1 INTRODUCTION TO DATABASE SYSTEMS

> Has everyone noticed that all the letters of the word *database* are typed with the left hand? Now the layout of the QWERTY typewriter keyboard was designed, among other things, to facilitate the even use of both hands. It follows, therefore, that writing about databases is not only unnatural, but a lot harder than it appears.
>
> —Anonymous

Today, more than at any previous time, the success of an organization depends on its ability to acquire accurate and timely data about its operations, to manage this data effectively, and to use it to analyze and guide its activities. Phrases such as the *information superhighway* have become ubiquitous, and information processing is a rapidly growing multibillion dollar industry.

The amount of information available to us is literally exploding, and the value of data as an organizational asset is widely recognized. Yet without the ability to manage this vast amount of data, and to quickly find the information that is relevant to a given question, as the amount of information increases, it tends to become a distraction and a liability, rather than an asset. This paradox drives the need for increasingly powerful and flexible data management systems. To get the most out of their large and complex datasets, users must have tools that simplify the tasks of managing the data and extracting useful information in a timely fashion. Otherwise, data can become a liability, with the cost of acquiring it and managing it far exceeding the value that is derived from it.

A **database** is a collection of data, typically describing the activities of one or more related organizations. For example, a university database might contain information about the following:

- *Entities* such as students, faculty, courses, and classrooms.
- *Relationships* between entities, such as students' enrollment in courses, faculty teaching courses, and the use of rooms for courses.

A **database management system**, or **DBMS**, is software designed to assist in maintaining and utilizing large collections of data, and the need for such systems, as well as their use, is growing rapidly. The alternative to using a DBMS is to use ad

hoc approaches that do not carry over from one application to another; for example, to store the data in files and write application-specific code to manage it. The use of a DBMS has several important advantages, as we will see in Section 1.4.

The area of database management systems is a microcosm of computer science in general. The issues addressed and the techniques used span a wide spectrum, including languages, object-orientation and other programming paradigms, compilation, operating systems, concurrent programming, data structures, algorithms, theory, parallel and distributed systems, user interfaces, expert systems and artificial intelligence, statistical techniques, and dynamic programming. We will not be able to go into all these aspects of database management in this book, but it should be clear that this is a rich and vibrant discipline.

1.1 OVERVIEW

The goal of this book is to present an in-depth introduction to database management systems, with an emphasis on how to organize information in a DBMS and to maintain it and retrieve it efficiently, that is, how to *design* a database and *use* a DBMS effectively. Not surprisingly, many decisions about how to use a DBMS for a given application depend on what capabilities the DBMS supports efficiently. Thus, to use a DBMS well, it is necessary to also understand how a DBMS *works*. The approach taken in this book is to emphasize how to *use* a DBMS, while covering DBMS implementation and architecture in sufficient detail to understand how to *design a database*.

Many kinds of database management systems are in use, but this book concentrates on *relational* systems, which are by far the dominant type of DBMS today. The following questions are addressed in the core chapters of this book:

1. **Database Design:** How can a user describe a real-world enterprise (e.g., a university) in terms of the data stored in a DBMS? What factors must be considered in deciding how to organize the stored data? (Chapters 2, 3, 15, 16, and 17.)
2. **Data Analysis:** How can a user answer questions about the enterprise by posing queries over the data in the DBMS? (Chapters 4, 5, 6, and 23.)
3. **Concurrency and Robustness:** How does a DBMS allow many users to access data concurrently, and how does it protect the data in the event of system failures? (Chapters 18, 19, and 20.)
4. **Efficiency and Scalability:** How does a DBMS store large datasets and answer questions against this data efficiently? (Chapters 7, 8, 9, 10, 11, 12, 13, and 14.)

Later chapters cover important and rapidly evolving topics such as parallel and distributed database management, Internet databases, data warehousing and complex

queries for decision support, data mining, object databases, spatial data management, and rule-oriented DBMS extensions.

In the rest of this chapter, we introduce the issues listed above. In Section 1.2, we begin with a brief history of the field and a discussion of the role of database management in modern information systems. We then identify benefits of storing data in a DBMS instead of a file system in Section 1.3, and discuss the advantages of using a DBMS to manage data in Section 1.4. In Section 1.5 we consider how information about an enterprise should be organized and stored in a DBMS. A user probably thinks about this information in high-level terms corresponding to the entities in the organization and their relationships, whereas the DBMS ultimately stores data in the form of (many, many) bits. The gap between how users think of their data and how the data is ultimately stored is bridged through several *levels of abstraction* supported by the DBMS. Intuitively, a user can begin by describing the data in fairly high-level terms, and then refine this description by considering additional storage and representation details as needed.

In Section 1.6 we consider how users can retrieve data stored in a DBMS and the need for techniques to efficiently compute answers to questions involving such data. In Section 1.7 we provide an overview of how a DBMS supports concurrent access to data by several users, and how it protects the data in the event of system failures.

We then briefly describe the internal structure of a DBMS in Section 1.8, and mention various groups of people associated with the development and use of a DBMS in Section 1.9.

1.2 A HISTORICAL PERSPECTIVE

From the earliest days of computers, storing and manipulating data have been a major application focus. The first general-purpose DBMS was designed by Charles Bachman at General Electric in the early 1960s and was called the Integrated Data Store. It formed the basis for the *network data model*, which was standardized by the Conference on Data Systems Languages (CODASYL) and strongly influenced database systems through the 1960s. Bachman was the first recipient of ACM's Turing Award (the computer science equivalent of a Nobel prize) for work in the database area; he received the award in 1973.

In the late 1960s, IBM developed the Information Management System (IMS) DBMS, used even today in many major installations. IMS formed the basis for an alternative data representation framework called the *hierarchical data model*. The SABRE system for making airline reservations was jointly developed by American Airlines and IBM around the same time, and it allowed several people to access the same data through

a computer network. Interestingly, today the same SABRE system is used to power popular Web-based travel services such as Travelocity!

In 1970, Edgar Codd, at IBM's San Jose Research Laboratory, proposed a new data representation framework called the *relational data model*. This proved to be a watershed in the development of database systems: it sparked rapid development of several DBMSs based on the relational model, along with a rich body of theoretical results that placed the field on a firm foundation. Codd won the 1981 Turing Award for his seminal work. Database systems matured as an academic discipline, and the popularity of relational DBMSs changed the commercial landscape. Their benefits were widely recognized, and the use of DBMSs for managing corporate data became standard practice.

In the 1980s, the relational model consolidated its position as the dominant DBMS paradigm, and database systems continued to gain widespread use. The SQL query language for relational databases, developed as part of IBM's System R project, is now the standard query language. SQL was standardized in the late 1980s, and the current standard, SQL-92, was adopted by the American National Standards Institute (ANSI) and International Standards Organization (ISO). Arguably, the most widely used form of concurrent programming is the concurrent execution of database programs (called *transactions*). Users write programs as if they are to be run by themselves, and the responsibility for running them concurrently is given to the DBMS. James Gray won the 1999 Turing award for his contributions to the field of transaction management in a DBMS.

In the late 1980s and the 1990s, advances have been made in many areas of database systems. Considerable research has been carried out into more powerful query languages and richer data models, and there has been a big emphasis on supporting complex analysis of data from all parts of an enterprise. Several vendors (e.g., IBM's DB2, Oracle 8, Informix UDS) have extended their systems with the ability to store new data types such as images and text, and with the ability to ask more complex queries. Specialized systems have been developed by numerous vendors for creating *data warehouses*, consolidating data from several databases, and for carrying out specialized analysis.

An interesting phenomenon is the emergence of several *enterprise resource planning (ERP)* and *management resource planning (MRP)* packages, which add a substantial layer of application-oriented features on top of a DBMS. Widely used packages include systems from Baan, Oracle, PeopleSoft, SAP, and Siebel. These packages identify a set of common tasks (e.g., inventory management, human resources planning, financial analysis) encountered by a large number of organizations and provide a general application layer to carry out these tasks. The data is stored in a relational DBMS, and the application layer can be customized to different companies, leading to lower

overall costs for the companies, compared to the cost of building the application layer from scratch.

Most significantly, perhaps, DBMSs have entered the Internet Age. While the first generation of Web sites stored their data exclusively in operating systems files, the use of a DBMS to store data that is accessed through a Web browser is becoming widespread. Queries are generated through Web-accessible forms and answers are formatted using a markup language such as HTML, in order to be easily displayed in a browser. All the database vendors are adding features to their DBMS aimed at making it more suitable for deployment over the Internet.

Database management continues to gain importance as more and more data is brought on-line, and made ever more accessible through computer networking. Today the field is being driven by exciting visions such as multimedia databases, interactive video, digital libraries, a host of scientific projects such as the human genome mapping effort and NASA's Earth Observation System project, and the desire of companies to consolidate their decision-making processes and *mine* their data repositories for useful information about their businesses. Commercially, database management systems represent one of the largest and most vigorous market segments. Thus the study of database systems could prove to be richly rewarding in more ways than one!

1.3 FILE SYSTEMS VERSUS A DBMS

To understand the need for a DBMS, let us consider a motivating scenario: A company has a large collection (say, 500 GB[1]) of data on employees, departments, products, sales, and so on. This data is accessed concurrently by several employees. Questions about the data must be answered quickly, changes made to the data by different users must be applied consistently, and access to certain parts of the data (e.g., salaries) must be restricted.

We can try to deal with this data management problem by storing the data in a collection of operating system files. This approach has many drawbacks, including the following:

- We probably do not have 500 GB of main memory to hold all the data. We must therefore store data in a storage device such as a disk or tape and bring relevant parts into main memory for processing as needed.
- Even if we have 500 GB of main memory, on computer systems with 32-bit addressing, we cannot refer directly to more than about 4 GB of data! We have to program some method of identifying all data items.

[1] A kilobyte (KB) is 1024 bytes, a megabyte (MB) is 1024 KBs, a gigabyte (GB) is 1024 MBs, a terabyte (TB) is 1024 GBs, and a petabyte (PB) is 1024 terabytes.

- We have to write special programs to answer each question that users may want to ask about the data. These programs are likely to be complex because of the large volume of data to be searched.
- We must protect the data from inconsistent changes made by different users accessing the data concurrently. If programs that access the data are written with such concurrent access in mind, this adds greatly to their complexity.
- We must ensure that data is restored to a consistent state if the system crashes while changes are being made.
- Operating systems provide only a password mechanism for security. This is not sufficiently flexible to enforce security policies in which different users have permission to access different subsets of the data.

A DBMS is a piece of software that is designed to make the preceding tasks easier. By storing data in a DBMS, rather than as a collection of operating system files, we can use the DBMS's features to manage the data in a robust and efficient manner. As the volume of data and the number of users grow—hundreds of gigabytes of data and thousands of users are common in current corporate databases—DBMS support becomes indispensable.

1.4 ADVANTAGES OF A DBMS

Using a DBMS to manage data has many advantages:

- **Data independence:** Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.
- **Efficient data access:** A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.
- **Data integrity and security:** If data is always accessed through the DBMS, the DBMS can enforce integrity constraints on the data. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, the DBMS can enforce *access controls* that govern what data is visible to different classes of users.
- **Data administration:** When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals who understand the nature of the data being managed, and how different groups of users use it, can be responsible for organizing the data representation to minimize redundancy and for fine-tuning the storage of the data to make retrieval efficient.

- **Concurrent access and crash recovery:** A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.
- **Reduced application development time:** Clearly, the DBMS supports many important functions that are common to many applications accessing data stored in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick development of applications. Such applications are also likely to be more robust than applications developed from scratch because many important tasks are handled by the DBMS instead of being implemented by the application.

Given all these advantages, is there ever a reason *not* to use a DBMS? A DBMS is a complex piece of software, optimized for certain kinds of workloads (e.g., answering complex queries or handling many concurrent requests), and its performance may not be adequate for certain specialized applications. Examples include applications with tight real-time constraints or applications with just a few well-defined critical operations for which efficient custom code must be written. Another reason for not using a DBMS is that an application may need to manipulate the data in ways not supported by the query language. In such a situation, the abstract view of the data presented by the DBMS does not match the application's needs, and actually gets in the way. As an example, relational databases do not support flexible analysis of text data (although vendors are now extending their products in this direction). If specialized performance or data manipulation requirements are central to an application, the application may choose not to use a DBMS, especially if the added benefits of a DBMS (e.g., flexible querying, security, concurrent access, and crash recovery) are not required. In most situations calling for large-scale data management, however, DBMSs have become an indispensable tool.

1.5 DESCRIBING AND STORING DATA IN A DBMS

The user of a DBMS is ultimately concerned with some real-world enterprise, and the data to be stored describes various aspects of this enterprise. For example, there are students, faculty, and courses in a university, and the data in a university database describes these entities and their relationships.

A **data model** is a collection of high-level data description constructs that hide many low-level storage details. A DBMS allows a user to define the data to be stored in terms of a data model. Most database management systems today are based on the **relational data model**, which we will focus on in this book.

While the data model of the DBMS hides many details, it is nonetheless closer to how the DBMS stores data than to how a user thinks about the underlying enterprise. A **semantic data model** is a more abstract, high-level data model that makes it easier

for a user to come up with a good initial description of the data in an enterprise. These models contain a wide variety of constructs that help describe a real application scenario. A DBMS is not intended to support all these constructs directly; it is typically built around a data model with just a few basic constructs, such as the relational model. A database design in terms of a semantic model serves as a useful starting point and is subsequently translated into a database design in terms of the data model the DBMS actually supports.

A widely used semantic data model called the entity-relationship (ER) model allows us to pictorially denote entities and the relationships among them. We cover the ER model in Chapter 2.

1.5.1 The Relational Model

In this section we provide a brief introduction to the relational model. The central data description construct in this model is a **relation**, which can be thought of as a set of **records**.

A description of data in terms of a data model is called a **schema**. In the relational model, the schema for a relation specifies its name, the name of each **field** (or **attribute** or **column**), and the type of each field. As an example, student information in a university database may be stored in a relation with the following schema:

Students(*sid:* `string`, *name:* `string`, *login:* `string`, *age:* `integer`, *gpa:* `real`)

The preceding schema says that each record in the Students relation has five fields, with field names and types as indicated.[2] An example instance of the Students relation appears in Figure 1.1.

sid	*name*	*login*	*age*	*gpa*
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 1.1 An Instance of the Students Relation

[2]Storing *date of birth* is preferable to storing *age*, since it does not change over time, unlike age. We've used *age* for simplicity in our discussion.

Each row in the Students relation is a record that describes a student. The description is not complete—for example, the student's height is not included—but is presumably adequate for the intended applications in the university database. Every row follows the schema of the Students relation. The schema can therefore be regarded as a template for describing a student.

We can make the description of a collection of students more precise by specifying **integrity constraints**, which are conditions that the records in a relation must satisfy. For example, we could specify that every student has a unique *sid* value. Observe that we cannot capture this information by simply adding another field to the Students schema. Thus, the ability to specify uniqueness of the values in a field increases the accuracy with which we can describe our data. The expressiveness of the constructs available for specifying integrity constraints is an important aspect of a data model.

Other Data Models

In addition to the relational data model (which is used in numerous systems, including IBM's DB2, Informix, Oracle, Sybase, Microsoft's Access, FoxBase, Paradox, Tandem, and Teradata), other important data models include the hierarchical model (e.g., used in IBM's IMS DBMS), the network model (e.g., used in IDS and IDMS), the object-oriented model (e.g., used in Objectstore and Versant), and the object-relational model (e.g., used in DBMS products from IBM, Informix, ObjectStore, Oracle, Versant, and others). While there are many databases that use the hierarchical and network models, and systems based on the object-oriented and object-relational models are gaining acceptance in the marketplace, the dominant model today is the relational model.

In this book, we will focus on the relational model because of its wide use and importance. Indeed, the object-relational model, which is gaining in popularity, is an effort to combine the best features of the relational and object-oriented models, and a good grasp of the relational model is necessary to understand object-relational concepts. (We discuss the object-oriented and object-relational models in Chapter 25.)

1.5.2 Levels of Abstraction in a DBMS

The data in a DBMS is described at three levels of abstraction, as illustrated in Figure 1.2. The database description consists of a schema at each of these three levels of abstraction: the *conceptual*, *physical*, and *external* schemas.

A **data definition language** (DDL) is used to define the external and conceptual schemas. We will discuss the DDL facilities of the most widely used database language, SQL, in Chapter 3. All DBMS vendors also support SQL commands to describe aspects of the physical schema, but these commands are not part of the SQL-92 language

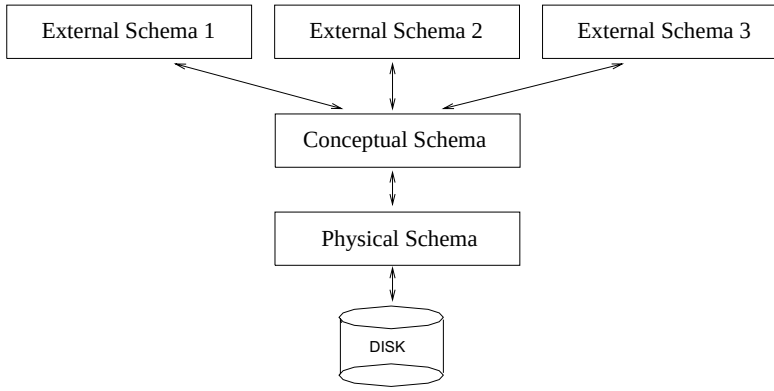

Figure 1.2 Levels of Abstraction in a DBMS

standard. Information about the conceptual, external, and physical schemas is stored in the **system catalogs** (Section 13.2). We discuss the three levels of abstraction in the rest of this section.

Conceptual Schema

The **conceptual schema** (sometimes called the **logical schema**) describes the stored data in terms of the data model of the DBMS. In a relational DBMS, the conceptual schema describes all relations that are stored in the database. In our sample university database, these relations contain information about *entities*, such as students and faculty, and about *relationships*, such as students' enrollment in courses. All student entities can be described using records in a Students relation, as we saw earlier. In fact, each collection of entities and each collection of relationships can be described as a relation, leading to the following conceptual schema:

Students(*sid:* `string`, *name:* `string`, *login:* `string`,
age: `integer`, *gpa:* `real`)
Faculty(*fid:* `string`, *fname:* `string`, *sal:* `real`)
Courses(*cid:* `string`, *cname:* `string`, *credits:* `integer`)
Rooms(*rno:* `integer`, *address:* `string`, *capacity:* `integer`)
Enrolled(*sid:* `string`, *cid:* `string`, *grade:* `string`)
Teaches(*fid:* `string`, *cid:* `string`)
Meets_In(*cid:* `string`, *rno:* `integer`, *time:* `string`)

The choice of relations, and the choice of fields for each relation, is not always obvious, and the process of arriving at a good conceptual schema is called **conceptual database design**. We discuss conceptual database design in Chapters 2 and 15.

Physical Schema

The **physical schema** specifies additional storage details. Essentially, the physical schema summarizes how the relations described in the conceptual schema are actually stored on secondary storage devices such as disks and tapes.

We must decide what file organizations to use to store the relations, and create auxiliary data structures called **indexes** to speed up data retrieval operations. A sample physical schema for the university database follows:

- Store all relations as unsorted files of records. (A file in a DBMS is either a collection of records or a collection of pages, rather than a string of characters as in an operating system.)
- Create indexes on the first column of the Students, Faculty, and Courses relations, the *sal* column of Faculty, and the *capacity* column of Rooms.

Decisions about the physical schema are based on an understanding of how the data is typically accessed. The process of arriving at a good physical schema is called **physical database design**. We discuss physical database design in Chapter 16.

External Schema

External schemas, which usually are also in terms of the data model of the DBMS, allow data access to be customized (and authorized) at the level of individual users or groups of users. Any given database has exactly one conceptual schema and one physical schema because it has just one set of stored relations, but it may have several external schemas, each tailored to a particular group of users. Each external schema consists of a collection of one or more **views** and relations from the conceptual schema. A view is conceptually a relation, but the records in a view are not stored in the DBMS. Rather, they are computed using a definition for the view, in terms of relations stored in the DBMS. We discuss views in more detail in Chapter 3.

The external schema design is guided by end user requirements. For example, we might want to allow students to find out the names of faculty members teaching courses, as well as course enrollments. This can be done by defining the following view:

Courseinfo(*cid:* `string`, *fname:* `string`, *enrollment:* `integer`)

A user can treat a view just like a relation and ask questions about the records in the view. Even though the records in the view are not stored explicitly, they are computed as needed. We did not include Courseinfo in the conceptual schema because we can compute Courseinfo from the relations in the conceptual schema, and to store it in addition would be redundant. Such redundancy, in addition to the wasted space, could

lead to inconsistencies. For example, a tuple may be inserted into the Enrolled relation, indicating that a particular student has enrolled in some course, without incrementing the value in the *enrollment* field of the corresponding record of Courseinfo (if the latter also is part of the conceptual schema and its tuples are stored in the DBMS).

1.5.3 Data Independence

A very important advantage of using a DBMS is that it offers **data independence**. That is, application programs are insulated from changes in the way the data is structured and stored. Data independence is achieved through use of the three levels of data abstraction; in particular, the conceptual schema and the external schema provide distinct benefits in this area.

Relations in the external schema (view relations) are in principle generated on demand from the relations corresponding to the conceptual schema.[3] If the underlying data is reorganized, that is, the conceptual schema is changed, the definition of a view relation can be modified so that the same relation is computed as before. For example, suppose that the Faculty relation in our university database is replaced by the following two relations:

Faculty_public(*fid:* `string`, *fname:* `string`, *office:* `integer`)
Faculty_private(*fid:* `string`, *sal:* `real`)

Intuitively, some confidential information about faculty has been placed in a separate relation and information about offices has been added. The Courseinfo view relation can be redefined in terms of Faculty_public and Faculty_private, which together contain all the information in Faculty, so that a user who queries Courseinfo will get the same answers as before.

Thus users can be shielded from changes in the logical structure of the data, or changes in the choice of relations to be stored. This property is called **logical data independence**.

In turn, the conceptual schema insulates users from changes in the physical storage of the data. This property is referred to as **physical data independence**. The conceptual schema hides details such as how the data is actually laid out on disk, the file structure, and the choice of indexes. As long as the conceptual schema remains the same, we can change these storage details without altering applications. (Of course, performance might be affected by such changes.)

[3] In practice, they could be precomputed and stored to speed up queries on view relations, but the computed view relations must be updated whenever the underlying relations are updated.

1.6 QUERIES IN A DBMS

The ease with which information can be obtained from a database often determines its value to a user. In contrast to older database systems, relational database systems allow a rich class of questions to be posed easily; this feature has contributed greatly to their popularity. Consider the sample university database in Section 1.5.2. Here are examples of questions that a user might ask:

1. What is the name of the student with student id 123456?
2. What is the average salary of professors who teach the course with cid CS564?
3. How many students are enrolled in course CS564?
4. What fraction of students in course CS564 received a grade better than B?
5. Is any student with a GPA less than 3.0 enrolled in course CS564?

Such questions involving the data stored in a DBMS are called **queries**. A DBMS provides a specialized language, called the **query language**, in which queries can be posed. A very attractive feature of the relational model is that it supports powerful query languages. **Relational calculus** is a formal query language based on mathematical logic, and queries in this language have an intuitive, precise meaning. **Relational algebra** is another formal query language, based on a collection of **operators** for manipulating relations, which is equivalent in power to the calculus.

A DBMS takes great care to evaluate queries as efficiently as possible. We discuss query optimization and evaluation in Chapters 12 and 13. Of course, the efficiency of query evaluation is determined to a large extent by how the data is stored physically. Indexes can be used to speed up many queries—in fact, a good choice of indexes for the underlying relations can speed up each query in the preceding list. We discuss data storage and indexing in Chapters 7, 8, 9, and 10.

A DBMS enables users to create, modify, and query data through a **data manipulation language** (DML). Thus, the query language is only one part of the DML, which also provides constructs to insert, delete, and modify data. We will discuss the DML features of SQL in Chapter 5. The DML and DDL are collectively referred to as the **data sublanguage** when embedded within a **host language** (e.g., C or COBOL).

1.7 TRANSACTION MANAGEMENT

Consider a database that holds information about airline reservations. At any given instant, it is possible (and likely) that several travel agents are looking up information about available seats on various flights and making new seat reservations. When several users access (and possibly modify) a database concurrently, the DBMS must order

their requests carefully to avoid conflicts. For example, when one travel agent looks up Flight 100 on some given day and finds an empty seat, another travel agent may simultaneously be making a reservation for that seat, thereby making the information seen by the first agent obsolete.

Another example of concurrent use is a bank's database. While one user's application program is computing the total deposits, another application may transfer money from an account that the first application has just 'seen' to an account that has not yet been seen, thereby causing the total to appear larger than it should be. Clearly, such anomalies should not be allowed to occur. However, disallowing concurrent access can degrade performance.

Further, the DBMS must protect users from the effects of system failures by ensuring that all data (and the status of active applications) is restored to a consistent state when the system is restarted after a crash. For example, if a travel agent asks for a reservation to be made, and the DBMS responds saying that the reservation has been made, the reservation should not be lost if the system crashes. On the other hand, if the DBMS has not yet responded to the request, but is in the process of making the necessary changes to the data while the crash occurs, the partial changes should be undone when the system comes back up.

A **transaction** is *any one execution* of a user program in a DBMS. (Executing the same program several times will generate several transactions.) This is the basic unit of change as seen by the DBMS: Partial transactions are not allowed, and the effect of a group of transactions is equivalent to some serial execution of all transactions. We briefly outline how these properties are guaranteed, deferring a detailed discussion to later chapters.

1.7.1 Concurrent Execution of Transactions

An important task of a DBMS is to schedule concurrent accesses to data so that each user can safely ignore the fact that others are accessing the data concurrently. The importance of this task cannot be underestimated because a database is typically shared by a large number of users, who submit their requests to the DBMS independently, and simply cannot be expected to deal with arbitrary changes being made concurrently by other users. A DBMS allows users to think of their programs as if they were executing in isolation, one after the other in some order chosen by the DBMS. For example, if a program that deposits cash into an account is submitted to the DBMS at the same time as another program that debits money from the same account, either of these programs could be run first by the DBMS, but their steps will not be interleaved in such a way that they interfere with each other.

A **locking protocol** is a set of rules to be followed by each transaction (and enforced by the DBMS), in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order. A **lock** is a mechanism used to control access to database objects. Two kinds of **locks** are commonly supported by a DBMS: **shared locks** on an object can be held by two different transactions at the same time, but an **exclusive lock** on an object ensures that no other transactions hold *any* lock on this object.

Suppose that the following locking protocol is followed: Every transaction begins by obtaining a shared lock on each data object that it needs to read and an exclusive lock on each data object that it needs to modify, and then releases all its locks after completing all actions. Consider two transactions $T1$ and $T2$ such that $T1$ wants to modify a data object and $T2$ wants to read the same object. Intuitively, if $T1$'s request for an exclusive lock on the object is granted first, $T2$ cannot proceed until $T1$ releases this lock, because $T2$'s request for a shared lock will not be granted by the DBMS until then. Thus, all of $T1$'s actions will be completed before any of $T2$'s actions are initiated. We consider locking in more detail in Chapters 18 and 19.

1.7.2 Incomplete Transactions and System Crashes

Transactions can be interrupted before running to completion for a variety of reasons, e.g., a system crash. A DBMS must ensure that the changes made by such incomplete transactions are removed from the database. For example, if the DBMS is in the middle of transferring money from account A to account B, and has debited the first account but not yet credited the second when the crash occurs, the money debited from account A must be restored when the system comes back up after the crash.

To do so, the DBMS maintains a **log** of all writes to the database. A crucial property of the log is that each write action must be recorded in the log (on disk) *before* the corresponding change is reflected in the database itself—otherwise, if the system crashes just after making the change in the database but before the change is recorded in the log, the DBMS would be unable to detect and undo this change. This property is called **Write-Ahead Log** or **WAL**. To ensure this property, the DBMS must be able to selectively force a page in memory to disk.

The log is also used to ensure that the changes made by a successfully completed transaction are not lost due to a system crash, as explained in Chapter 20. Bringing the database to a consistent state after a system crash can be a slow process, since the DBMS must ensure that the effects of all transactions that completed prior to the crash are restored, and that the effects of incomplete transactions are undone. The time required to recover from a crash can be reduced by periodically forcing some information to disk; this periodic operation is called a **checkpoint**.

1.7.3 Points to Note

In summary, there are three points to remember with respect to DBMS support for concurrency control and recovery:

1. Every object that is read or written by a transaction is first locked in shared or exclusive mode, respectively. Placing a lock on an object restricts its availability to other transactions and thereby affects performance.
2. For efficient log maintenance, the DBMS must be able to selectively force a collection of pages in main memory to disk. Operating system support for this operation is not always satisfactory.
3. Periodic checkpointing can reduce the time needed to recover from a crash. Of course, this must be balanced against the fact that checkpointing too often slows down normal execution.

1.8 STRUCTURE OF A DBMS

Figure 1.3 shows the structure (with some simplification) of a typical DBMS based on the relational data model.

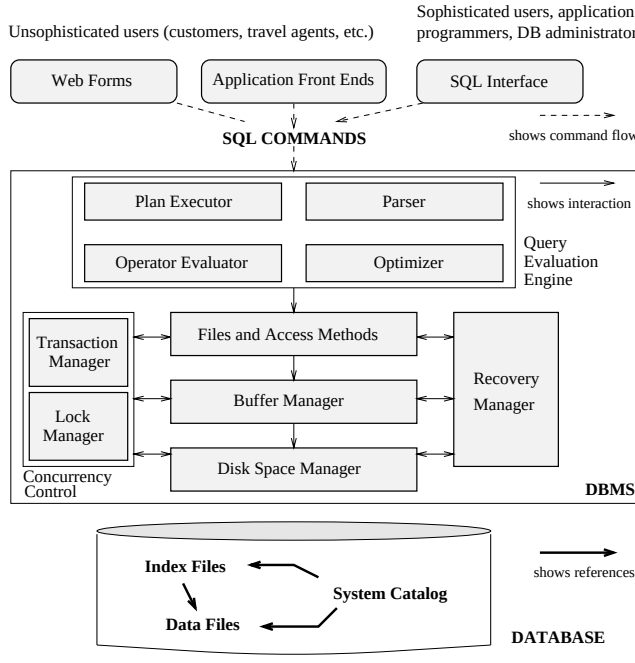

Figure 1.3 Architecture of a DBMS

The DBMS accepts SQL commands generated from a variety of user interfaces, produces query evaluation plans, executes these plans against the database, and returns the answers. (This is a simplification: SQL commands can be embedded in host-language application programs, e.g., Java or COBOL programs. We ignore these issues to concentrate on the core DBMS functionality.)

When a user issues a query, the parsed query is presented to a **query optimizer**, which uses information about how the data is stored to produce an efficient execution plan for evaluating the query. An **execution plan** is a blueprint for evaluating a query, and is usually represented as a tree of relational operators (with annotations that contain additional detailed information about which access methods to use, etc.). We discuss query optimization in Chapter 13. Relational operators serve as the building blocks for evaluating queries posed against the data. The implementation of these operators is discussed in Chapter 12.

The code that implements relational operators sits on top of the file and access methods layer. This layer includes a variety of software for supporting the concept of a **file**, which, in a DBMS, is a collection of pages or a collection of records. This layer typically supports a **heap file**, or file of unordered pages, as well as indexes. In addition to keeping track of the pages in a file, this layer organizes the information within a page. File and page level storage issues are considered in Chapter 7. File organizations and indexes are considered in Chapter 8.

The files and access methods layer code sits on top of the **buffer manager**, which brings pages in from disk to main memory as needed in response to read requests. Buffer management is discussed in Chapter 7.

The lowest layer of the DBMS software deals with management of space on disk, where the data is stored. Higher layers allocate, deallocate, read, and write pages through (routines provided by) this layer, called the **disk space manager**. This layer is discussed in Chapter 7.

The DBMS supports concurrency and crash recovery by carefully scheduling user requests and maintaining a log of all changes to the database. DBMS components associated with concurrency control and recovery include the **transaction manager**, which ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transactions; the **lock manager**, which keeps track of requests for locks and grants locks on database objects when they become available; and the **recovery manager**, which is responsible for maintaining a log, and restoring the system to a consistent state after a crash. The disk space manager, buffer manager, and file and access method layers must interact with these components. We discuss concurrency control and recovery in detail in Chapter 18.

1.9 PEOPLE WHO DEAL WITH DATABASES

Quite a variety of people are associated with the creation and use of databases. Obviously, there are **database implementors**, who build DBMS software, and **end users** who wish to store and use data in a DBMS. Database implementors work for vendors such as IBM or Oracle. End users come from a diverse and increasing number of fields. As data grows in complexity and volume, and is increasingly recognized as a major asset, the importance of maintaining it professionally in a DBMS is being widely accepted. Many end users simply use applications written by database application programmers (see below), and so require little technical knowledge about DBMS software. Of course, sophisticated users who make more extensive use of a DBMS, such as writing their own queries, require a deeper understanding of its features.

In addition to end users and implementors, two other classes of people are associated with a DBMS: *application programmers* and *database administrators* (DBAs).

Database application programmers develop packages that facilitate data access for end users, who are usually not computer professionals, using the host or data languages and software tools that DBMS vendors provide. (Such tools include report writers, spreadsheets, statistical packages, etc.) Application programs should ideally access data through the external schema. It is possible to write applications that access data at a lower level, but such applications would compromise data independence.

A personal database is typically maintained by the individual who owns it and uses it. However, corporate or enterprise-wide databases are typically important enough and complex enough that the task of designing and maintaining the database is entrusted to a professional called the **database administrator**. The DBA is responsible for many critical tasks:

- **Design of the conceptual and physical schemas:** The DBA is responsible for interacting with the users of the system to understand what data is to be stored in the DBMS and how it is likely to be used. Based on this knowledge, the DBA must design the conceptual schema (decide what relations to store) and the physical schema (decide how to store them). The DBA may also design widely used portions of the external schema, although users will probably augment this schema by creating additional views.

- **Security and authorization:** The DBA is responsible for ensuring that unauthorized data access is not permitted. In general, not everyone should be able to access all the data. In a relational DBMS, users can be granted permission to access only certain views and relations. For example, although you might allow students to find out course enrollments and who teaches a given course, you would not want students to see faculty salaries or each others' grade information.

The DBA can enforce this policy by giving students permission to read only the Courseinfo view.

- **Data availability and recovery from failures:** The DBA must take steps to ensure that if the system fails, users can continue to access as much of the uncorrupted data as possible. The DBA must also work to restore the data to a consistent state. The DBMS provides software support for these functions, but the DBA is responsible for implementing procedures to back up the data periodically and to maintain logs of system activity (to facilitate recovery from a crash).
- **Database tuning:** The needs of users are likely to evolve with time. The DBA is responsible for modifying the database, in particular the conceptual and physical schemas, to ensure adequate performance as user requirements change.

1.10 POINTS TO REVIEW

- A database management system (DBMS) is software that supports management of large collections of data. A DBMS provides efficient data access, data independence, data integrity, security, quick application development, support for concurrent access, and recovery from system failures. **(Section 1.1)**
- Storing data in a DBMS versus storing it in operating system files has many advantages. **(Section 1.3)**
- Using a DBMS provides the user with data independence, efficient data access, automatic data integrity, and security. **(Section 1.4)**
- The structure of the data is described in terms of a *data model* and the description is called a *schema*. The *relational model* is currently the most popular data model. A DBMS distinguishes between *external*, *conceptual*, and *physical schema* and thus allows a view of the data at three levels of abstraction. Physical and logical *data independence*, which are made possible by these three levels of abstraction, insulate the users of a DBMS from the way the data is structured and stored inside a DBMS. **(Section 1.5)**
- A *query language* and a *data manipulation language* enable high-level access and modification of the data. **(Section 1.6)**
- A *transaction* is a logical unit of access to a DBMS. The DBMS ensures that either all or none of a transaction's changes are applied to the database. For performance reasons, the DBMS processes multiple transactions concurrently, but ensures that the result is equivalent to running the transactions one after the other in some order. The DBMS maintains a record of all changes to the data in the *system log*, in order to undo partial transactions and recover from system crashes. *Checkpointing* is a periodic operation that can reduce the time for recovery from a crash. **(Section 1.7)**

- DBMS code is organized into several modules: the disk space manager, the buffer manager, a layer that supports the abstractions of files and index structures, a layer that implements relational operators, and a layer that optimizes queries and produces an execution plan in terms of relational operators. **(Section 1.8)**
- A *database administrator (DBA)* manages a DBMS for an enterprise. The DBA designs schemas, provide security, restores the system after a failure, and periodically tunes the database to meet changing user needs. Application programmers develop applications that use DBMS functionality to access and manipulate data, and end users invoke these applications. **(Section 1.9)**

EXERCISES

Exercise 1.1 Why would you choose a database system instead of simply storing data in operating system files? When would it make sense *not* to use a database system?

Exercise 1.2 What is logical data independence and why is it important?

Exercise 1.3 Explain the difference between logical and physical data independence.

Exercise 1.4 Explain the difference between external, internal, and conceptual schemas. How are these different schema layers related to the concepts of logical and physical data independence?

Exercise 1.5 What are the responsibilities of a DBA? If we assume that the DBA is never interested in running his or her own queries, does the DBA still need to understand query optimization? Why?

Exercise 1.6 Scrooge McNugget wants to store information (names, addresses, descriptions of embarrassing moments, etc.) about the many ducks on his payroll. Not surprisingly, the volume of data compels him to buy a database system. To save money, he wants to buy one with the fewest possible features, and he plans to run it as a stand-alone application on his PC clone. Of course, Scrooge does not plan to share his list with anyone. Indicate which of the following DBMS features Scrooge should pay for; in each case also indicate why Scrooge should (or should not) pay for that feature in the system he buys.

1. A security facility.
2. Concurrency control.
3. Crash recovery.
4. A view mechanism.
5. A query language.

Exercise 1.7 Which of the following plays an important role in *representing* information about the real world in a database? Explain briefly.

1. The data definition language.

2. The data manipulation language.
3. The buffer manager.
4. The data model.

Exercise 1.8 Describe the structure of a DBMS. If your operating system is upgraded to support some new functions on OS files (e.g., the ability to force some sequence of bytes to disk), which layer(s) of the DBMS would you have to rewrite in order to take advantage of these new functions?

Exercise 1.9 Answer the following questions:

1. What is a transaction?
2. Why does a DBMS interleave the actions of different transactions, instead of executing transactions one after the other?
3. What must a user guarantee with respect to a transaction and database consistency? What should a DBMS guarantee with respect to concurrent execution of several transactions and database consistency?
4. Explain the strict two-phase locking protocol.
5. What is the WAL property, and why is it important?

PROJECT-BASED EXERCISES

Exercise 1.10 Use a Web browser to look at the HTML documentation for Minibase. Try to get a feel for the overall architecture.

BIBLIOGRAPHIC NOTES

The evolution of database management systems is traced in [248]. The use of data models for describing real-world data is discussed in [361], and [363] contains a taxonomy of data models. The three levels of abstraction were introduced in [155, 623]. The network data model is described in [155], and [680] discusses several commercial systems based on this model. [634] contains a good annotated collection of systems-oriented papers on database management.

Other texts covering database management systems include [169, 208, 289, 600, 499, 656, 669]. [169] provides a detailed discussion of the relational model from a conceptual standpoint and is notable for its extensive annotated bibliography. [499] presents a performance-oriented perspective, with references to several commercial systems. [208] and [600] offer broad coverage of database system concepts, including a discussion of the hierarchical and network data models. [289] emphasizes the connection between database query languages and logic programming. [669] emphasizes data models. Of these texts, [656] provides the most detailed discussion of theoretical issues. Texts devoted to theoretical aspects include [38, 436, 3]. Handbook [653] includes a section on databases that contains introductory survey articles on a number of topics.