C H A P T E R 1

Introduction

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data. These concepts and technique form the focus of this book. This chapter briefly introduces the principles of database systems.

1.1 Database System Applications

Databases are widely used. Here are some representative applications:

- *Banking*: For customer information, accounts, and loans, and banking transactions.
- *Airlines*: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner—terminals situated around the world accessed the central database system through phone lines and other data networks.
- *Universities*: For student information, course registrations, and grades.

- *Credit card transactions*: For purchases on credit cards and generation of monthly statements.
- *Telecommunication*: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.
- *Finance*: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds.
- *Sales*: For customer, product, and purchase information.
- *Manufacturing*: For management of supply chain and for tracking production of items in factories, inventories of items in warehouses/stores, and orders for items.
- *Human resources*: For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.

As the list illustrates, databases form an essential part of almost all enterprises today.

Over the course of the last four decades of the twentieth century, use of databases grew in all enterprises. In the early days, very few people interacted directly with database systems, although without realizing it they interacted with databases indirectly—through printed reports such as credit card statements, or through agents such as bank tellers and airline reservation agents. Then automated teller machines came along and let users interact directly with databases. Phone interfaces to computers (interactive voice response systems) also allowed users to deal directly with databases—a caller could dial a number, and press phone keys to enter information or to select alternative options, to find flight arrival/departure times, for example, or to register for courses in a university.

The internet revolution of the late 1990s sharply increased direct user access to databases. Organizations converted many of their phone interfaces to databases into Web interfaces, and made a variety of services and information available online. For instance, when you access an online bookstore and browse a book or music collection, you are accessing data stored in a database. When you enter an order online, your order is stored in a database. When you access a bank Web site and retrieve your bank balance and transaction information, the information is retrieved from the bank's database system. When you access a Web site, information about you may be retrieved from a database, to select which advertisements should be shown to you. Furthermore, data about your Web accesses may be stored in a database.

Thus, although user interfaces hide details of access to a database, and most people are not even aware they are dealing with a database, accessing databases forms an essential part of almost everyone's life today.

The importance of database systems can be judged in another way—today, database system vendors like Oracle are among the largest software companies in the world, and database systems form an important part of the product line of more diversified companies like Microsoft and IBM.

1.2 Database Systems versus File Systems

Consider part of a savings-bank enterprise that keeps information about all customers and savings accounts. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including

- A program to debit or credit an account
- A program to add a new account
- A program to find the balance of an account
- A program to generate monthly statements

System programmers wrote these application programs to meet the needs of the bank.

New application programs are added to the system as the need arises. For example, suppose that the savings bank decides to offer checking accounts. As a result, the bank creates new permanent files that contain information about all the checking accounts maintained in the bank, and it may have to write new application programs to deal with situations that do not arise in savings accounts, such as overdrafts. Thus, as time goes by, the system acquires more files and more application programs.

This typical **file-processing system** is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) came along, organizations usually stored information in such systems.

Keeping organizational information in a file-processing system has a number of major disadvantages:

- **Data redundancy and inconsistency**. Since different programmers create the files and application programs over a long period, the various files are likely to have different formats and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, the address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.

- **Difficulty in accessing data**. Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* customers. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same officer needs to trim that list to include only those customers who have an account balance of $10,000 or more. As expected, a program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory.

The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

- **Data isolation**. Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

- **Integrity problems**. The data values stored in the database must satisfy certain types of **consistency constraints**. For example, the balance of a bank account may never fall below a prescribed amount (say, $25). Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

- **Atomicity problems**. A computer system, like any other mechanical or electrical device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $50 from account *A* to account *B*. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account *A* but was not credited to account *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

- **Concurrent-access anomalies**. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data. Consider bank account *A*, containing $500. If two customers withdraw funds (say $50 and $100 respectively) from account *A* at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $500, and write back $450 and $400, respectively. Depending on which one writes the value last, the account may contain either $450 or $400, rather than the correct value of $350. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

- **Security problems**. Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the system in an ad hoc manner, enforcing such security constraints is difficult.

These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems. In most of this book, we use a bank enterprise as a running example of a typical data-processing application found in a corporation.

1.3 View of Data

A database system is a collection of interrelated files and a set of programs that allow users to access and modify these files. A major purpose of a database system is to provide users with an *abstract* view of the data. That is, the system hides certain details of how the data are stored and maintained.

1.3.1 Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led designers to use complex data structures to represent data in the database. Since many database-systems users are not computer trained, developers hide the complexity from users through several levels of abstraction, to simplify users' interactions with the system:

- **Physical level**. The lowest level of abstraction describes *how* the data are actually stored. The physical level describes complex low-level data structures in detail.

- **Logical level**. The next-higher level of abstraction describes *what* data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical-level structures, the user of the logical level does not need to be aware of this complexity. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.

- **View level**. The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

Figure 1.1 shows the relationship among the three levels of abstraction.

An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Most high-level programming languages support the notion of a record type. For example, in a Pascal-like language, we may declare a record as follows:

```
type customer = record
                    customer-id : string;
                    customer-name : string;
                    customer-street : string;
                    customer-city : string;
                end;
```

This code defines a new record type called *customer* with four fields. Each field has a name and a type associated with it. A banking enterprise may have several such record types, including

- *account*, with fields *account-number* and *balance*
- *employee*, with fields *employee-name* and *salary*

At the physical level, a *customer*, *account*, or *employee* record can be described as a block of consecutive storage locations (for example, words or bytes). The language

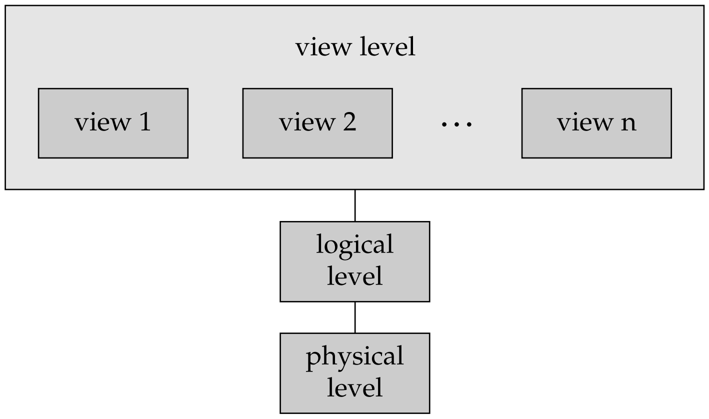

Figure 1.1 The three levels of data abstraction.

compiler hides this level of detail from programmers. Similarly, the database system hides many of the lowest-level storage details from database programmers. Database administrators, on the other hand, may be aware of certain details of the physical organization of the data.

At the logical level, each such record is described by a type definition, as in the previous code segment, and the interrelationship of these record types is defined as well. Programmers using a programming language work at this level of abstraction. Similarly, database administrators usually work at this level of abstraction.

Finally, at the view level, computer users see a set of application programs that hide details of the data types. Similarly, at the view level, several views of the database are defined, and database users see these views. In addition to hiding details of the logical level of the database, the views also provide a security mechanism to prevent users from accessing certain parts of the database. For example, tellers in a bank see only that part of the database that has information on customer accounts; they cannot access information about salaries of employees.

1.3.2 Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema.

Database systems have several schemas, partitioned according to the levels of abstraction. The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas**, that describe different views of the database.

Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

We study languages for describing schemas, after introducing the notion of data models in the next section.

1.4 Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. To illustrate the concept of a data model, we outline two data models in this section: the entity-relationship model and the relational model. Both provide a way to describe the design of a database at the logical level.

1.4.1 The Entity-Relationship Model

The entity-relationship (E-R) data model is based on a perception of a real world that consists of a collection of basic objects, called *entities*, and of *relationships* among these objects. An entity is a “thing” or “object” in the real world that is distinguishable from other objects. For example, each person is an entity, and bank accounts can be considered as entities.

Entities are described in a database by a set of **attributes**. For example, the attributes *account-number* and *balance* may describe one particular account in a bank, and they form attributes of the *account* entity set. Similarly, attributes *customer-name*, *customer-street* address and *customer-city* may describe a *customer* entity.

An extra attribute *customer-id* is used to uniquely identify customers (since it may be possible to have two customers with the same name, street address, and city). A unique customer identifier must be assigned to each customer. In the United States, many enterprises use the social-security number of a person (a unique number the U.S. government assigns to every person in the United States) as a customer identifier.

A **relationship** is an association among several entities. For example, a *depositor* relationship associates a customer with each account that she has. The set of all entities of the same type and the set of all relationships of the same type are termed an **entity set** and **relationship set**, respectively.

The overall logical structure (schema) of a database can be expressed graphically by an *E-R diagram*, which is built up from the following components:

- **Rectangles**, which represent entity sets
- **Ellipses**, which represent attributes
- **Diamonds**, which represent relationships among entity sets
- **Lines**, which link attributes to entity sets and entity sets to relationships

Each component is labeled with the entity or relationship that it represents.

As an illustration, consider part of a database banking system consisting of customers and of the accounts that these customers have. Figure 1.2 shows the corresponding E-R diagram. The E-R diagram indicates that there are two entity sets, *customer* and *account*, with attributes as outlined earlier. The diagram also shows a relationship *depositor* between customer and account.

In addition to entities and relationships, the E-R model represents certain constraints to which the contents of a database must conform. One important constraint is **mapping cardinalities**, which express the number of entities to which another entity can be associated via a relationship set. For example, if each account must belong to only one customer, the E-R model can express that constraint.

The entity-relationship model is widely used in database design, and Chapter 2 explores it in detail.

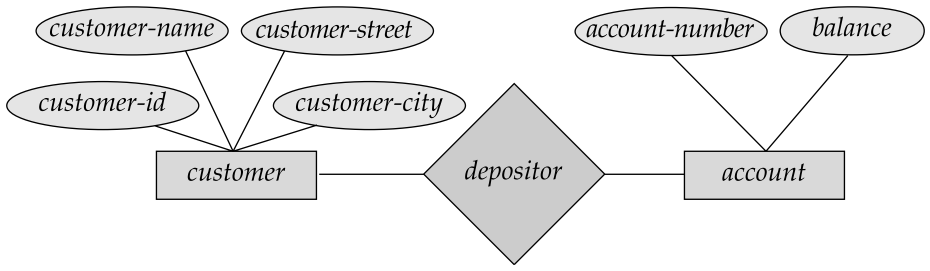

Figure 1.2 A sample E-R diagram.

1.4.2 Relational Model

The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Figure 1.3 presents a sample relational database comprising three tables: One shows details of bank customers, the second shows accounts, and the third shows which accounts belong to which customers.

The first table, the *customer* table, shows, for example, that the customer identified by customer-id 192-83-7465 is named Johnson and lives at 12 Alma St. in Palo Alto. The second table, *account*, shows, for example, that account A-101 has a balance of $500, and A-201 has a balance of $900.

The third table shows which accounts belong to which customers. For example, account number A-101 belongs to the customer whose customer-id is 192-83-7465, namely Johnson, and customers 192-83-7465 (Johnson) and 019-28-3746 (Smith) share account number A-201 (they may share a business venture).

The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type.

It is not hard to see how tables may be stored in files. For instance, a special character (such as a comma) may be used to delimit the different attributes of a record, and another special character (such as a newline character) may be used to delimit records. The relational model hides such low-level implementation details from database developers and users.

The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model. Chapters 3 through 7 cover the relational model in detail.

The relational model is at a lower level of abstraction than the E-R model. Database designs are often carried out in the E-R model, and then translated to the relational model; Chapter 2 describes the translation process. For example, it is easy to see that the tables *customer* and *account* correspond to the entity sets of the same name, while the table *depositor* corresponds to the relationship set *depositor*.

We also note that it is possible to create schemas in the relational model that have problems such as unnecessarily duplicated information. For example, suppose we

customer-id	*customer-name*	*customer-street*	*customer-city*
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

account-number	*balance*
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

customer-id	*account-number*
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

Figure 1.3 A sample relational database.

store *account-number* as an attribute of the *customer* record. Then, to represent the fact that accounts A-101 and A-201 both belong to customer Johnson (with customer-id 192-83-7465), we would need to store two rows in the *customer* table. The values for customer-name, customer-street, and customer-city for Johnson would get unnecessarily duplicated in the two rows. In Chapter 7, we shall study how to distinguish good schema designs from bad schema designs.

1.4.3 Other Data Models

The **object-oriented data model** is another data model that has seen increasing attention. The object-oriented model can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. Chapter 8 examines the object-oriented data model.

The **object-relational data model** combines features of the object-oriented data model and relational data model. Chapter 9 examines it.

Semistructured data models permit the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast with the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **extensible markup language (XML)** is widely used to represent semistructured data. Chapter 10 covers it.

Historically, two other data models, the **network data model** and the **hierarchical data model**, preceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data. As a result they are little used now, except in old database code that is still in service in some places. They are outlined in Appendices A and B, for interested readers.

1.5 Database Languages

A database system provides a **data definition language** to specify the database schema and a **data manipulation language** to express database queries and updates. In practice, the data definition and data manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

1.5.1 Data-Definition Language

We specify a database schema by a set of definitions expressed by a special language called a **data-definition language** (**DDL**).

For instance, the following statement in the SQL language defines the *account* table:

```
create table account
       (account-number char(10),
        balance integer)
```

Execution of the above DDL statement creates the *account* table. In addition, it updates a special set of tables called the **data dictionary** or **data directory**.

A data dictionary contains **metadata**—that is, data about data. The schema of a table is an example of metadata. A database system consults the data dictionary before reading or modifying actual data.

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a **data storage and definition** language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

The data values stored in the database must satisfy certain **consistency constraints**. For example, suppose the balance on an account should not fall below $100. The DDL provides facilities to specify such constraints. The database systems check these constraints every time the database is updated.

1.5.2 Data-Manipulation Language

Data manipulation is

- The retrieval of information stored in the database
- The insertion of new information into the database
- The deletion of information from the database
- The modification of information stored in the database

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model. There are basically two types:

- **Procedural DMLs** require a user to specify *what* data are needed and *how* to get those data.
- **Declarative DMLs** (also referred to as **nonprocedural** DMLs) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data. The DML component of the SQL language is nonprocedural.

A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data-manipulation language* synonymously.

This query in the SQL language finds the name of the customer whose customer-id is 192-83-7465:

```
select customer.customer-name
from customer
where customer.customer-id = 192-83-7465
```

The query specifies that those rows *from* the table *customer where* the *customer-id* is 192-83-7465 must be retrieved, and the *customer-name* attribute of these rows must be displayed. If the query were run on the table in Figure 1.3, the name Johnson would be displayed.

Queries may involve information from more than one table. For instance, the following query finds the balance of all accounts owned by the customer with customer-id 192-83-7465.

```
select account.balance
from depositor, account
where depositor.customer-id = 192-83-7465 and
      depositor.account-number = account.account-number
```

If the above query were run on the tables in Figure 1.3, the system would find that the two accounts numbered A-101 and A-201 are owned by customer 192-83-7465 and would print out the balances of the two accounts, namely 500 and 900.

There are a number of database query languages in use, either commercially or experimentally. We study the most widely used query language, SQL, in Chapter 4. We also study some other query languages in Chapter 5.

The levels of abstraction that we discussed in Section 1.3 apply not only to defining or structuring data, but also to manipulating data. At the physical level, we must define algorithms that allow efficient access to data. At higher levels of abstraction, we emphasize ease of use. The goal is to allow humans to interact efficiently with the system. The query processor component of the database system (which we study in Chapters 13 and 14) translates DML queries into sequences of actions at the physical level of the database system.

1.5.3 Database Access from Application Programs

Application programs are programs that are used to interact with the database. Application programs are usually written in a *host* language, such as Cobol, C, C++, or Java. Examples in a banking system are programs that generate payroll checks, debit accounts, credit accounts, or transfer funds between accounts.

To access the database, DML statements need to be executed from the host language. There are two ways to do this:

- By providing an application program interface (set of procedures) that can be used to send DML and DDL statements to the database, and retrieve the results.

  The Open Database Connectivity (ODBC) standard defined by Microsoft for use with the C language is a commonly used application program interface standard. The Java Database Connectivity (JDBC) standard provides corresponding features to the Java language.

- By extending the host language syntax to embed DML calls within the host language program. Usually, a special character prefaces DML calls, and a preprocessor, called the *DML* **precompiler**, converts the DML statements to normal procedure calls in the host language.

1.6 Database Users and Administrators

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

1.6.1 Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

- **Naive users** are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*. This program asks the teller for the amount of money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred.

  As another example, consider a user who wishes to find her account balance over the World Wide Web. Such a user may access a form, where she enters her account number. An application program at the Web server then retrieves the account balance, using the given account number, and passes this information back to the user.

  The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naive users may also simply read *reports* generated from the database.

- **Application programmers** are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. **Rapid application development (RAD)** tools are tools that enable an application programmer to construct forms and reports without writing a program. There are also special types of programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language. These languages, sometimes called *fourth-generation languages*, often include special features to facilitate the generation of forms and the display of data on the screen. Most major commercial database systems include a fourth-generation language.

- **Sophisticated users** interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category.

  Online analytical processing (OLAP) tools simplify analysts' tasks by letting them view summaries of data in different ways. For instance, an analyst can see total sales by region (for example, North, South, East, and West), or by product, or by a combination of region and product (that is, total sales of each product in each region). The tools also permit the analyst to select specific regions, look at data in more detail (for example, sales by city within a region) or look at the data in less detail (for example, aggregate products together by category).

  Another class of tools for analysts is **data mining** tools, which help them find certain kinds of patterns in data.

  We study OLAP tools and data mining in Chapter 22.

- **Specialized users** are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems, knowledgebase and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems. Chapters 8 and 9 cover several of these applications.

1.6.2 Database Administrator

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator** (**DBA**). The functions of a DBA include:

- **Schema definition**. The DBA creates the original database schema by executing a set of data definition statements in the DDL.
- **Storage structure and access-method definition**.
- **Schema and physical-organization modification**. The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.
- **Granting of authorization for data access**. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
- **Routine maintenance**. Examples of the database administrator's routine maintenance activities are:
  - ☐ Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
  - ☐ Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
  - ☐ Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

1.7 Transaction Management

Often, several operations on the database form a single logical unit of work. An example is a funds transfer, as in Section 1.2, in which one account (say A) is debited and another account (say B) is credited. Clearly, it is essential that either both the credit and debit occur, or that neither occur. That is, the funds transfer must happen in its entirety or not at all. This all-or-none requirement is called **atomicity**. In addition, it is essential that the execution of the funds transfer preserve the consistency of the database. That is, the value of the sum $A + B$ must be preserved. This correctness requirement is called **consistency**. Finally, after the successful execution of a funds transfer, the new values of accounts A and B must persist, despite the possibility of system failure. This persistence requirement is called **durability**.

A **transaction** is a collection of operations that performs a single logical function in a database application. Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. However, during the execution of a transaction, it may be necessary temporarily to allow inconsistency, since either the debit of *A* or the credit of *B* must be done before the other. This temporary inconsistency, although necessary, may lead to difficulty if a failure occurs.

It is the programmer's responsibility to define properly the various transactions, so that each preserves the consistency of the database. For example, the transaction to transfer funds from account *A* to account *B* could be defined to be composed of two separate programs: one that debits account *A*, and another that credits account *B*. The execution of these two programs one after the other will indeed preserve consistency. However, each program by itself does not transform the database from a consistent state to a new consistent state. Thus, those programs are not transactions.

Ensuring the atomicity and durability properties is the responsibility of the database system itself—specifically, of the **transaction-management component**. In the absence of failures, all transactions complete successfully, and atomicity is achieved easily. However, because of various types of failure, a transaction may not always complete its execution successfully. If we are to ensure the atomicity property, a failed transaction must have no effect on the state of the database. Thus, the database must be restored to the state in which it was before the transaction in question started executing. The database system must therefore perform **failure recovery**, that is, detect system failures and restore the database to the state that existed prior to the occurrence of the failure.

Finally, when several transactions update the database concurrently, the consistency of data may no longer be preserved, even though each individual transaction is correct. It is the responsibility of the **concurrency-control manager** to control the interaction among the concurrent transactions, to ensure the consistency of the database.

Database systems designed for use on small personal computers may not have all these features. For example, many small systems allow only one user to access the database at a time. Others do not offer backup and recovery, leaving that to the user. These restrictions allow for a smaller data manager, with fewer requirements for physical resources—especially main memory. Although such a low-cost, low-feature approach is adequate for small personal databases, it is inadequate for a medium- to large-scale enterprise.

1.8 Database System Structure

A database system is partitioned into modules that deal with each of the responsibilites of the overall system. The functional components of a database system can be broadly divided into the storage manager and the query processor components.

The storage manager is important because databases typically require a large amount of storage space. Corporate databases range in size from hundreds of gigabytes to, for the largest databases, terabytes of data. A gigabyte is 1000 megabytes

(1 billion bytes), and a terabyte is 1 million megabytes (1 trillion bytes). Since the main memory of computers cannot store this much information, the information is stored on disks. Data are moved between disk storage and main memory as needed. Since the movement of data to and from disk is slow relative to the speed of the central processing unit, it is imperative that the database system structure the data so as to minimize the need to move data between disk and main memory.

The query processor is important because it helps the database system simplify and facilitate access to data. High-level views help to achieve this goal; with them, users of the system are not be burdened unnecessarily with the physical details of the implementation of the system. However, quick processing of updates and queries is important. It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level.

1.8.1 Storage Manager

A *storage manager* is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The raw data are stored on the disk using the file system, which is usually provided by a conventional operating system. The storage manager translates the various DML statements into low-level file-system commands. Thus, the storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

- **Authorization and integrity manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
- **Transaction manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
- **File manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
- **Buffer manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

The storage manager implements several data structures as part of the physical system implementation:

- **Data files**, which store the database itself.
- **Data dictionary**, which stores metadata about the structure of the database, in particular the schema of the database.
- **Indices**, which provide fast access to data items that hold particular values.

1.8.2 The Query Processor

The query processor components include

- **DDL interpreter**, which interprets DDL statements and records the definitions in the data dictionary.
- **DML** compiler, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.

  A query can usually be translated into any of a number of alternative evaluation plans that all give the same result. The DML compiler also performs **query optimization**, that is, it picks the lowest cost evaluation plan from among the alternatives.
- **Query evaluation engine**, which executes low-level instructions generated by the DML compiler.

Figure 1.4 shows these components and the connections among them.

1.9 Application Architectures

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between **client** machines, on which remote database users work, and **server** machines, on which the database system runs.

Database applications are usually partitioned into two or three parts, as in Figure 1.5. In a **two-tier architecture**, the application is partitioned into a component that resides at the client machine, which invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

In contrast, in a **three-tier architecture**, the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an **application server**, usually through a forms interface. The application server in turn communicates with a database system to access data. The **business logic** of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web.

1.10 History of Database Systems

Data processing drives the growth of computers, as it has from the earliest days of commercial computers. In fact, automation of data processing tasks predates computers. Punched cards, invented by Hollerith, were used at the very beginning of the twentieth century to record U.S. census data, and mechanical systems were used to

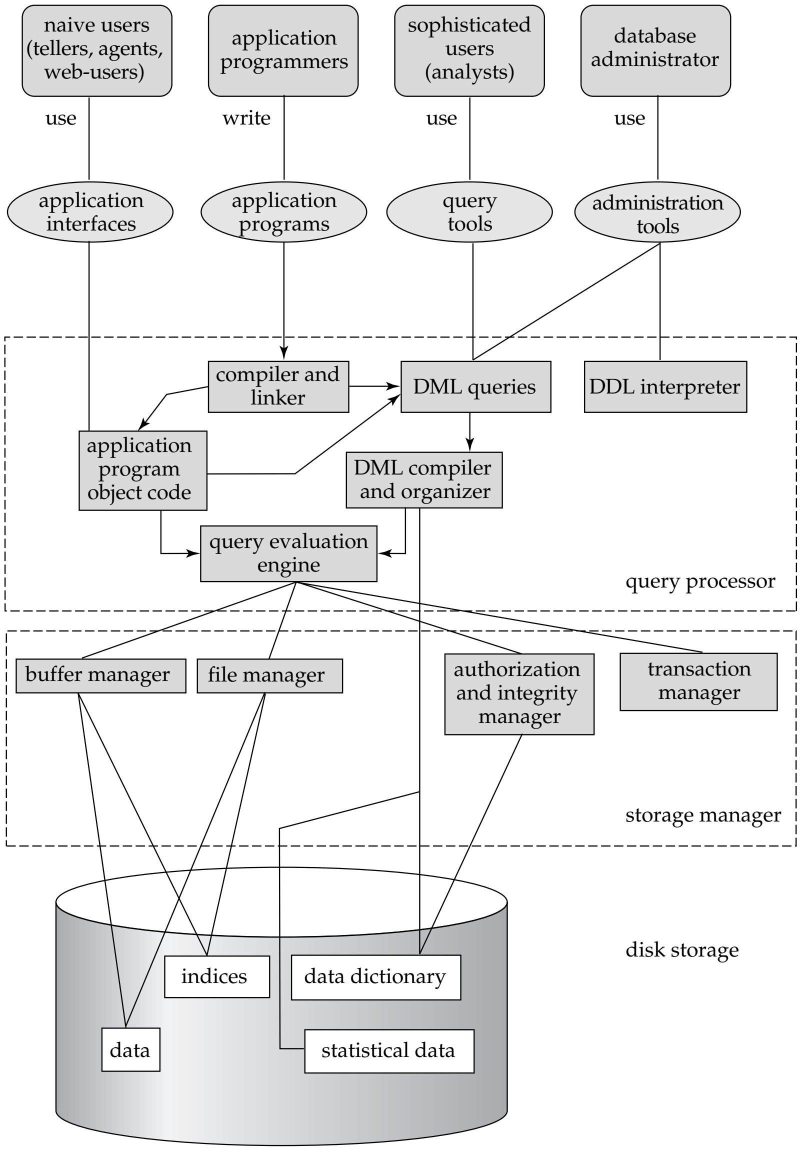

Figure 1.4 System structure.

process the cards and tabulate results. Punched cards were later widely used as a means of entering data into computers.

Techniques for data storage and processing have evolved over the years:

- **1950s and early 1960s**: Magnetic tapes were developed for data storage. Data processing tasks such as payroll were automated, with data stored on tapes. Processing of data consisted of reading data from one or more tapes and

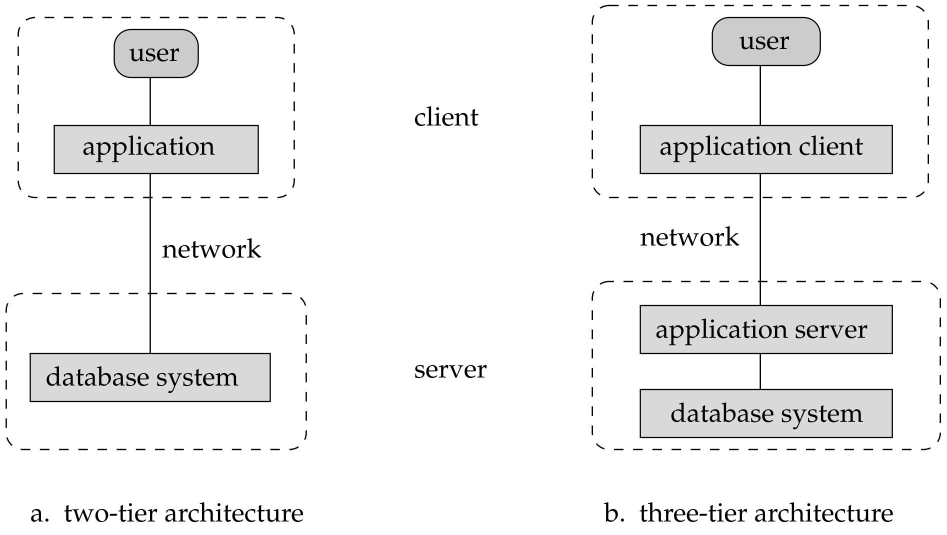

Figure 1.5 Two-tier and three-tier architectures.

writing data to a new tape. Data could also be input from punched card decks, and output to printers. For example, salary raises were processed by entering the raises on punched cards and reading the punched card deck in synchronization with a tape containing the master salary details. The records had to be in the same sorted order. The salary raises would be added to the salary read from the master tape, and written to a new tape; the new tape would become the new master tape.

Tapes (and card decks) could be read only sequentially, and data sizes were much larger than main memory; thus, data processing programs were forced to process data in a particular order, by reading and merging data from tapes and card decks.

- **Late 1960s and 1970s**: Widespread use of hard disks in the late 1960s changed the scenario for data processing greatly, since hard disks allowed direct access to data. The position of data on disk was immaterial, since any location on disk could be accessed in just tens of milliseconds. Data were thus freed from the tyranny of sequentiality. With disks, network and hierarchical databases could be created that allowed data structures such as lists and trees to be stored on disk. Programmers could construct and manipulate these data structures.

  A landmark paper by Codd [1970] defined the relational model, and nonprocedural ways of querying data in the relational model, and relational databases were born. The simplicity of the relational model and the possibility of hiding implementation details completely from the programmer were enticing indeed. Codd later won the prestigious Association of Computing Machinery Turing Award for his work.

- **1980s**: Although academically interesting, the relational model was not used in practice initially, because of its perceived performance disadvantages; relational databases could not match the performance of existing network and hierarchical databases. That changed with System R, a groundbreaking project at IBM Research that developed techniques for the construction of an efficient relational database system. Excellent overviews of System R are provided by Astrahan et al. [1976] and Chamberlin et al. [1981]. The fully functional System R prototype led to IBM's first relational database product, SQL/DS. Initial commercial relational database systems, such as IBM DB2, Oracle, Ingres, and DEC Rdb, played a major role in advancing techniques for efficient processing of declarative queries. By the early 1980s, relational databases had become competitive with network and hierarchical database systems even in the area of performance. Relational databases were so easy to use that they eventually replaced network/hierarchical databases; programmers using such databases were forced to deal with many low-level implementation details, and had to code their queries in a procedural fashion. Most importantly, they had to keep efficiency in mind when designing their programs, which involved a lot of effort. In contrast, in a relational database, almost all these low-level tasks are carried out automatically by the database, leaving the programmer free to work at a logical level. Since attaining dominance in the 1980s, the relational model has reigned supreme among data models.

  The 1980s also saw much research on parallel and distributed databases, as well as initial work on object-oriented databases.

- **Early 1990s**: The SQL language was designed primarily for decision support applications, which are query intensive, yet the mainstay of databases in the 1980s was transaction processing applications, which are update intensive. Decision support and querying re-emerged as a major application area for databases. Tools for analyzing large amounts of data saw large growths in usage.

  Many database vendors introduced parallel database products in this period. Database vendors also began to add object-relational support to their databases.

- **Late 1990s**: The major event was the explosive growth of the World Wide Web. Databases were deployed much more extensively than ever before. Database systems now had to support very high transaction processing rates, as well as very high reliability and 24×7 availability (availability 24 hours a day, 7 days a week, meaning no downtime for scheduled maintenance activities). Database systems also had to support Web interfaces to data.

1.11 Summary

- A **database-management system** (DBMS) consists of a collection of interrelated data and a collection of programs to access that data. The data describe one particular enterprise.

- The primary goal of a DBMS is to provide an environment that is both convenient and efficient for people to use in retrieving and storing information.
- Database systems are ubiquitous today, and most people interact, either directly or indirectly, with databases many times every day.
- Database systems are designed to store large bodies of information. The management of data involves both the definition of structures for the storage of information and the provision of mechanisms for the manipulation of information. In addition, the database system must provide for the safety of the information stored, in the face of system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.
- A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained.
- Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and data constraints. The entity-relationship (E-R) data model is a widely used data model, and it provides a convenient graphical representation to view data, relationships and constraints. The relational data model is widely used to store data in databases. Other data models are the object-oriented model, the object-relational model, and semistructured data models.
- The overall design of the database is called the database **schema**. A database schema is specified by a set of definitions that are expressed using a **data-definition language (DDL)**.
- A **data-manipulation language (DML)** is a language that enables users to access or manipulate data. Nonprocedural DMLs, which require a user to specify only what data are needed, without specifying exactly how to get those data, are widely used today.
- Database users can be categorized into several classes, and each class of users usually uses a different type of interface to the database.
- A database system has several subsystems.
  - ☐ The **transaction manager** subsystem is responsible for ensuring that the database remains in a consistent (correct) state despite system failures. The transaction manager also ensures that concurrent transaction executions proceed without conflicting.
  - ☐ The **query processor** subsystem compiles and executes DDL and DML statements.
  - ☐ The **storage manager** subsystem provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

- Database applications are typically broken up into a front-end part that runs at client machines and a part that runs at the back end. In two-tier architectures, the front-end directly communicates with a database running at the back end. In three-tier architectures, the back end part is itself broken up into an application server and a database server.

Review Terms

- Database management system (DBMS)
- Database systems applications
- File systems
- Data inconsistency
- Consistency constraints
- Data views
- Data abstraction
- Database instance
- Schema
  - ☐ Database schema
  - ☐ Physical schema
  - ☐ Logical schema
- Physical data independence
- Data models
  - ☐ Entity-relationship model
  - ☐ Relational data model
  - ☐ Object-oriented data model
  - ☐ Object-relational data model
- Database languages
  - ☐ Data definition language
  - ☐ Data manipulation language
  - ☐ Query language
- Data dictionary
- Metadata
- Application program
- Database administrator (DBA)
- Transactions
- Concurrency
- Client and server machines

Exercises

1.1 List four significant differences between a file-processing system and a DBMS.

1.2 This chapter has described several major advantages of a database system. What are two disadvantages?

1.3 Explain the difference between physical and logical data independence.

1.4 List five responsibilities of a database management system. For each responsibility, explain the problems that would arise if the responsibility were not discharged.

1.5 What are five main functions of a database administrator?

1.6 List seven programming languages that are procedural and two that are nonprocedural. Which group is easier to learn and use? Explain your answer.

1.7 List six major steps that you would take in setting up a database for a particular enterprise.

1.8 Consider a two-dimensional integer array of size $n \times m$ that is to be used in your favorite programming language. Using the array as an example, illustrate the difference (a) between the three levels of data abstraction, and (b) between a schema and instances.

Bibliographical Notes

We list below general purpose books, research paper collections, and Web sites on databases. Subsequent chapters provide references to material on each topic outlined in this chapter.

Textbooks covering database systems include Abiteboul et al. [1995], Date [1995], Elmasri and Navathe [2000], O'Neil and O'Neil [2000], Ramakrishnan and Gehrke [2000], and Ullman [1988]. Textbook coverage of transaction processing is provided by Bernstein and Newcomer [1997] and Gray and Reuter [1993].

Several books contain collections of research papers on database management. Among these are Bancilhon and Buneman [1990], Date [1986], Date [1990], Kim [1995], Zaniolo et al. [1997], and Stonebraker and Hellerstein [1998].

A review of accomplishments in database management and an assessment of future research challenges appears in Silberschatz et al. [1990], Silberschatz et al. [1996] and Bernstein et al. [1998]. The home page of the ACM Special Interest Group on Management of Data (see www.acm.org/sigmod) provides a wealth of information about database research. Database vendor Web sites (see the tools section below) provide details about their respective products.

Codd [1970] is the landmark paper that introduced the relational model. Discussions concerning the evolution of DBMSs and the development of database technology are offered by Fry and Sibley [1976] and Sibley [1976].

Tools

There are a large number of commercial database systems in use today. The major ones include: IBM DB2 (www.ibm.com/software/data), Oracle (www.oracle.com), Microsoft SQL Server (www.microsoft.com/sql), Informix (www.informix.com), and Sybase (www.sybase.com). Some of these systems are available free for personal or noncommercial use, or for development, but are not free for actual deployment.

There are also a number of free/public domain database systems; widely used ones include MySQL (www.mysql.com) and PostgresSQL (www.postgressql.org).

A more complete list of links to vendor Web sites and other information is available from the home page of this book, at www.research.bell-labs.com/topic/books/db-book.