a College_degrees attribute applies only to people with college degrees. For such situations, a special value called NULL is created. An address of a single-family home would have NULL for its Apartment_number attribute, and a person with no college degree would have NULL for College_degrees. NULL can also be used if we do not know the value of an attribute for a particular entity—for example, if we do not know the home phone number of 'John Smith' in Figure 7.3. The meaning of the former type of NULL is *not applicable*, whereas the meaning of the latter is *unknown*. The *unknown* category of NULL can be further classified into two cases. The first case arises when it is known that the attribute value exists but is *missing*—for instance, if the Height attribute of a person is listed as NULL. The second case arises when it is *not known* whether the attribute value exists—for example, if the Home_phone attribute of a person is NULL.

Complex Attributes. Notice that, in general, composite and multivalued attributes can be nested arbitrarily. We can represent arbitrary nesting by grouping com-

ponents of a composite attribute between parentheses () and separating the components with commas, and by displaying multivalued attributes between braces { }. Such attributes are called **complex attributes**. For example, if a person can have more than one residence and each residence can have a single address and multiple phones, an attribute Address_phone for a person can be specified as shown in Figure 7.5.[4] Both Phone and Address are themselves composite attributes.

7.3.2 Entity Types, Entity Sets, Keys, and Value Sets

Entity Types and Entity Sets. A database usually contains groups of entities that are similar. For example, a company employing hundreds of employees may want to store similar information concerning each of the employees. These employee entities share the same attributes, but each entity has its *own value*(*s*) for each attribute. An **entity type** defines a *collection* (or *set*) of entities that have the same attributes. Each entity type in the database is described by its name and attributes. Figure 7.6 shows two entity types: EMPLOYEE and COMPANY, and a list of some of the attributes for

{Address_phone({Phone(Area_code,Phone_number)},Address(Street_address (Number,Street,Apartment_number),City,State,Zip))}

Figure 7.5
A complex attribute: Address_phone.

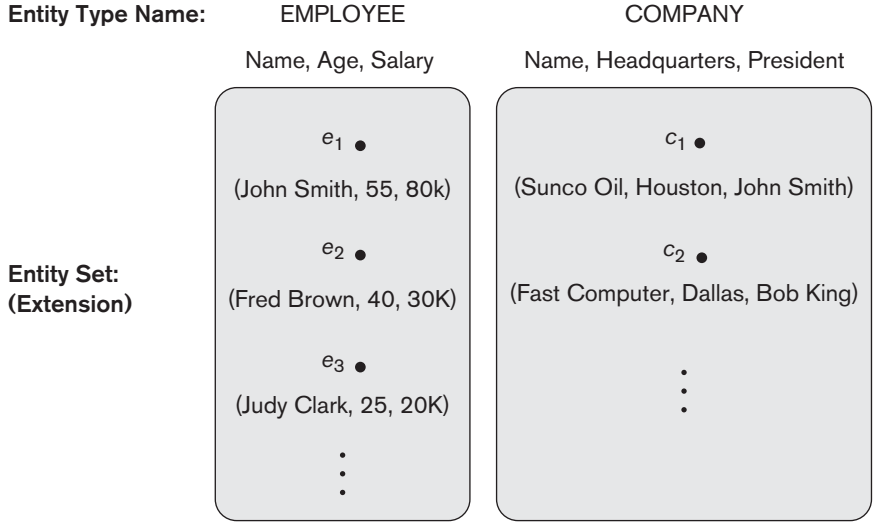


Figure 7.6
Two entity types, EMPLOYEE and COMPANY, and some member entities of each.

[4]For those familiar with XML, we should note that complex attributes are similar to complex elements in XML (see Chapter 12).

each. A few individual entities of each type are also illustrated, along with the values of their attributes. The collection of all entities of a particular entity type in the database at any point in time is called an **entity set**; the entity set is usually referred to using the same name as the entity type. For example, EMPLOYEE refers to both a *type of entity* as well as the current set *of all employee entities* in the database.

An entity type is represented in ER diagrams[5] (see Figure 7.2) as a rectangular box enclosing the entity type name. Attribute names are enclosed in ovals and are attached to their entity type by straight lines. Composite attributes are attached to their component attributes by straight lines. Multivalued attributes are displayed in double ovals. Figure 7.7(a) shows a CAR entity type in this notation.

An entity type describes the **schema** or **intension** for a *set of entities* that share the same structure. The collection of entities of a particular entity type is grouped into an entity set, which is also called the **extension** of the entity type.

Key Attributes of an Entity Type. An important constraint on the entities of an entity type is the **key** or **uniqueness constraint** on attributes. An entity type usually

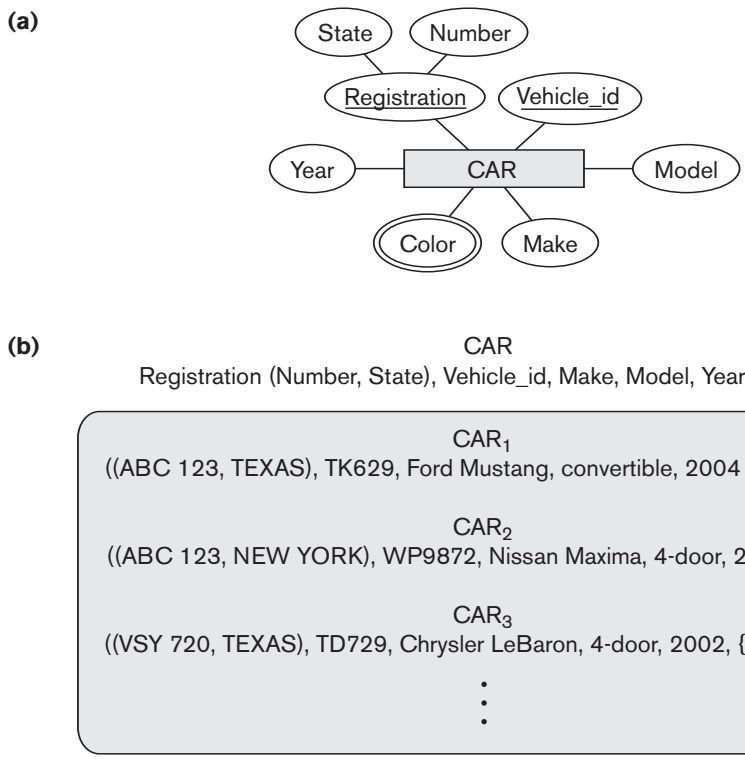


Figure 7.7
The CAR entity type with two key attributes, Registration and Vehicle_id. (a) ER diagram notation. (b) Entity set with three entities.

[5]We use a notation for ER diagrams that is close to the original proposed notation (Chen 1976). Many other notations are in use; we illustrate some of them later in this chapter when we present UML class diagrams and in Appendix A.

has one or more attributes whose values are distinct for each individual entity in the entity set. Such an attribute is called a **key attribute**, and its values can be used to identify each entity uniquely. For example, the Name attribute is a key of the COMPANY entity type in Figure 7.6 because no two companies are allowed to have the same name. For the PERSON entity type, a typical key attribute is Ssn (Social Security number). Sometimes several attributes together form a key, meaning that the *combination* of the attribute values must be distinct for each entity. If a set of attributes possesses this property, the proper way to represent this in the ER model that we describe here is to define a *composite attribute* and designate it as a key attribute of the entity type. Notice that such a composite key must be *minimal*; that is, all component attributes must be included in the composite attribute to have the uniqueness property. Superfluous attributes must not be included in a key. In ER diagrammatic notation, each key attribute has its name **underlined** inside the oval, as illustrated in Figure 7.7(a).

Specifying that an attribute is a key of an entity type means that the preceding uniqueness property must hold for *every entity set* of the entity type. Hence, it is a constraint that prohibits any two entities from having the same value for the key attribute at the same time. It is not the property of a particular entity set; rather, it is a constraint on *any entity set* of the entity type at any point in time. This key constraint (and other constraints we discuss later) is derived from the constraints of the miniworld that the database represents.

Some entity types have *more than one* key attribute. For example, each of the Vehicle_id and Registration attributes of the entity type CAR (Figure 7.7) is a key in its own right. The Registration attribute is an example of a composite key formed from two simple component attributes, State and Number, neither of which is a key on its own. An entity type may also have *no key*, in which case it is called a *weak entity type* (see Section 7.5).

In our diagrammatic notation, if two attributes are underlined separately, then *each is a key on its own*. Unlike the relational model (see Section 3.2.2), there is no concept of primary key in the ER model that we present here; the primary key will be chosen during mapping to a relational schema (see Chapter 9).

Value Sets (Domains) of Attributes. Each simple attribute of an entity type is associated with a **value set** (or **domain** of values), which specifies the set of values that may be assigned to that attribute for each individual entity. In Figure 7.6, if the range of ages allowed for employees is between 16 and 70, we can specify the value set of the Age attribute of EMPLOYEE to be the set of integer numbers between 16 and 70. Similarly, we can specify the value set for the Name attribute to be the set of strings of alphabetic characters separated by blank characters, and so on. Value sets are not displayed in ER diagrams, and are typically specified using the basic **data types** available in most programming languages, such as integer, string, Boolean, float, enumerated type, subrange, and so on. Additional data types to represent common database types such as date, time, and other concepts are also employed.

Mathematically, an attribute A of entity set E whose value set is V can be defined as a **function** from E to the power set[6] $P(V)$ of V:

$$A : E \rightarrow P(V)$$

We refer to the value of attribute A for entity e as $A(e)$. The previous definition covers both single-valued and multivalued attributes, as well as NULLs. A NULL value is represented by the *empty set*. For single-valued attributes, $A(e)$ is restricted to being a *singleton set* for each entity e in E, whereas there is no restriction on multivalued attributes.[7] For a composite attribute A, the value set V is the power set of the Cartesian product of $P(V_1)$, $P(V_2)$, ..., $P(V_n)$, where V_1, V_2, ..., V_n are the value sets of the simple component attributes that form A:

$$V = P\,(P(V_1) \times P(V_2) \times ... \times P(V_n))$$

The value set provides all possible values. Usually only a small number of these values exist in the database at a particular time. Those values represent the data from the current state of the miniworld. They correspond to the data as it actually exists in the miniworld.

7.3.3 Initial Conceptual Design of the COMPANY Database

We can now define the entity types for the COMPANY database, based on the requirements described in Section 7.2. After defining several entity types and their attributes here, we refine our design in Section 7.4 after we introduce the concept of a relationship. According to the requirements listed in Section 7.2, we can identify four entity types—one corresponding to each of the four items in the specification (see Figure 7.8):

1. An entity type DEPARTMENT with attributes Name, Number, Locations, Manager, and Manager_start_date. Locations is the only multivalued attribute. We can specify that both Name and Number are (separate) key attributes because each was specified to be unique.
2. An entity type PROJECT with attributes Name, Number, Location, and Controlling_department. Both Name and Number are (separate) key attributes.
3. An entity type EMPLOYEE with attributes Name, Ssn, Sex, Address, Salary, Birth_date, Department, and Supervisor. Both Name and Address may be composite attributes; however, this was not specified in the requirements. We must go back to the users to see if any of them will refer to the individual components of Name—First_name, Middle_initial, Last_name—or of Address.
4. An entity type DEPENDENT with attributes Employee, Dependent_name, Sex, Birth_date, and Relationship (to the employee).

---

[6]The **power set** $P(V)$ of a set V is the set of all subsets of V.

[7]A **singleton** set is a set with only one element (value).

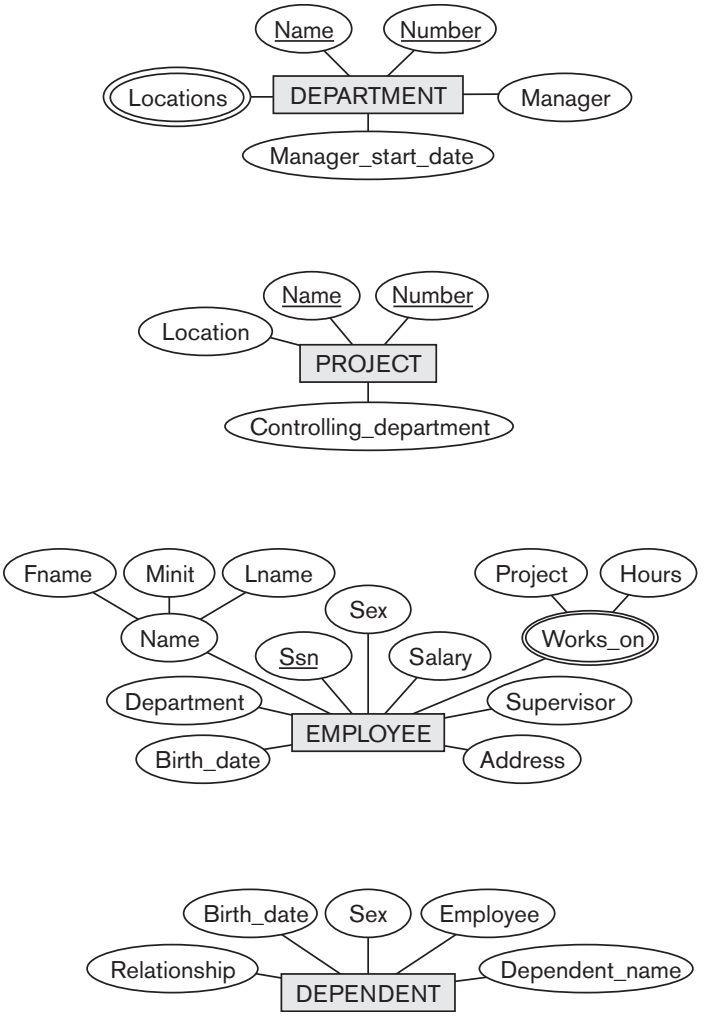


Figure 7.8
Preliminary design of entity types for the COMPANY database. Some of the shown attributes will be refined into relationships.

So far, we have not represented the fact that an employee can work on several projects, nor have we represented the number of hours per week an employee works on each project. This characteristic is listed as part of the third requirement in Section 7.2, and it can be represented by a multivalued composite attribute of EMPLOYEE called Works_on with the simple components (Project, Hours). Alternatively, it can be represented as a multivalued composite attribute of PROJECT called Workers with the simple components (Employee, Hours). We choose the first alternative in Figure 7.8, which shows each of the entity types just described. The Name attribute of EMPLOYEE is shown as a composite attribute, presumably after consultation with the users.

7.4 Relationship Types, Relationship Sets, Roles, and Structural Constraints

In Figure 7.8 there are several *implicit relationships* among the various entity types. In fact, whenever an attribute of one entity type refers to another entity type, some relationship exists. For example, the attribute Manager of DEPARTMENT refers to an employee who manages the department; the attribute Controlling_department of PROJECT refers to the department that controls the project; the attribute Supervisor of EMPLOYEE refers to another employee (the one who supervises this employee); the attribute Department of EMPLOYEE refers to the department for which the employee works; and so on. In the ER model, these references should not be represented as attributes but as **relationships**, which are discussed in this section. The COMPANY database schema will be refined in Section 7.6 to represent relationships explicitly. In the initial design of entity types, relationships are typically captured in the form of attributes. As the design is refined, these attributes get converted into relationships between entity types.

This section is organized as follows: Section 7.4.1 introduces the concepts of relationship types, relationship sets, and relationship instances. We define the concepts of relationship degree, role names, and recursive relationships in Section 7.4.2, and then we discuss structural constraints on relationships—such as cardinality ratios and existence dependencies—in Section 7.4.3. Section 7.4.4 shows how relationship types can also have attributes.

7.4.1 Relationship Types, Sets, and Instances

A **relationship type** R among n entity types $E_1, E_2, \ldots, E_n$ defines a set of associations—or a **relationship set**—among entities from these entity types. As for the case of entity types and entity sets, a relationship type and its corresponding relationship set are customarily referred to by the *same name*, R. Mathematically, the relationship set R is a set of **relationship instances** r_i, where each r_i associates n individual entities $(e_1, e_2, \ldots, e_n)$, and each entity e_j in r_i is a member of entity set E_j, $1 \leq j \leq n$. Hence, a relationship set is a mathematical relation on $E_1, E_2, \ldots, E_n$; alternatively, it can be defined as a subset of the Cartesian product of the entity sets $E_1 \times E_2 \times \ldots \times E_n$. Each of the entity types $E_1, E_2, \ldots, E_n$ is said to participate in the relationship type R; similarly, each of the individual entities $e_1, e_2, \ldots, e_n$ is said to **participate** in the relationship instance $r_i = (e_1, e_2, \ldots, e_n)$.

Informally, each relationship instance r_i in R is an association of entities, where the association includes exactly one entity from each participating entity type. Each such relationship instance r_i represents the fact that the entities participating in r_i are related in some way in the corresponding miniworld situation. For example, consider a relationship type WORKS_FOR between the two entity types EMPLOYEE and DEPARTMENT, which associates each employee with the department for which the employee works in the corresponding entity set. Each relationship instance in the relationship set WORKS_FOR associates one EMPLOYEE entity and one DEPARTMENT entity. Figure 7.9 illustrates this example, where each relationship

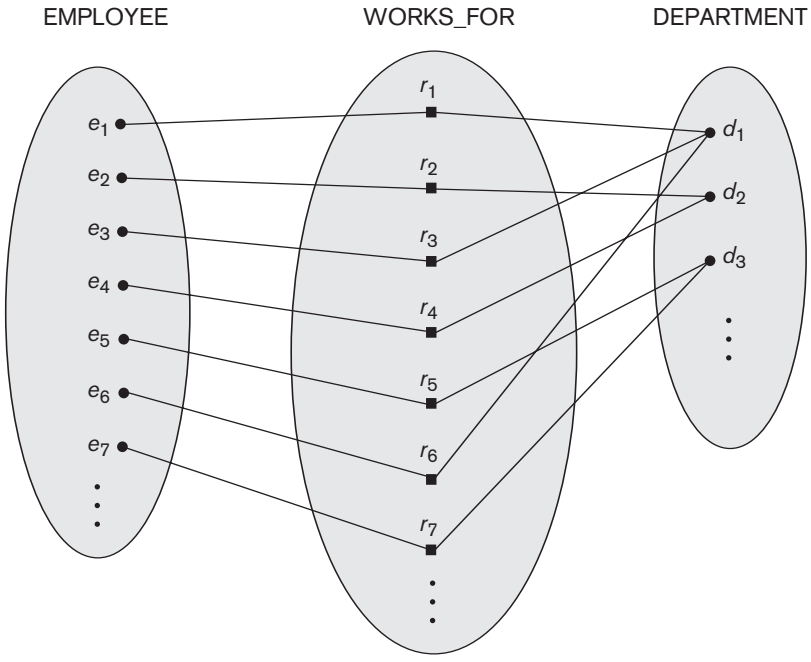


Figure 7.9
Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

instance r_i is shown connected to the EMPLOYEE and DEPARTMENT entities that participate in r_i. In the miniworld represented by Figure 7.9, employees e_1, e_3, and e_6 work for department d_1; employees e_2 and e_4 work for department d_2; and employees e_5 and e_7 work for department d_3.

In ER diagrams, relationship types are displayed as diamond-shaped boxes, which are connected by straight lines to the rectangular boxes representing the participating entity types. The relationship name is displayed in the diamond-shaped box (see Figure 7.2).

7.4.2 Relationship Degree, Role Names, and Recursive Relationships

Degree of a Relationship Type. The **degree** of a relationship type is the number of participating entity types. Hence, the WORKS_FOR relationship is of degree two. A relationship type of degree two is called **binary**, and one of degree three is called **ternary**. An example of a ternary relationship is SUPPLY, shown in Figure 7.10, where each relationship instance r_i associates three entities—a supplier s, a part p, and a project j—whenever s supplies part p to project j. Relationships can generally be of any degree, but the ones most common are binary relationships. Higher-degree relationships are generally more complex than binary relationships; we characterize them further in Section 7.9.

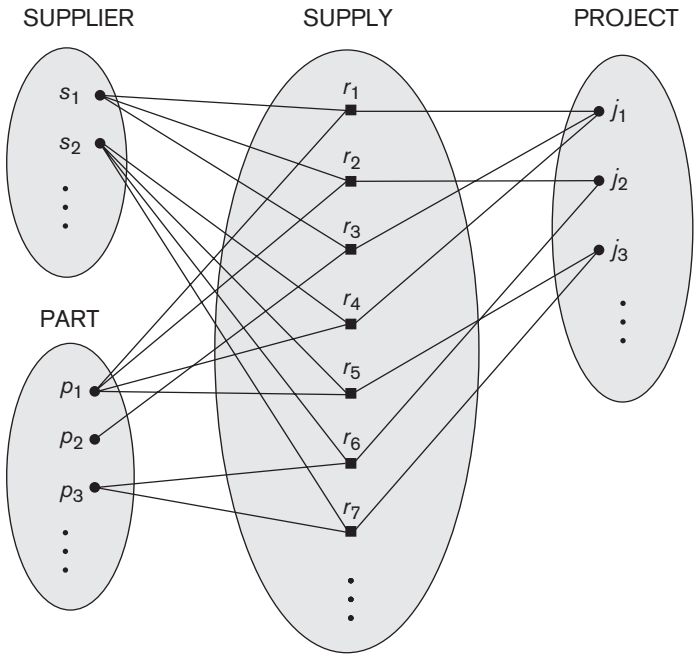


Figure 7.10
Some relationship instances in the SUPPLY ternary relationship set.

Relationships as Attributes. It is sometimes convenient to think of a binary relationship type in terms of attributes, as we discussed in Section 7.3.3. Consider the WORKS_FOR relationship type in Figure 7.9. One can think of an attribute called Department of the EMPLOYEE entity type, where the value of Department for each EMPLOYEE entity is (a reference to) the DEPARTMENT entity for which that employee works. Hence, the value set for this Department attribute is the set of *all* DEPARTMENT entities, which is the DEPARTMENT entity set. This is what we did in Figure 7.8 when we specified the initial design of the entity type EMPLOYEE for the COMPANY database. However, when we think of a binary relationship as an attribute, we always have two options. In this example, the alternative is to think of a multivalued attribute Employee of the entity type DEPARTMENT whose values for each DEPARTMENT entity is the set of EMPLOYEE entities who work for that department. The value set of this Employee attribute is the power set of the EMPLOYEE entity set. Either of these two attributes—Department of EMPLOYEE or Employee of DEPARTMENT—can represent the WORKS_FOR relationship type. If both are represented, they are constrained to be inverses of each other.[8]

[8]This concept of representing relationship types as attributes is used in a class of data models called **functional data models**. In object databases (see Chapter 11), relationships can be represented by reference attributes, either in one direction or in both directions as inverses. In relational databases (see Chapter 3), foreign keys are a type of reference attribute used to represent relationships.

Role Names and Recursive Relationships. Each entity type that participates in a relationship type plays a particular role in the relationship. The **role name** signifies the role that a participating entity from the entity type plays in each relationship instance, and helps to explain what the relationship means. For example, in the WORKS_FOR relationship type, EMPLOYEE plays the role of *employee* or *worker* and DEPARTMENT plays the role of *department* or *employer*.

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each participating entity type name can be used as the role name. However, in some cases the *same* entity type participates more than once in a relationship type in *different roles*. In such cases the role name becomes essential for distinguishing the meaning of the role that each participating entity plays. Such relationship types are called **recursive relationships**. Figure 7.11 shows an example. The SUPERVISION relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity set. Hence, the EMPLOYEE entity type *participates twice* in SUPERVISION: once in the role of *supervisor* (or *boss*), and once in the role of *supervisee* (or *subordinate*). Each relationship instance r_i in SUPERVISION associates two employee entities e_j and e_k, one of which plays the role of supervisor and the other the role of supervisee. In Figure 7.11, the lines marked '1' represent the supervisor role, and those marked '2' represent the supervisee role; hence, e_1 supervises e_2 and e_3, e_4 supervises e_6 and e_7, and e_5 supervises e_1 and e_4. In this example, each relationship instance must be connected with two lines, one marked with '1' (supervisor) and the other with '2' (supervisee).

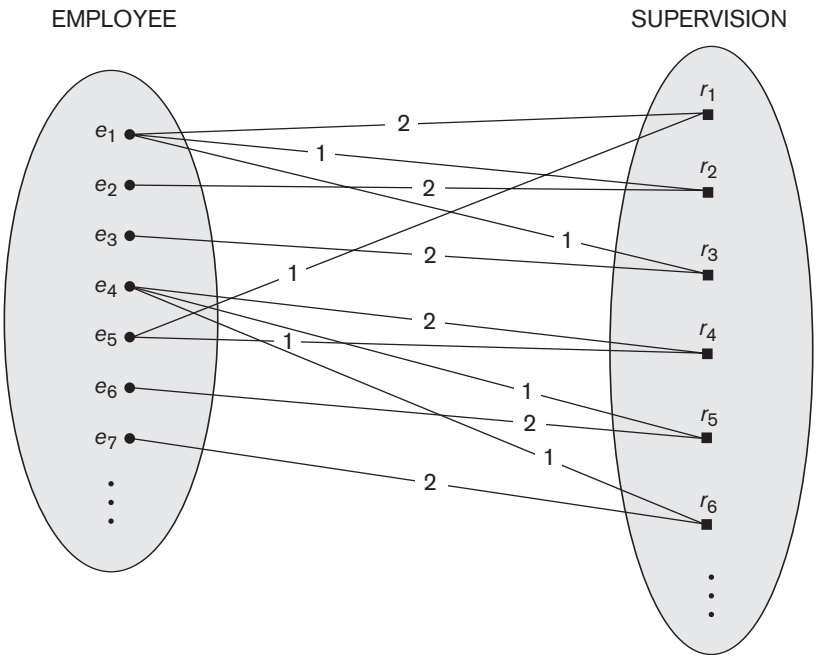


Figure 7.11
A recursive relationship SUPERVISION between EMPLOYEE in the *supervisor* role (1) and EMPLOYEE in the *subordinate* role (2).

7.4.3 Constraints on Binary Relationship Types

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. These constraints are determined from the miniworld situation that the relationships represent. For example, in Figure 7.9, if the company has a rule that each employee must work for exactly one department, then we would like to describe this constraint in the schema. We can distinguish two main types of binary relationship constraints: *cardinality ratio* and *participation*.

Cardinality Ratios for Binary Relationships. The **cardinality ratio** for a binary relationship specifies the *maximum* number of relationship instances that an entity can participate in. For example, in the WORKS_FOR binary relationship type, DEPARTMENT:EMPLOYEE is of cardinality ratio 1:N, meaning that each department can be related to (that is, employs) any number of employees,[9] but an employee can be related to (work for) only one department. This means that for this particular relationship WORKS_FOR, a particular department entity can be related to any number of employees (N indicates there is no maximum number). On the other hand, an employee can be related to a maximum of one department. The possible cardinality ratios for binary relationship types are 1:1, 1:N, N:1, and M:N.

An example of a 1:1 binary relationship is MANAGES (Figure 7.12), which relates a department entity to the employee who manages that department. This represents the miniworld constraints that—at any point in time—an employee can manage one department only and a department can have one manager only. The relationship type WORKS_ON (Figure 7.13) is of cardinality ratio M:N, because the mini-

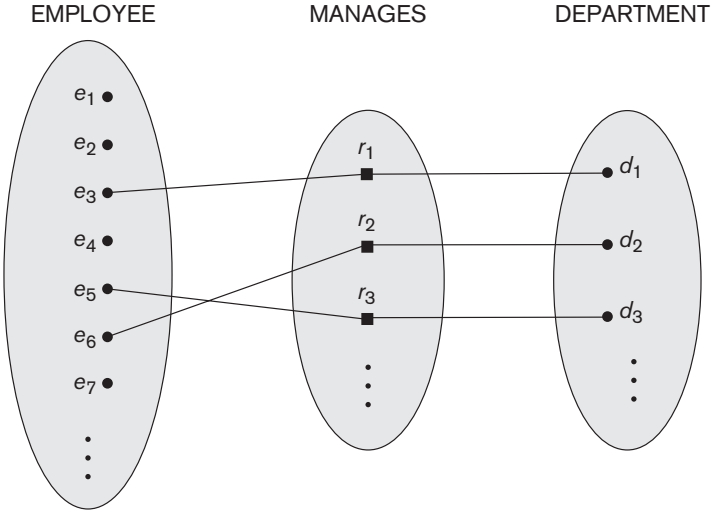


Figure 7.12
A 1:1 relationship, MANAGES.

[9]N stands for *any number* of related entities (zero or more).

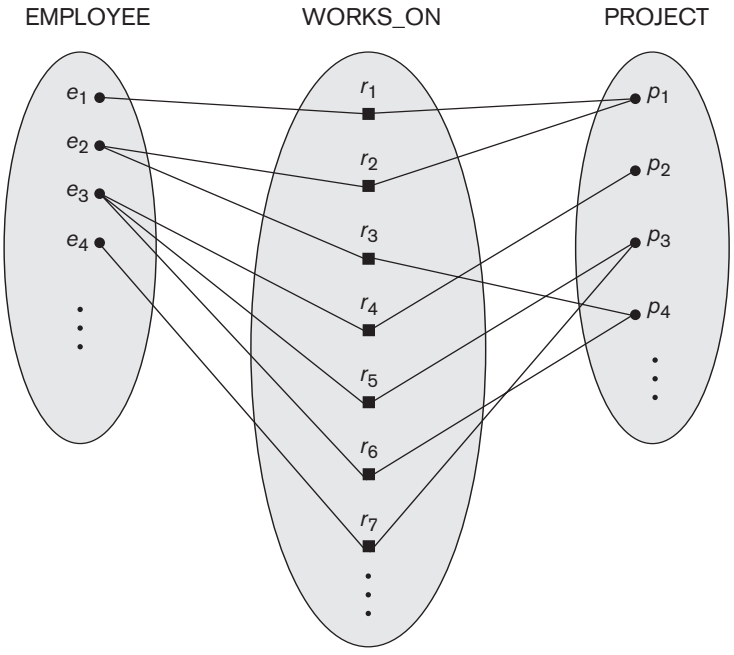


Figure 7.13
An M:N relationship, WORKS_ON.

world rule is that an employee can work on several projects and a project can have several employees.

Cardinality ratios for binary relationships are represented on ER diagrams by displaying 1, M, and N on the diamonds as shown in Figure 7.2. Notice that in this notation, we can either specify no maximum (N) or a maximum of one (1) on participation. An alternative notation (see Section 7.7.4) allows the designer to specify a specific *maximum number* on participation, such as 4 or 5.

Participation Constraints and Existence Dependencies. The **participation constraint** specifies whether the existence of an entity depends on its being related to another entity via the relationship type. This constraint specifies the *minimum* number of relationship instances that each entity can participate in, and is sometimes called the **minimum cardinality constraint**. There are two types of participation constraints—total and partial—that we illustrate by example. If a company policy states that *every* employee must work for a department, then an employee entity can exist only if it participates in at least one WORKS_FOR relationship instance (Figure 7.9). Thus, the participation of EMPLOYEE in WORKS_FOR is called **total participation**, meaning that every entity in *the total set* of employee entities must be related to a department entity via WORKS_FOR. Total participation is also called **existence dependency**. In Figure 7.12 we do not expect every employee to manage a department, so the participation of EMPLOYEE in the MANAGES relationship type is **partial**, meaning that *some* or *part of the set of* employee entities are related to some department entity via MANAGES, but not necessarily all. We will

refer to the cardinality ratio and participation constraints, taken together, as the **structural constraints** of a relationship type.

In ER diagrams, total participation (or existence dependency) is displayed as a *double line* connecting the participating entity type to the relationship, whereas partial participation is represented by a *single line* (see Figure 7.2). Notice that in this notation, we can either specify no minimum (partial participation) or a minimum of one (total participation). The alternative notation (see Section 7.7.4) allows the designer to specify a specific *minimum number* on participation in the relationship, such as 4 or 5.

We will discuss constraints on higher-degree relationships in Section 7.9.

7.4.4 Attributes of Relationship Types

Relationship types can also have attributes, similar to those of entity types. For example, to record the number of hours per week that an employee works on a particular project, we can include an attribute Hours for the WORKS_ON relationship type in Figure 7.13. Another example is to include the date on which a manager started managing a department via an attribute Start_date for the MANAGES relationship type in Figure 7.12.

Notice that attributes of 1:1 or 1:N relationship types can be migrated to one of the participating entity types. For example, the Start_date attribute for the MANAGES relationship can be an attribute of either EMPLOYEE or DEPARTMENT, although conceptually it belongs to MANAGES. This is because MANAGES is a 1:1 relationship, so every department or employee entity participates in *at most one* relationship instance. Hence, the value of the Start_date attribute can be determined separately, either by the participating department entity or by the participating employee (manager) entity.

For a 1:N relationship type, a relationship attribute can be migrated *only* to the entity type on the N-side of the relationship. For example, in Figure 7.9, if the WORKS_FOR relationship also has an attribute Start_date that indicates when an employee started working for a department, this attribute can be included as an attribute of EMPLOYEE. This is because each employee works for only one department, and hence participates in at most one relationship instance in WORKS_FOR. In both 1:1 and 1:N relationship types, the decision where to place a relationship attribute—as a relationship type attribute or as an attribute of a participating entity type—is determined subjectively by the schema designer.

For M:N relationship types, some attributes may be determined by the *combination of participating entities* in a relationship instance, not by any single entity. Such attributes *must be specified as relationship attributes*. An example is the Hours attribute of the M:N relationship WORKS_ON (Figure 7.13); the number of hours per week an employee currently works on a project is determined by an employee-project combination and not separately by either entity.

7.5 Weak Entity Types

Entity types that do not have key attributes of their own are called **weak entity types**. In contrast, **regular entity types** that do have a key attribute—which include all the examples discussed so far—are called **strong entity types**. Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. We call this other entity type the **identifying** or **owner entity type**,[10] and we call the relationship type that relates a weak entity type to its owner the **identifying relationship** of the weak entity type.[11] A weak entity type always has a *total participation constraint* (existence dependency) with respect to its identifying relationship because a weak entity cannot be identified without an owner entity. However, not every existence dependency results in a weak entity type. For example, a DRIVER_LICENSE entity cannot exist unless it is related to a PERSON entity, even though it has its own key (License_number) and hence is not a weak entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents of each employee via a 1:N relationship (Figure 7.2). In our example, the attributes of DEPENDENT are Name (the first name of the dependent), Birth_date, Sex, and Relationship (to the employee). Two dependents of *two distinct employees* may, by chance, have the same values for Name, Birth_date, Sex, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the *particular employee entity* to which each dependent is related. Each employee entity is said to *own* the dependent entities that are related to it.

A weak entity type normally has a **partial key**, which is the attribute that can uniquely identify weak entities that are *related to the same owner entity*.[12] In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key. In the worst case, a composite attribute of *all the weak entity's attributes* will be the partial key.

In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with double lines (see Figure 7.2). The partial key attribute is underlined with a dashed or dotted line.

Weak entity types can sometimes be represented as complex (composite, multivalued) attributes. In the preceding example, we could specify a multivalued attribute Dependents for EMPLOYEE, which is a composite attribute with component attributes Name, Birth_date, Sex, and Relationship. The choice of which representation to use is made by the database designer. One criterion that may be used is to choose the

[10]The identifying entity type is also sometimes called the **parent entity type** or the **dominant entity type**.

[11]The weak entity type is also sometimes called the **child entity type** or the **subordinate entity type**.

[12]The partial key is sometimes called the **discriminator**.

weak entity type representation if there are many attributes. If the weak entity participates independently in relationship types other than its identifying relationship type, then it should not be modeled as a complex attribute.

In general, any number of levels of weak entity types can be defined; an owner entity type may itself be a weak entity type. In addition, a weak entity type may have more than one identifying entity type and an identifying relationship type of degree higher than two, as we illustrate in Section 7.9.

7.6 Refining the ER Design for the COMPANY Database

We can now refine the database design in Figure 7.8 by changing the attributes that represent relationships into relationship types. The cardinality ratio and participation constraint of each relationship type are determined from the requirements listed in Section 7.2. If some cardinality ratio or dependency cannot be determined from the requirements, the users must be questioned further to determine these structural constraints.

In our example, we specify the following relationship types:

- MANAGES, a 1:1 relationship type between EMPLOYEE and DEPARTMENT. EMPLOYEE participation is partial. DEPARTMENT participation is not clear from the requirements. We question the users, who say that a department must have a manager at all times, which implies total participation.[13] The attribute Start_date is assigned to this relationship type.
- WORKS_FOR, a 1:N relationship type between DEPARTMENT and EMPLOYEE. Both participations are total.
- CONTROLS, a 1:N relationship type between DEPARTMENT and PROJECT. The participation of PROJECT is total, whereas that of DEPARTMENT is determined to be partial, after consultation with the users indicates that some departments may control no projects.
- SUPERVISION, a 1:N relationship type between EMPLOYEE (in the supervisor role) and EMPLOYEE (in the supervisee role). Both participations are determined to be partial, after the users indicate that not every employee is a supervisor and not every employee has a supervisor.
- WORKS_ON, determined to be an M:N relationship type with attribute Hours, after the users indicate that a project can have several employees working on it. Both participations are determined to be total.
- DEPENDENTS_OF, a 1:N relationship type between EMPLOYEE and DEPENDENT, which is also the identifying relationship for the weak entity

[13]The rules in the miniworld that determine the constraints are sometimes called the *business rules*, since they are determined by the *business* or organization that will utilize the database.

type DEPENDENT. The participation of EMPLOYEE is partial, whereas that of DEPENDENT is total.

After specifying the above six relationship types, we remove from the entity types in Figure 7.8 all attributes that have been refined into relationships. These include Manager and Manager_start_date from DEPARTMENT; Controlling_department from PROJECT; Department, Supervisor, and Works_on from EMPLOYEE; and Employee from DEPENDENT. It is important to have the least possible redundancy when we design the conceptual schema of a database. If some redundancy is desired at the storage level or at the user view level, it can be introduced later, as discussed in Section 1.6.1.

7.7 ER Diagrams, Naming Conventions, and Design Issues

7.7.1 Summary of Notation for ER Diagrams

Figures 7.9 through 7.13 illustrate examples of the participation of entity types in relationship types by displaying their sets or extensions—the individual entity instances in an entity set and the individual relationship instances in a relationship set. In ER diagrams the emphasis is on representing the schemas rather than the instances. This is more useful in database design because a database schema changes rarely, whereas the contents of the entity sets change frequently. In addition, the schema is obviously easier to display, because it is much smaller.

Figure 7.2 displays the COMPANY **ER database schema** as an **ER diagram**. We now review the full ER diagram notation. Entity types such as EMPLOYEE, DEPARTMENT, and PROJECT are shown in rectangular boxes. Relationship types such as WORKS_FOR, MANAGES, CONTROLS, and WORKS_ON are shown in diamond-shaped boxes attached to the participating entity types with straight lines. Attributes are shown in ovals, and each attribute is attached by a straight line to its entity type or relationship type. Component attributes of a composite attribute are attached to the oval representing the composite attribute, as illustrated by the Name attribute of EMPLOYEE. Multivalued attributes are shown in double ovals, as illustrated by the Locations attribute of DEPARTMENT. Key attributes have their names underlined. Derived attributes are shown in dotted ovals, as illustrated by the Number_of_employees attribute of DEPARTMENT.

Weak entity types are distinguished by being placed in double rectangles and by having their identifying relationship placed in double diamonds, as illustrated by the DEPENDENT entity type and the DEPENDENTS_OF identifying relationship type. The partial key of the weak entity type is underlined with a dotted line.

In Figure 7.2 the cardinality ratio of each *binary* relationship type is specified by attaching a 1, M, or N on each participating edge. The cardinality ratio of DEPARTMENT:EMPLOYEE in MANAGES is 1:1, whereas it is 1:N for DEPARTMENT: EMPLOYEE in WORKS_FOR, and M:N for WORKS_ON. The participation

Review Questions

7.1. Discuss the role of a high-level data model in the database design process.

7.2. List the various cases where use of a NULL value would be appropriate.

7.3. Define the following terms: *entity, attribute, attribute value, relationship instance, composite attribute, multivalued attribute, derived attribute, complex attribute, key attribute,* and *value set* (*domain*).

7.4. What is an entity type? What is an entity set? Explain the differences among an entity, an entity type, and an entity set.

7.5. Explain the difference between an attribute and a value set.

7.6. What is a relationship type? Explain the differences among a relationship instance, a relationship type, and a relationship set.

7.7. What is a participation role? When is it necessary to use role names in the description of relationship types?

7.8. Describe the two alternatives for specifying structural constraints on relationship types. What are the advantages and disadvantages of each?

7.9. Under what conditions can an attribute of a binary relationship type be migrated to become an attribute of one of the participating entity types?

7.10. When we think of relationships as attributes, what are the value sets of these attributes? What class of data models is based on this concept?

7.11. What is meant by a recursive relationship type? Give some examples of recursive relationship types.

7.12. When is the concept of a weak entity used in data modeling? Define the terms *owner entity type, weak entity type, identifying relationship type,* and *partial key*.

7.13. Can an identifying relationship of a weak entity type be of a degree greater than two? Give examples to illustrate your answer.

7.14. Discuss the conventions for displaying an ER schema as an ER diagram.

7.15. Discuss the naming conventions used for ER schema diagrams.

Exercises

7.16. Consider the following set of requirements for a UNIVERSITY database that is used to keep track of students' transcripts. This is similar but not identical to the database shown in Figure 1.2:

a. The university keeps track of each student's name, student number, Social Security number, current address and phone number, permanent address and phone number, birth date, sex, class (freshman, sophomore, ..., graduate), major department, minor department (if any), and degree program

(B.A., B.S., ..., Ph.D.). Some user applications need to refer to the city, state, and ZIP Code of the student's permanent address and to the student's last name. Both Social Security number and student number have unique values for each student.

b. Each department is described by a name, department code, office number, office phone number, and college. Both name and code have unique values for each department.

c. Each course has a course name, description, course number, number of semester hours, level, and offering department. The value of the course number is unique for each course.

d. Each section has an instructor, semester, year, course, and section number. The section number distinguishes sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, ..., up to the number of sections taught during each semester.

e. A grade report has a student, section, letter grade, and numeric grade (0, 1, 2, 3, or 4).

Design an ER schema for this application, and draw an ER diagram for the schema. Specify key attributes of each entity type, and structural constraints on each relationship type. Note any unspecified requirements, and make appropriate assumptions to make the specification complete.

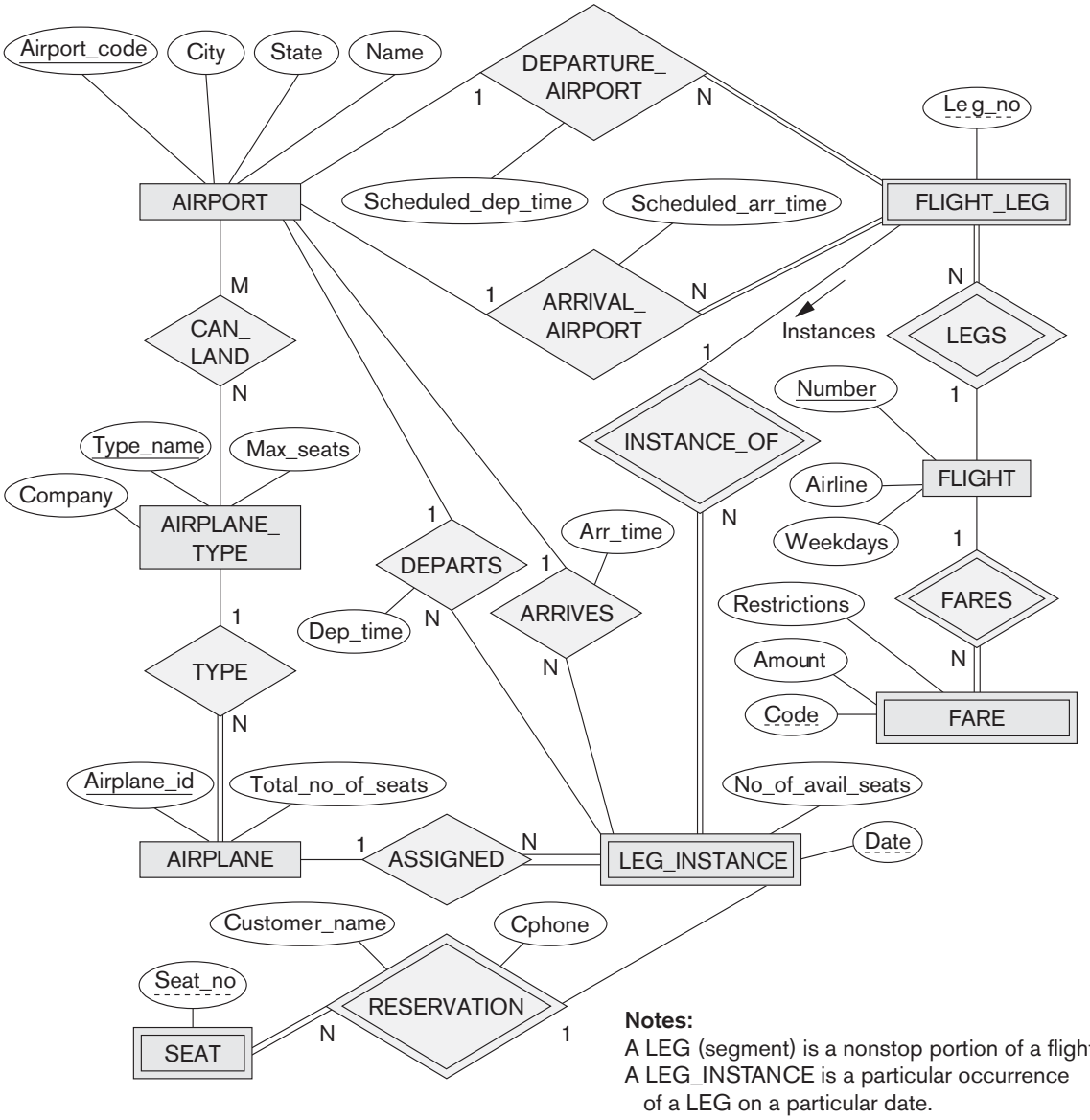
7.17. Composite and multivalued attributes can be nested to any number of levels. Suppose we want to design an attribute for a STUDENT entity type to keep track of previous college education. Such an attribute will have one entry for each college previously attended, and each such entry will be composed of college name, start and end dates, degree entries (degrees awarded at that college, if any), and transcript entries (courses completed at that college, if any). Each degree entry contains the degree name and the month and year the degree was awarded, and each transcript entry contains a course name, semester, year, and grade. Design an attribute to hold this information. Use the conventions in Figure 7.5.

7.18. Show an alternative design for the attribute described in Exercise 7.17 that uses only entity types (including weak entity types, if needed) and relationship types.

7.19. Consider the ER diagram in Figure 7.20, which shows a simplified schema for an airline reservations system. Extract from the ER diagram the requirements and constraints that produced this schema. Try to be as precise as possible in your requirements and constraints specification.

7.20. In Chapters 1 and 2, we discussed the database environment and database users. We can consider many entity types to describe such an environment, such as DBMS, stored database, DBA, and catalog/data dictionary. Try to specify all the entity types that can fully describe a database system and its environment; then specify the relationship types among them, and draw an ER diagram to describe such a general database environment.

Figure 7.20
An ER diagram for an AIRLINE database schema.



7.21. Design an ER schema for keeping track of information about votes taken in the U.S. House of Representatives during the current two-year congressional session. The database needs to keep track of each U.S. STATE's Name (e.g., 'Texas', 'New York', 'California') and include the Region of the state (whose domain is {'Northeast', 'Midwest', 'Southeast', 'Southwest', 'West'}). Each

CONGRESS_PERSON in the House of Representatives is described by his or her Name, plus the District represented, the Start_date when the congressperson was first elected, and the political Party to which he or she belongs (whose domain is {'Republican', 'Democrat', 'Independent', 'Other'}). The database keeps track of each BILL (i.e., proposed law), including the Bill_name, the Date_of_vote on the bill, whether the bill Passed_or_failed (whose domain is {'Yes', 'No'}), and the Sponsor (the congressperson(s) who sponsored—that is, proposed—the bill). The database also keeps track of how each congressperson voted on each bill (domain of Vote attribute is {'Yes', 'No', 'Abstain', 'Absent'}). Draw an ER schema diagram for this application. State clearly any assumptions you make.

7.22. A database is being constructed to keep track of the teams and games of a sports league. A team has a number of players, not all of whom participate in each game. It is desired to keep track of the players participating in each game for each team, the positions they played in that game, and the result of the game. Design an ER schema diagram for this application, stating any assumptions you make. Choose your favorite sport (e.g., soccer, baseball, football).

7.23. Consider the ER diagram shown in Figure 7.21 for part of a BANK database. Each bank can have multiple branches, and each branch can have multiple accounts and loans.

a. List the strong (nonweak) entity types in the ER diagram.

b. Is there a weak entity type? If so, give its name, partial key, and identifying relationship.

c. What constraints do the partial key and the identifying relationship of the weak entity type specify in this diagram?

d. List the names of all relationship types, and specify the (min, max) constraint on each participation of an entity type in a relationship type. Justify your choices.

e. List concisely the user requirements that led to this ER schema design.

f. Suppose that every customer must have at least one account but is restricted to at most two loans at a time, and that a bank branch cannot have more than 1,000 loans. How does this show up on the (min, max) constraints?

7.24. Consider the ER diagram in Figure 7.22. Assume that an employee may work in up to two departments or may not be assigned to any department. Assume that each department must have one and may have up to three phone numbers. Supply (min, max) constraints on this diagram. *State clearly any additional assumptions you make.* Under what conditions would the relationship HAS_PHONE be redundant in this example?

7.25. Consider the ER diagram in Figure 7.23. Assume that a course may or may not use a textbook, but that a text by definition is a book that is used in some course. A course may not use more than five books. Instructors teach from

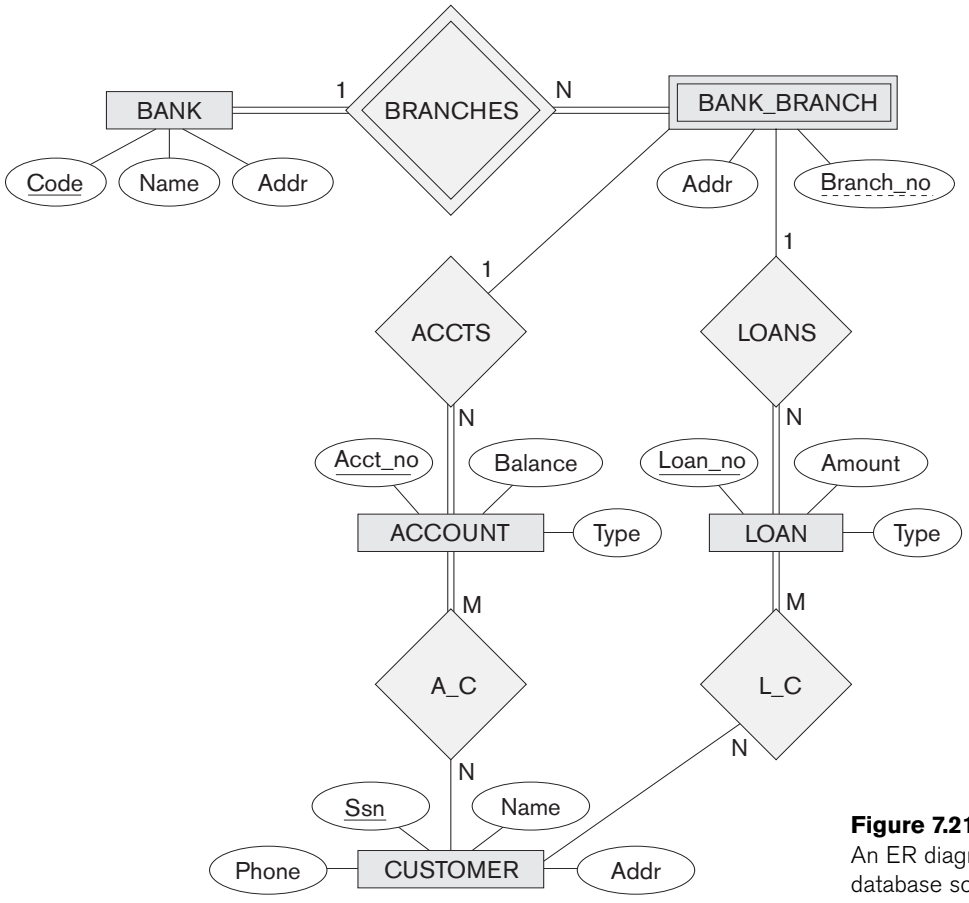


Figure 7.21
An ER diagram for a BANK database schema.

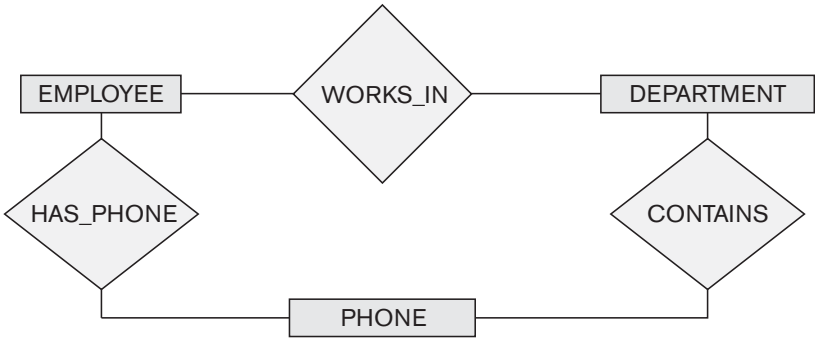


Figure 7.22
Part of an ER diagram for a COMPANY database.

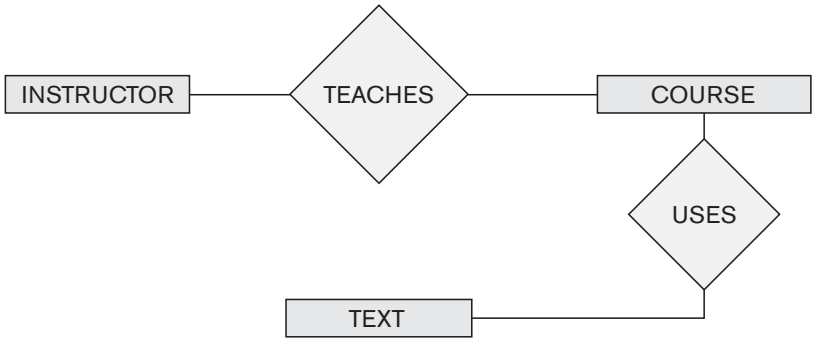


Figure 7.23
Part of an ER diagram for a COURSES database.

two to four courses. Supply (min, max) constraints on this diagram. *State clearly any additional assumptions you make.* If we add the relationship ADOPTS, to indicate the textbook(s) that an instructor uses for a course, should it be a binary relationship between INSTRUCTOR and TEXT, or a ternary relationship between all three entity types? What (min, max) constraints would you put on it? Why?

7.26. Consider an entity type SECTION in a UNIVERSITY database, which describes the section offerings of courses. The attributes of SECTION are Section_number, Semester, Year, Course_number, Instructor, Room_no (where section is taught), Building (where section is taught), Weekdays (domain is the possible combinations of weekdays in which a section can be offered {'MWF', 'MW', 'TT', and so on}), and Hours (domain is all possible time periods during which sections are offered {'9–9:50 A.M.', '10–10:50 A.M.', …, '3:30–4:50 P.M.', '5:30–6:20 P.M.', and so on}). Assume that Section_number is unique for each course within a particular semester/year combination (that is, if a course is offered multiple times during a particular semester, its section offerings are numbered 1, 2, 3, and so on). There are several composite keys for section, and some attributes are components of more than one key. Identify three composite keys, and show how they can be represented in an ER schema diagram.

7.27. Cardinality ratios often dictate the detailed design of a database. The cardinality ratio depends on the real-world meaning of the entity types involved and is defined by the specific application. For the following binary relationships, suggest cardinality ratios based on the common-sense meaning of the entity types. Clearly state any assumptions you make.

	Entity 1	Cardinality Ratio	Entity 2
1.	STUDENT	_____	SOCIAL_SECURITY_CARD
2.	STUDENT	_____	TEACHER
3.	CLASSROOM	_____	WALL

4. COUNTRY	_____	CURRENT_PRESIDENT
5. COURSE	_____	TEXTBOOK
6. ITEM (that can be found in an order)	_____	ORDER
7. STUDENT	_____	CLASS
8. CLASS	_____	INSTRUCTOR
9. INSTRUCTOR	_____	OFFICE
10. EBAY_AUCTION_ITEM	_____	EBAY_BID

7.28. Consider the ER schema for the MOVIES database in Figure 7.24.

Assume that MOVIES is a populated database. ACTOR is used as a generic term and includes actresses. Given the constraints shown in the ER schema, respond to the following statements with *True, False*, or *Maybe*. Assign a response of *Maybe* to statements that, while not explicitly shown to be *True*, cannot be proven *False* based on the schema as shown. Justify each answer.

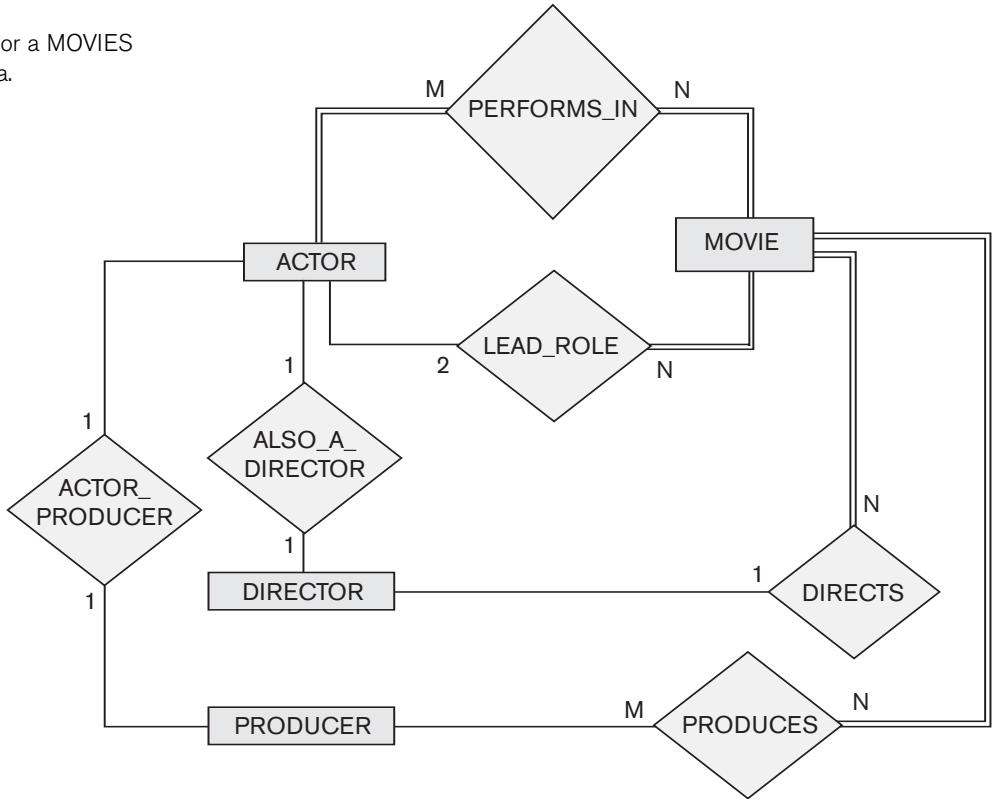


Figure 7.24
An ER diagram for a MOVIES database schema.

a. There are no actors in this database that have been in no movies.
b. There are some actors who have acted in more than ten movies.
c. Some actors have done a lead role in multiple movies.
d. A movie can have only a maximum of two lead actors.
e. Every director has been an actor in some movie.
f. No producer has ever been an actor.
g. A producer cannot be an actor in some other movie.
h. There are movies with more than a dozen actors.
i. Some producers have been a director as well.
j. Most movies have one director and one producer.
k. Some movies have one director but several producers.
l. There are some actors who have done a lead role, directed a movie, and produced some movie.
m. No movie has a director who also acted in that movie.

7.29. Given the ER schema for the MOVIES database in Figure 7.24, draw an instance diagram using three movies that have been released recently. Draw instances of each entity type: MOVIES, ACTORS, PRODUCERS, DIRECTORS involved; make up instances of the relationships as they exist in reality for those movies.

7.30. Illustrate the UML Diagram for Exercise 7.16. Your UML design should observe the following requirements:

a. A student should have the ability to compute his/her GPA and add or drop majors and minors.
b. Each department should be to able add or delete courses and hire or terminate faculty.
c. Each instructor should be able to assign or change a student's grade for a course.

Note: Some of these functions may be spread over multiple classes.

Laboratory Exercises

7.31. Consider the UNIVERSITY database described in Exercise 7.16. Build the ER schema for this database using a data modeling tool such as ERwin or Rational Rose.

7.32. Consider a MAIL_ORDER database in which employees take orders for parts from customers. The data requirements are summarized as follows:

- The mail order company has employees, each identified by a unique employee number, first and last name, and Zip Code.
- Each customer of the company is identified by a unique customer number, first and last name, and Zip Code.

- Each part sold by the company is identified by a unique part number, a part name, price, and quantity in stock.
- Each order placed by a customer is taken by an employee and is given a unique order number. Each order contains specified quantities of one or more parts. Each order has a date of receipt as well as an expected ship date. The actual ship date is also recorded.

Design an Entity-Relationship diagram for the mail order database and build the design using a data modeling tool such as ERwin or Rational Rose.

7.33. Consider a MOVIE database in which data is recorded about the movie industry. The data requirements are summarized as follows:

- Each movie is identified by title and year of release. Each movie has a length in minutes. Each has a production company, and each is classified under one or more genres (such as horror, action, drama, and so forth). Each movie has one or more directors and one or more actors appear in it. Each movie also has a plot outline. Finally, each movie has zero or more quotable quotes, each of which is spoken by a particular actor appearing in the movie.
- Actors are identified by name and date of birth and appear in one or more movies. Each actor has a role in the movie.
- Directors are also identified by name and date of birth and direct one or more movies. It is possible for a director to act in a movie (including one that he or she may also direct).
- Production companies are identified by name and each has an address. A production company produces one or more movies.

Design an Entity-Relationship diagram for the movie database and enter the design using a data modeling tool such as ERwin or Rational Rose.

7.34. Consider a CONFERENCE_REVIEW database in which researchers submit their research papers for consideration. Reviews by reviewers are recorded for use in the paper selection process. The database system caters primarily to reviewers who record answers to evaluation questions for each paper they review and make recommendations regarding whether to accept or reject the paper. The data requirements are summarized as follows:

- Authors of papers are uniquely identified by e-mail id. First and last names are also recorded.
- Each paper is assigned a unique identifier by the system and is described by a title, abstract, and the name of the electronic file containing the paper.
- A paper may have multiple authors, but one of the authors is designated as the contact author.
- Reviewers of papers are uniquely identified by e-mail address. Each reviewer's first name, last name, phone number, affiliation, and topics of interest are also recorded.

- Each paper is assigned between two and four reviewers. A reviewer rates each paper assigned to him or her on a scale of 1 to 10 in four categories: technical merit, readability, originality, and relevance to the conference. Finally, each reviewer provides an overall recommendation regarding each paper.
- Each review contains two types of written comments: one to be seen by the review committee only and the other as feedback to the author(s).

Design an Entity-Relationship diagram for the CONFERENCE_REVIEW database and build the design using a data modeling tool such as ERwin or Rational Rose.

7.35. Consider the ER diagram for the AIRLINE database shown in Figure 7.20. Build this design using a data modeling tool such as ERwin or Rational Rose.

Selected Bibliography

The Entity-Relationship model was introduced by Chen (1976), and related work appears in Schmidt and Swenson (1975), Wiederhold and Elmasri (1979), and Senko (1975). Since then, numerous modifications to the ER model have been suggested. We have incorporated some of these in our presentation. Structural constraints on relationships are discussed in Abrial (1974), Elmasri and Wiederhold (1980), and Lenzerini and Santucci (1983). Multivalued and composite attributes are incorporated in the ER model in Elmasri et al. (1985). Although we did not discuss languages for the ER model and its extensions, there have been several proposals for such languages. Elmasri and Wiederhold (1981) proposed the GORDAS query language for the ER model. Another ER query language was proposed by Markowitz and Raz (1983). Senko (1980) presented a query language for Senko's DIAM model. A formal set of operations called the ER algebra was presented by Parent and Spaccapietra (1985). Gogolla and Hohenstein (1991) presented another formal language for the ER model. Campbell et al. (1985) presented a set of ER operations and showed that they are relationally complete. A conference for the dissemination of research results related to the ER model has been held regularly since 1979. The conference, now known as the International Conference on Conceptual Modeling, has been held in Los Angeles (ER 1979, ER 1983, ER 1997), Washington, D.C. (ER 1981), Chicago (ER 1985), Dijon, France (ER 1986), New York City (ER 1987), Rome (ER 1988), Toronto (ER 1989), Lausanne, Switzerland (ER 1990), San Mateo, California (ER 1991), Karlsruhe, Germany (ER 1992), Arlington, Texas (ER 1993), Manchester, England (ER 1994), Brisbane, Australia (ER 1995), Cottbus, Germany (ER 1996), Singapore (ER 1998), Paris, France (ER 1999), Salt Lake City, Utah (ER 2000), Yokohama, Japan (ER 2001), Tampere, Finland (ER 2002), Chicago, Illinois (ER 2003), Shanghai, China (ER 2004), Klagenfurt, Austria (ER 2005), Tucson, Arizona (ER 2006), Auckland, New Zealand (ER 2007), Barcelona, Catalonia, Spain (ER 2008), and Gramado, RS, Brazil (ER 2009). The 2010 conference is to be held in Vancouver, BC, Canada.