CHAPTER 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 techniques form the focus of this book. This chapter briefly introduces the principles of database systems.

1.1 Database-System Applications

The earliest database systems arose in the 1960s in response to the computerized management of commercial data. Those earlier applications were relatively simple compared to modern database applications. Modern applications include highly sophisticated, worldwide enterprises.

All database applications, old and new, share important common elements. The central aspect of the application is not a program performing some calculation, but rather the data themselves. Today, some of the most valuable corporations are valuable not because of their physical assets, but rather because of the information they own. Imagine a bank without its data on accounts and customers or a social-network site that loses the connections among its users. Such companies' value would be almost totally lost under such circumstances.

Database systems are used to manage collections of data that:

- are highly valuable,
- are relatively large, and
- are accessed by multiple users and applications, often at the same time.

The first database applications had only simple, precisely formatted, structured data. Today, database applications may include data with complex relationships and a more variable structure. As an example of an application with structured data, consider a university's records regarding courses, students, and course registration. The university keeps the same type of information about each course: course-identifier, title, department, course number, etc., and similarly for students: student-identifier, name, address, phone, etc. Course registration is a collection of pairs: one course identifier and one student identifier. Information of this sort has a standard, repeating structure and is representative of the type of database applications that go back to the 1960s. Contrast this simple university database application with a social-networking site. Users of the site post varying types of information about themselves ranging from simple items such as name or date of birth, to complex posts consisting of text, images, videos, and links to other users. There is only a limited amount of common structure among these data. Both of these applications, however, share the basic features of a database.

Modern database systems exploit commonalities in the structure of data to gain efficiency but also allow for weakly structured data and for data whose formats are highly variable. As a result, a database system is a large, complex software system whose task is to manage a large, complex collection of data.

Managing complexity is challenging, not only in the management of data but in any domain. Key to the management of complexity is the concept of *abstraction*. Abstraction allows a person to use a complex device or system without having to know the details of how that device or system is constructed. A person is able, for example, to drive a car by knowing how to operate its controls. However, the driver does not need to know how the motor was built nor how it operates. All the driver needs to know is an abstraction of what the motor does. Similarly, for a large, complex collection of data, a database system provides a simpler, abstract view of the information so that users and application programmers do not need to be aware of the underlying details of how data are stored and organized. By providing a high level of abstraction, a database system makes it possible for an enterprise to combine data of various types into a unified repository of the information needed to run the enterprise.

Here are some representative applications:

- **Enterprise Information**
  - **Sales:** For customer, product, and purchase information.

- **Accounting:** For payments, receipts, account balances, assets, and other accounting information.
- **Human resources:** For information about employees, salaries, payroll taxes, and benefits, and for generation of paychecks.

- **Manufacturing:** For management of the supply chain and for tracking production of items in factories, inventories of items in warehouses and stores, and orders for items.
- **Banking and Finance**
  - **Banking:** For customer information, accounts, loans, and banking transactions.
  - **Credit card transactions:** For purchases on credit cards and generation of monthly statements.
  - **Finance:** For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds; also for storing real-time market data to enable online trading by customers and automated trading by the firm.
- **Universities:** For student information, course registrations, and grades (in addition to standard enterprise information such as human resources and accounting).
- **Airlines:** For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner.
- **Telecommunication:** For keeping records of calls, texts, and data usage, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.
- **Web-based services**
  - **Social-media:** For keeping records of users, connections between users (such as friend/follows information), posts made by users, rating/like information about posts, etc.
  - **Online retailers:** For keeping records of sales data and orders as for any retailer, but also for tracking a user's product views, search terms, etc., for the purpose of identifying the best items to recommend to that user.
  - **Online advertisements:** For keeping records of click history to enable targeted advertisements, product suggestions, news articles, etc. People access such databases every time they do a web search, make an online purchase, or access a social-networking site.
- **Document databases:** For maintaining collections of new articles, patents, published research papers, etc.
- **Navigation systems:** For maintaining the locations of varies places of interest along with the exact routes of roads, train systems, buses, etc.

As this list illustrates, databases form an essential part not only of every enterprise but also of a large part of a person's daily activities.

The ways in which people interact with databases has changed over time. Early databases were maintained as back-office systems with which users interacted via printed reports and paper forms for input. As database systems became more sophisticated, better languages were developed for programmers to use in interacting with the data, along with user interfaces that allowed end users within the enterprise to query and update data.

As the support for programmer interaction with databases improved, and computer hardware performance increased even as hardware costs decreased, more sophisticated applications emerged that brought database data into more direct contact not only with end users within an enterprise but also with the general public. Whereas once bank customers had to interact with a teller for every transaction, automated-teller machines (ATMs) allowed direct customer interaction. Today, virtually every enterprise employs web applications or mobile applications to allow its customers to interact directly with the enterprise's database, and, thus, with the enterprise itself.

The user, or customer, can focus on the product or service without being aware of the details of the large database that makes the interaction possible. For instance, when you read a social-media post, or 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 you should see. Almost every interaction with a smartphone results in some sort of database access. 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.

Broadly speaking, there are two modes in which databases are used.

- The first mode is to support **online transaction processing**, where a large number of users use the database, with each user retrieving relatively small amounts of data, and performing small updates. This is the primary mode of use for the vast majority of users of database applications such as those that we outlined earlier.
- The second mode is to support **data analytics**, that is, the processing of data to draw conclusions, and infer rules or decision procedures, which are then used to drive business decisions.

  For example, banks need to decide whether to give a loan to a loan applicant, online advertisers need to decide which advertisement to show to a particular user. These tasks are addressed in two steps. First, data-analysis techniques attempt to automatically discover rules and patterns from data and create *predictive models*. These models take as input attributes ("features") of individuals, and output pre-

dictions such as likelihood of paying back a loan, or clicking on an advertisement, which are then used to make the business decision.

As another example, manufacturers and retailers need to make decisions on what items to manufacture or order in what quantities; these decisions are driven significantly by techniques for analyzing past data, and predicting trends. The cost of making wrong decisions can be very high, and organizations are therefore willing to invest a lot of money to gather or purchase required data, and build systems that can use the data to make accurate predictions.

The field of *data mining* combines knowledge-discovery techniques invented by artificial intelligence researchers and statistical analysts with efficient implementation techniques that enable them to be used on extremely large databases.

1.2 Purpose of Database Systems

To understand the purpose of database systems, consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. 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 programs to:

- Add new students, instructors, and courses.
- Register students for courses and generate class rosters.
- Assign grades to students, compute grade point averages (GPA), and generate transcripts.

Programmers develop these application programs to meet the needs of the university.

New application programs are added to the system as the need arises. For example, suppose that a university decides to create a new major. As a result, the university creates a new department and creates new permanent files (or adds information to existing files) to record information about all the instructors in the department, students in that major, course offerings, degree requirements, and so on. The university may have to write new application programs to deal with rules specific to the new major. New application programs may also have to be written to handle new rules in the university. 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.

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 structures, and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, if a student has a double major (say, music and mathematics), the address and telephone number of that student may appear in a file that consists of student records of students in the Music department and in a file that consists of student records of students in the Mathematics department. 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 student address may be reflected in the Music department records but not elsewhere in the system.

- **Difficulty in accessing data**. Suppose that one of the university clerks needs to find out the names of all students who live within a particular postal-code area. The clerk 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* students. The university clerk now has two choices: either obtain the list of all students and extract the needed information manually or ask a 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 clerk needs to trim that list to include only those students who have taken at least 60 credit hours. As expected, a program to generate such a list does not exist. Again, the clerk 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**. Suppose the university maintains an account for each department, and records the balance amount in each account. Suppose also that the university requires that the account balance of a department may never fall below zero. 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 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 banking system with a program to transfer $500 from account *A* to account *B*. If a system failure occurs during the execution of the program, it is possible that the $500 was removed from the balance of account *A* but was not credited to the balance of 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. Indeed, today, the largest internet retailers may have millions of accesses per day to their data by shoppers. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. Consider account *A*, with a balance of $10,000. If two bank clerks debit the account balance (by say $500 and $100, respectively) of account *A* at almost exactly the same time, the result of the concurrent executions may leave the account balance 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 $10,000, and write back $9500 and $9900, respectively. Depending on which one writes the value last, the balance of account *A* may contain either $9500 or $9900, rather than the correct value of $9400. 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.

  As another example, suppose a registration program maintains a count of students registered for a course in order to enforce limits on the number of students registered. When a student registers, the program reads the current count for the courses, verifies that the count is not already at the limit, adds one to the count, and stores the count back in the database. Suppose two students register concurrently, with the count at 39. The two program executions may both read the value 39, and both would then write back 40, leading to an incorrect increase of only 1, even though two students successfully registered for the course and the count should be 41. Furthermore, suppose the course registration limit was 40; in the above case both students would be able to register, leading to a violation of the limit of 40 students.

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

These difficulties, among others, prompted both the initial development of database systems and the transition of file-based applications to database systems, back in the 1960s and 1970s.

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 university organization as a running example of a typical data-processing application.

1.3 View of Data

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. 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 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.

There are a number of different data models that we shall cover in the text. The data models can be classified into four different categories:

- **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. Tables are also known as **relations**. 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. 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. Chapter 2 and Chapter 7 cover the relational model in detail.
- **Entity-Relationship Model**. The entity-relationship (E-R) data model uses a collection of basic objects, called *entities*, and *relationships* among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design. Chapter 6 explores it in detail.
- **Semi-structured Data Model**. The semi-structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. *JSON* and *Extensible Markup Language* (*XML*) are widely used semi-structured data representations. Semi-structured data models are explored in detail in Chapter 8.

- **Object-Based Data Model**. Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led initially to the development of a distinct object-oriented data model, but today the concept of objects is well integrated into relational databases. Standards exist to store objects in relational tables. Database systems allow procedures to be stored in the database system and executed by the database system. This can be seen as extending the relational model with notions of encapsulation, methods, and object identity. Object-based data models are summarized in Chapter 8.

A large portion of this text is focused on the relational model because it serves as the foundation for most database applications.

1.3.2 Relational Data Model

In the relational model, data are represented in the form of tables. Each table has multiple columns, and each column has a unique name. Each row of the table represents one piece of information. Figure 1.1 presents a sample relational database comprising two tables: one shows details of university instructors and the other shows details of the various university departments.

The first table, the *instructor* table, shows, for example, that an instructor named Einstein with *ID* 22222 is a member of the Physics department and has an annual salary of $95,000. The second table, *department*, shows, for example, that the Biology department is located in the Watson building and has a budget of $90,000. Of course, a real-world university would have many more departments and instructors. We use small tables in the text to illustrate concepts. A larger example for the same schema is available online.

1.3.3 Data Abstraction

For the system to be usable, it must retrieve data efficiently. The need for efficiency has led database system developers to use complex data structures to represent data in the database. Since many database-system users are not computer trained, developers hide the complexity from users through several levels of **data 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. This is referred to as **physical data indepen-**

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

(b) The *department* table

Figure 1.1 A sample relational database.

dence. 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 1.2 shows the relationship among the three levels of abstraction.

An important feature of data models, such as the relational model, is that they hide such low-level implementation details from not just database users, but even from

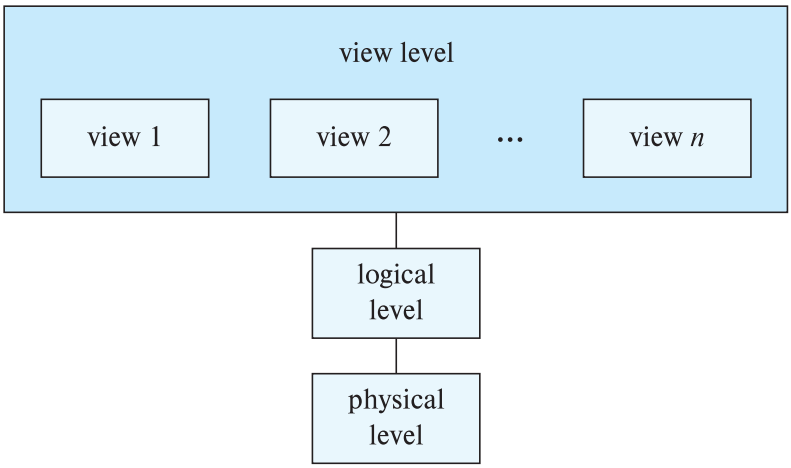


Figure 1.2 The three levels of data abstraction.

database-application developers. The database system allows application developers to store and retrieve data using the abstractions of the data model, and converts the abstract operations into operations on the low-level implementation.

An analogy to the concept of data types in programming languages may clarify the distinction among levels of abstraction. Many high-level programming languages support the notion of a structured type. We may describe the type of a record abstractly as follows:[1]

```
type instructor = record
                    ID : char (5);
                    name : char (20);
                    dept_name : char (20);
                    salary : numeric (8,2);
                  end;
```

This code defines a new record type called *instructor* with four fields. Each field has a name and a type associated with it. For example, **char**(20) specifies a string with 20 characters, while **numeric**(8,2) specifies a number with 8 digits, two of which are to the right of the decimal point. A university organization may have several such record types, including:

- *department*, with fields *dept_name*, *building*, and *budget*.
- *course*, with fields *course_id*, *title*, *dept_name*, and *credits*.
- *student*, with fields *ID*, *name*, *dept_name*, and *tot_cred*.

[1]The actual type declaration depends on the language being used. C and C++ use **struct** declarations. Java does not have such a declaration, but a simple class can be defined to the same effect.

At the physical level, an *instructor*, *department*, or *student* record can be described as a block of consecutive bytes. The 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. For example, there are many possible ways to store tables in files. One way is to store a table as a sequence of records in a file, with a special character (such as a comma) used to delimit the different attributes of a record, and another special character (such as a new-line character) may be used to delimit records. If all attributes have fixed length, the lengths of attributes may be stored separately, and delimiters may be omitted from the file. Variable length attributes could be handled by storing the length, followed by the data. Databases use a type of data structure called an index to support efficient retrieval of records; these too form part of the physical level.

At the logical level, each such record is described by a type definition, as in the previous code segment. The interrelationship of these record types is also defined at the logical level; a requirement that the *dept_name* value of an *instructor* record must appear in the *department* table is an example of such an interrelationship. 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. At the view level, several views of the database are defined, and a database user sees some or all of 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, clerks in the university registrar office can see only that part of the database that has information about students; they cannot access information about salaries of instructors.

1.3.4 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**. 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 also note that it is possible to create schemas that have problems, such as unnecessarily duplicated information. For example, suppose we store the department *budget* as an attribute of the *instructor* record. Then, whenever the value of the budget for a department (say the Physics department) changes, that change must be reflected in the records of all instructors associated with the department. In Chapter 7, we shall study how to distinguish good schema designs from bad schema designs.

Traditionally, logical schemas were changed infrequently, if at all. Many newer database applications, however, require more flexible logical schemas where, for example, different records in a single relation may have different attributes.

1.4 Database Languages

A database system provides a **data-definition language** (**DDL**) to specify the database schema and a **data-manipulation language** (**DML**) 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 SQL language. Almost all relational database systems employ the SQL language, which we cover in great detail in Chapter 3, Chapter 4, and Chapter 5.

1.4.1 Data-Definition Language

We specify a database schema by a set of definitions expressed by a special language called a data-definition language (DDL). The DDL is also used to specify additional properties of the 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 university requires that the account balance of a department must never be negative. The DDL provides facilities to specify such constraints. The database system checks these constraints every time the database is updated. In general, a constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, database systems implement only those integrity constraints that can be tested with minimal overhead:

- **Domain Constraints**. A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it

can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

- **Referential Integrity**. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the department listed for each course must be one that actually exists in the university. More precisely, the *dept_name* value in a *course* record must appear in the *dept_name* attribute of some record of the *department* relation. Database modifications can cause violations of referential integrity. When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation.
- **Authorization**. We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of **authorization**, the most common being: **read authorization**, which allows reading, but not modification, of data; **insert authorization**, which allows insertion of new data, but not modification of existing data; **update authorization**, which allows modification, but not deletion, of data; and **delete authorization**, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization.

The processing of DDL statements, just like those of any other programming language, generates some output. The output of the DDL is placed in the **data dictionary**, which contains **metadata**—that is, data about data. The data dictionary is considered to be a special type of table that can be accessed and updated only by the database system itself (not a regular user). The database system consults the data dictionary before reading or modifying actual data.

1.4.2 The SQL Data-Definition Language

SQL provides a rich DDL that allows one to define tables with data types and integrity constraints.

For instance, the following SQL DDL statement defines the *department* table:

```
create table department
    (dept_name     char (20),
     building      char (15),
     budget        numeric (12,2));
```

Execution of the preceding DDL statement creates the *department* table with three columns: *dept_name*, *building*, and *budget*, each of which has a specific data type associated with it. We discuss data types in more detail in Chapter 3.

The SQL DDL also supports a number of types of integrity constraints. For example, one can specify that the *dept_name* attribute value is a *primary key*, ensuring that no

two departments can have the same department name. As another example, one can specify that the *dept_name* attribute value appearing in any *instructor* record must also appear in the *dept_name* attribute of some record of the *department* table. We discuss SQL support for integrity constraints and authorizations in Chapter 3 and Chapter 4.

1.4.3 Data-Manipulation Language

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

- Retrieval of information stored in the database.
- Insertion of new information into the database.
- Deletion of information from the database.
- Modification of information stored in the database.

There are basically two types of data-manipulation language:

- **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.

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.

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 3 through 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 Chapter 15 and Chapter 16) translates DML queries into sequences of actions at the physical level of the database system. In Chapter 22, we study the processing of queries in the increasingly common parallel and distributed settings.

1.4.4 The SQL Data-Manipulation Language

The SQL query language is nonprocedural. A query takes as input several tables (possibly only one) and always returns a single table. Here is an example of an SQL query that finds the names of all instructors in the History department:

```
select instructor.name
from instructor
where instructor.dept_name = 'History';
```

The query specifies that those rows from the table *instructor* where the *dept_name* is History must be retrieved, and the *name* attribute of these rows must be displayed. The result of executing this query is a table with a single column labeled *name* and a set of rows, each of which contains the name of an instructor whose *dept_name* is History. If the query is run on the table in Figure 1.1, the result consists of two rows, one with the name El Said and the other with the name Califieri.

Queries may involve information from more than one table. For instance, the following query finds the instructor ID and department name of all instructors associated with a department with a budget of more than $95,000.

```
select instructor.ID, department.dept_name
from instructor, department
where instructor.dept_name= department.dept_name and
      department.budget > 95000;
```

If the preceding query were run on the tables in Figure 1.1, the system would find that there are two departments with a budget of greater than $95,000—Computer Science and Finance; there are five instructors in these departments. Thus, the result consists of a table with two columns (*ID*, *dept_name*) and five rows: (12121, Finance), (45565, Computer Science), (10101, Computer Science), (83821, Computer Science), and (76543, Finance).

1.4.5 Database Access from Application Programs

Non-procedural query languages such as SQL are not as powerful as a universal Turing machine; that is, there are some computations that are possible using a general-purpose programming language but are not possible using SQL. SQL also does not support actions such as input from users, output to displays, or communication over the network. Such computations and actions must be written in a *host* language, such as C/C++, Java, or Python, with embedded SQL queries that access the data in the database. **Application programs** are programs that are used to interact with the database in this fashion. Examples in a university system are programs that allow students to register for courses, generate class rosters, calculate student GPA, generate payroll checks, and perform other tasks.

To access the database, DML statements need to be sent from the host to the database where they will be executed. This is most commonly done by using 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 defines application program interfaces for use with C and several other languages. The Java Database Connectivity (JDBC) standard defines a corresponding interface for the Java language.

1.5 Database Design

Database systems are designed to manage large bodies of information. These large bodies of information do not exist in isolation. They are part of the operation of some enterprise whose end product may be information from the database or may be some device or service for which the database plays only a supporting role.

Database design mainly involves the design of the database schema. The design of a complete database application environment that meets the needs of the enterprise being modeled requires attention to a broader set of issues. In this text, we focus on the writing of database queries and the design of database schemas, but discuss application design later, in Chapter 9.

A high-level data model provides the database designer with a conceptual framework in which to specify the data requirements of the database users and how the database will be structured to fulfill these requirements. The initial phase of database design, then, is to characterize fully the data needs of the prospective database users. The database designer needs to interact extensively with domain experts and users to carry out this task. The outcome of this phase is a specification of user requirements.

Next, the designer chooses a data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database. The schema developed at this **conceptual-design** phase provides a detailed overview of the enterprise. The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another. The designer can also examine the design to remove any redundant features. The focus at this point is on describing the data and their relationships, rather than on specifying physical storage details.

In terms of the relational model, the conceptual-design process involves decisions on *what* attributes we want to capture in the database and *how to group* these attributes to form the various tables. The "what" part is basically a business decision, and we shall not discuss it further in this text. The "how" part is mainly a computer-science problem. There are principally two ways to tackle the problem. The first one is to use the entity-relationship model (Chapter 6); the other is to employ a set of algorithms (collectively known as **normalization** that takes as input the set of all attributes and generates a set of tables (Chapter 7).

A fully developed conceptual schema indicates the functional requirements of the enterprise. In a **specification of functional requirements**, users describe the kinds of oper-

ations (or transactions) that will be performed on the data. Example operations include modifying or updating data, searching for and retrieving specific data, and deleting data. At this stage of conceptual design, the designer can review the schema to ensure it meets functional requirements.

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases. In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database are specified. These features include the form of file organization and the internal storage structures; they are discussed in Chapter 13.

1.6 Database Engine

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

The storage manager is important because databases typically require a large amount of storage space. Corporate databases commonly range in size from hundreds of gigabytes to terabytes of data. A gigabyte is approximately 1 billion bytes, or 1000 megabytes (more precisely, 1024 megabytes), while a terabyte is approximately 1 trillion bytes or 1 million megabytes (more precisely, 1024 gigabytes). The largest enterprises have databases that reach into the multi-petabyte range (a petabyte is 1024 terabytes). Since the main memory of computers cannot store this much information, and since the contents of main memory are lost in a system crash, 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. Increasingly, solid-state disks (SSDs) are being used for database storage. SSDs are faster than traditional disks but also more costly.

The query processor is important because it helps the database system to simplify and facilitate access to data. The query processor allows database users to obtain good performance while being able to work at the view level and not be burdened with understanding the physical-level details of the implementation of the system. 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.

The transaction manager is important because it allows application developers to treat a sequence of database accesses as if they were a single unit that either happens in its entirety or not at all. This permits application developers to think at a higher level of

abstraction about the application without needing to be concerned with the lower-level details of managing the effects of concurrent access to the data and of system failures.

While database engines were traditionally centralized computer systems, today parallel processing is key for handling very large amounts of data efficiently. Modern database engines pay a lot of attention to parallel data storage and parallel query processing.

1.6.1 Storage Manager

The **storage manager** is the component of a database system 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 provided by the 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 conflicts.
- **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 can provide fast access to data items. Like the index in this textbook, a database index provides pointers to those data items that hold a particular value. For example, we could use an index to find the *instructor* record with a particular *ID*, or all *instructor* records with a particular *name*.

We discuss storage media, file structures, and buffer management in Chapter 12 and Chapter 13. Methods of accessing data efficiently are discussed in Chapter 14.

1.6.2 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.

Query evaluation is covered in Chapter 15, while the methods by which the query optimizer chooses from among the possible evaluation strategies are discussed in Chapter 16.

1.6.3 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 A is debited and another account 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 preserves the consistency of the database. That is, the value of the sum of the balances of A and B must be preserved. This correctness requirement is called **consistency**. Finally, after the successful execution of a funds transfer, the new values of the balances 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 properly define 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 **recovery manager**. 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, it must 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. The **transaction manager** consists of the concurrency-control manager and the recovery manager.

The basic concepts of transaction processing are covered in Chapter 17. The management of concurrent transactions is covered in Chapter 18. Chapter 19 covers failure recovery in detail.

The concept of a transaction has been applied broadly in database systems and applications. While the initial use of transactions was in financial applications, the concept is now used in real-time applications in telecommunication, as well as in the management of long-duration activities such as product design or administrative workflows.

1.7 Database and Application Architecture

We are now in a position to provide a single picture of the various components of a database system and the connections among them. Figure 1.3 shows the architecture of a database system that runs on a centralized server machine. The figure summarizes how different types of users interact with a database, and how the different components of a database engine are connected to each other.

The centralized architecture shown in Figure 1.3 is applicable to *shared-memory* server architectures, which have multiple CPUs and exploit parallel processing, but all

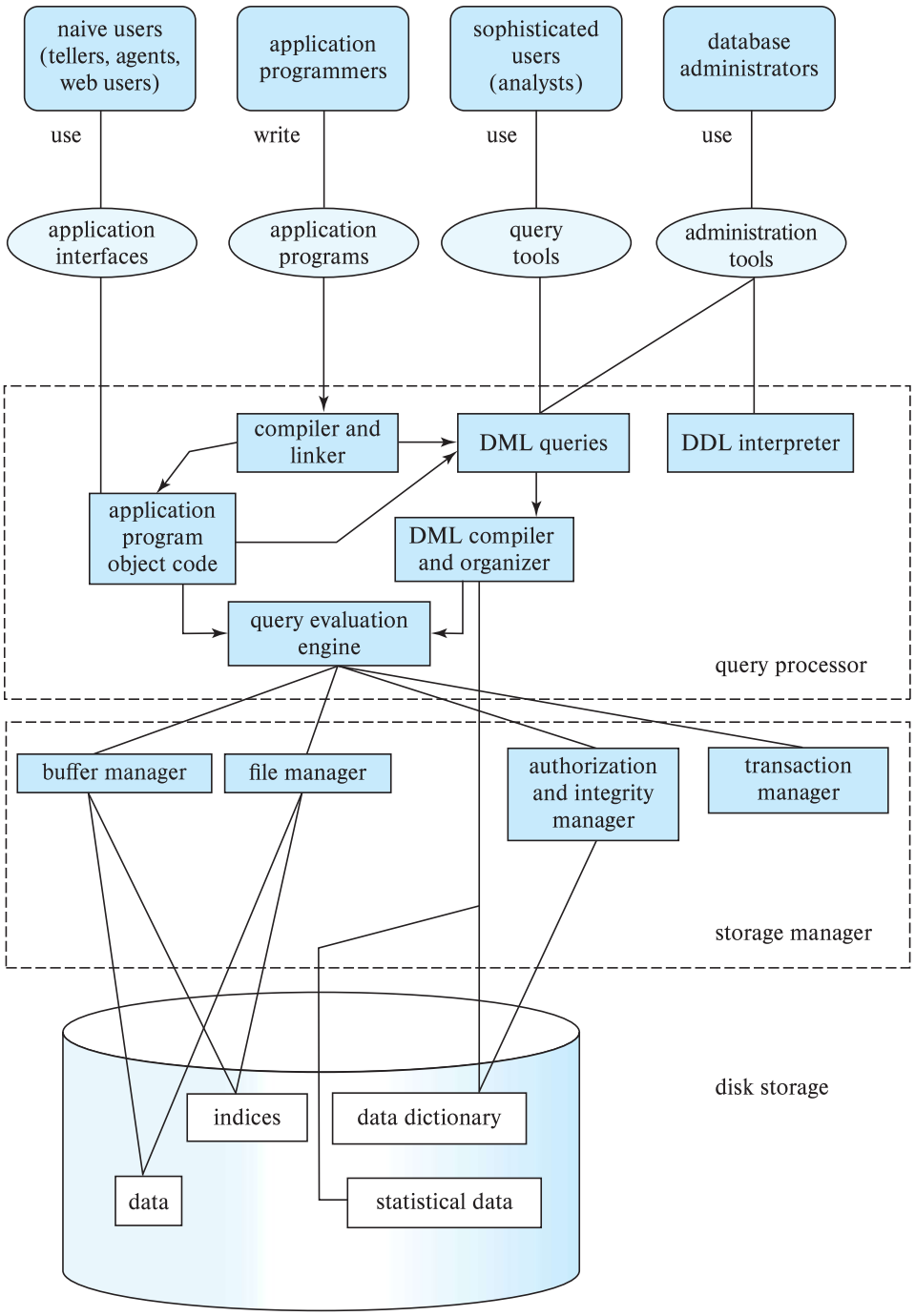


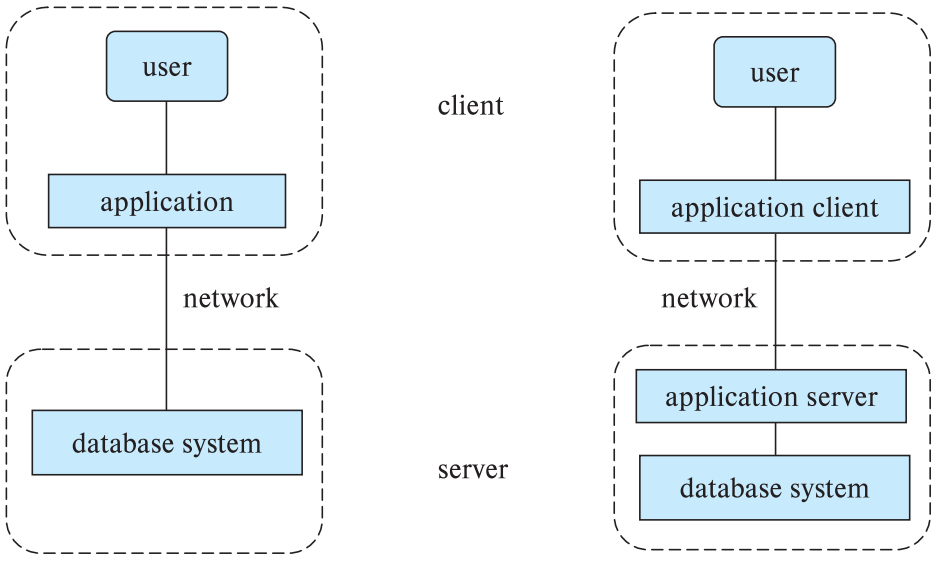
Figure 1.3 System structure.

the CPUs access a common shared memory. To scale up to even larger data volumes and even higher processing speeds, *parallel databases* are designed to run on a cluster consisting of multiple machines. Further, *distributed databases* allow data storage and query processing across multiple geographically separated machines.

In Chapter 20, we cover the general structure of modern computer systems, with a focus on parallel system architectures. Chapter 21 and Chapter 22 describe how query processing can be implemented to exploit parallel and distributed processing. Chapter 23 presents a number of issues that arise in processing transactions in a parallel or a distributed database and describes how to deal with each issue. The issues include how to store data, how to ensure atomicity of transactions that execute at multiple sites, how to perform concurrency control, and how to provide high availability in the presence of failures.

We now consider the architecture of applications that use databases as their back-end. Database applications can be partitioned into two or three parts, as shown in Figure 1.4. Earlier-generation database applications used a **two-tier architecture**, where the application resides at the client machine, and invokes database system functionality at the server machine through query language statements.

In contrast, modern database applications use a **three-tier architecture**, where the client machine acts as merely a front end and does not contain any direct database calls; web browsers and mobile applications are the most commonly used application clients today. The front end communicates with an **application server**. 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 provide better security as well as better performance than two-tier applications.



(a) Two-tier architecture (b) Three-tier architecture

Figure 1.4 Two-tier and three-tier architectures.

1.8 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.

1.8.1 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.

- **Naïve users** are unsophisticated users who interact with the system by using predefined user interfaces, such as web or mobile applications. The typical user interface for naïve users is a forms interface, where the user can fill in appropriate fields of the form. Naïve users may also view read *reports* generated from the database.

  As an example, consider a student, who during class registration period, wishes to register for a class by using a web interface. Such a user connects to a web application program that runs at a web server. The application first verifies the identity of the user and then allows her to access a form where she enters the desired information. The form information is sent back to the web application at the server, which then determines if there is room in the class (by retrieving information from the database) and if so adds the student information to the class roster in the database.

- **Application programmers** are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces.

- **Sophisticated users** interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.

1.8.2 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**. The DBA may specify some parameters pertaining to the physical organization of the data and the indices to be created.

- **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 a user tries to access the data in the system.
- **Routine maintenance**. Examples of the database administrator's routine maintenance activities are:
  - Periodically backing up the database 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.

1.9 History of Database Systems

Information 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 Herman Hollerith, were used at the very beginning of the twentieth century to record U.S. census data, and mechanical systems were used to 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 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 early 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 the advent of disks, the network and hierarchical data models were developed, which 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 Edgar Codd in 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.

- **Late 1970s and 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. The fully functional System R prototype led to IBM's first relational database product, SQL/DS. At the same time, the Ingres system was being developed at the University of California at Berkeley. It led to a commercial product of the same name. Also around this time, the first version of Oracle was released. 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 and hierarchical databases. Programmers using those older models were forced to deal with many low-level implementation details, and they 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 system, 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.

- **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.

  In the early 1990s, decision support and querying re-emerged as a major application area for databases. Tools for analyzing large amounts of data saw a large growth in usage. Many database vendors introduced parallel database products in this period. Database vendors also began to add object-relational support to their databases.

  The major event of the 1990s 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.

- **2000s**: The types of data stored in database systems evolved rapidly during this period. Semi-structured data became increasingly important. XML emerged as a data-exchange standard. JSON, a more compact data-exchange format well suited for storing objects from JavaScript or other programming languages subsequently grew increasingly important. Increasingly, such data were stored in relational database systems as support for the XML and JSON formats was added to the major commercial systems. Spatial data (that is, data that include geographic information) saw widespread use in navigation systems and advanced applications. Database systems added support for such data.

  Open-source database systems, notably PostgreSQL and MySQL saw increased use. "Auto-admin" features were added to database systems in order to allow automatic reconfiguration to adapt to changing workloads. This helped reduce the human workload in administering a database.

  Social network platforms grew at a rapid pace, creating a need to manage data about connections between people and their posted data, that did not fit well into a tabular row-and-column format. This led to the development of graph databases.

  In the latter part of the decade, the use of data analytics and **data mining** in enterprises became ubiquitous. Database systems were developed specifically to serve this market. These systems featured physical data organizations suitable for analytic processing, such as "column-stores," in which tables are stored by column rather than the traditional row-oriented storage of the major commercial database systems.

  The huge volumes of data, as well as the fact that much of the data used for analytics was textual or semi-structured, led to the development of programming frameworks, such as *map-reduce*, to facilitate application programmers' use of parallelism in analyzing data. In time, support for these features migrated into traditional database systems. Even in the late 2010s, debate continued in the database

research community over the relative merits of a single database system serving both traditional transaction processing applications and the newer data-analysis applications versus maintaining separate systems for these roles.

The variety of new data-intensive applications and the need for rapid development, particularly by startup firms, led to "NoSQL" systems that provide a lightweight form of data management. The name was derived from those systems' lack of support for the ubiquitous database query language SQL, though the name is now often viewed as meaning "not only SQL." The lack of a high-level query language based on the relational model gave programmers greater flexibility to work with new types of data. The lack of traditional database systems' support for strict data consistency provided more flexibility in an application's use of distributed data stores. The NoSQL model of "eventual consistency" allowed for distributed copies of data to be inconsistent as long they would eventually converge in the absence of further updates.

- **2010s**: The limitations of NoSQL systems, such as lack of support for consistency, and lack of support for declarative querying, were found acceptable by many applications (e.g., social networks), in return for the benefits they provided such as scalability and availability. However, by the early 2010s it was clear that the limitations made life significantly more complicated for programmers and database administrators. As a result, these systems evolved to provide features to support stricter notions of consistency, while continuing to support high scalability and availability. Additionally, these systems increasingly support higher levels of abstraction to avoid the need for programmers to have to reimplement features that are standard in a traditional database system.

  Enterprises are increasingly outsourcing the storage and management of their data. Rather than maintaining in-house systems and expertise, enterprises may store their data in "cloud" services that host data for various clients in multiple, widely distributed server farms. Data are delivered to users via web-based services. Other enterprises are outsourcing not only the storage of their data but also whole applications. In such cases, termed "software as a service," the vendor not only stores the data for an enterprise but also runs (and maintains) the application software. These trends result in significant savings in costs, but they create new issues not only in responsibility for security breaches, but also in data ownership, particularly in cases where a government requests access to data.

  The huge influence of data and data analytics in daily life has made the management of data a frequent aspect of the news. There is an unresolved tradeoff between an individual's right of privacy and society's need to know. Various national governments have put regulations on privacy in place. High-profile security breaches have created a public awareness of the challenges in cybersecurity and the risks of storing data.

1.10 Summary

- A database-management system (DBMS) consists of a collection of interrelated data and a collection of programs to access those 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 relational data model is the most widely deployed model for storing data in databases. Other data models are the object-oriented model, the object-relational model, and semi-structured data models.
- 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.
- A data-definition language (DDL) is a language for specifying the database schema and other properties of the data.
- Database design mainly involves the design of the database schema. The entity-relationship (E-R) data model is a widely used model for database design. It provides a convenient graphical representation to view data, relationships, and constraints.
- A database system has several subsystems.
    - 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.

  - The query processor subsystem compiles and executes DDL and DML statements.
- Transaction management ensures that the database remains in a consistent (correct) state despite system failures. The transaction manager ensures that concurrent transaction executions proceed without conflicts.
- The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or parallel, involving multiple machines. Distributed databases span multiple geographically separated machines.
- Database applications are typically broken up into a front-end part that runs at client machines and a part that runs at the backend. 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.
- 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.
- Data-analysis techniques attempt to automatically discover rules and patterns from data. The field of data mining combines knowledge-discovery techniques invented by artificial intelligence researchers and statistical analysts with efficient implementation techniques that enable them to be used on extremely large databases.

Review Terms

- Database-management system (DBMS)
- Database-system applications
- Online transaction processing
- Data analytics
- File-processing systems
- Data inconsistency
- Consistency constraints
- Data abstraction
  - Physical level
  - Logical level
  - View level
- Instance
- Schema
  - Physical schema
  - Logical schema
  - Subschema
- Physical data independence
- Data models
  - Entity-relationship model
  - Relational data model
  - Semi-structured data model
  - Object-based data model

- Database languages
  - Data-definition language
  - Data-manipulation language
    - Procedural DML
    - Declarative DML
    - nonprocedural DML
  - Query language
- Data-definition language
  - Domain Constraints
  - Referential Integrity
  - Authorization
    - Read authorization
    - Insert authorization
    - Update authorization
    - Delete authorization
- Metadata
- Application program
- Database design
  - Conceptual design
  - Normalization
  - Specification of functional requirements
  - Physical-design phase
- Database Engine
  - Storage manager
    - Authorization and integrity manager
    - Transaction manager
    - File manager
    - Buffer manager
    - Data files
    - Data dictionary
    - Indices
  - Query processor
    - DDL interpreter
    - DML compiler
    - Query optimization
    - Query evaluation engine
  - Transactions
    - Atomicity
    - Consistency
    - Durability
    - Recovery manager
    - Failure recovery
    - Concurrency-control manager
- Database Architecture
  - Centralized
  - Parallel
  - Distributed
- Database Application Architecture
  - Two-tier
  - Three-tier
  - Application server
- Database administrator (DBA)

Practice Exercises

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

1.2 List five ways in which the type declaration system of a language such as Java or C++ differs from the data definition language used in a database.

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

1.4 Suppose you want to build a video site similar to YouTube. Consider each of the points listed in Section 1.2 as disadvantages of keeping data in a file-processing system. Discuss the relevance of each of these points to the storage of actual video data, and to metadata about the video, such as title, the user who uploaded it, tags, and which users viewed it.

1.5 Keyword queries used in web search are quite different from database queries. List key differences between the two, in terms of the way the queries are specified and in terms of what is the result of a query.

Exercises

1.6 List four applications you have used that most likely employed a database system to store persistent data.

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

1.8 Explain the concept of physical data independence and its importance in database systems.

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

1.10 List at least two reasons why database systems support data manipulation using a declarative query language such as SQL, instead of just providing a library of C or C++ functions to carry out data manipulation.

1.11 Assume that two students are trying to register for a course in which there is only one open seat. What component of a database system prevents both students from being given that last seat?

1.12 Explain the difference between two-tier and three-tier application architectures. Which is better suited for web applications? Why?

1.13 List two features developed in the 2000s and that help database systems handle data-analytics workloads.

1.14 Explain why NoSQL systems emerged in the 2000s, and briefly contrast their features with traditional database systems.

1.15 Describe at least three tables that might be used to store information in a social-networking system such as Facebook.

Tools

There are a large number of commercial database systems in use today. The major ones include: IBM DB2 (www.ibm.com/software/data/db2), Oracle (www.oracle.com), Microsoft SQL Server (www.microsoft.com/sql), IBM Informix (www.ibm.com/software/data/informix), SAP Adaptive Server Enterprise (formerly Sybase) (www.sap.com/products/sybase-ase.html), and SAP HANA (www.sap.com/products/hana.html). Some of these systems are available free for personal or non-commercial 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), PostgreSQL (www.postgresql.org), and the embedded database SQLite (www.sqlite.org).

A more complete list of links to vendor web sites and other information is available from the home page of this book, at db-book.com.

Further Reading

[Codd (1970)] is the landmark paper that introduced the relational model. Textbook coverage of database systems is provided by [O'Neil and O'Neil (2000)], [Ramakrishnan and Gehrke (2002)], [Date (2003)], [Kifer et al. (2005)], [Garcia-Molina et al. (2008)], and [Elmasri and Navathe (2016)], in addition to this textbook,

A review of accomplishments in database management and an assessment of future research challenges appears in [Abadi et al. (2016)]. The home page of the ACM Special Interest Group on Management of Data (www.acm.org/sigmod) provides a wealth of information about database research. Database vendor web sites (see the Tools section above) provide details about their respective products.

Bibliography

[Abadi et al. (2016)] D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. RÂ´e, D. Suciu, M. Stonebraker, T. Walter, and J. Widom, "The Beckman Report on Database Research", *Communications of the ACM*, Volume 59, Number 2 (2016), pages 92–99.

[Codd (1970)] E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, Volume 13, Number 6 (1970), pages 377–387.

[Date (2003)] C. J. Date, *An Introduction to Database Systems*, 8th edition, Addison Wesley (2003).

[Elmasri and Navathe (2016)] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th edition, Addison Wesley (2016).

[Garcia-Molina et al. (2008)] H. Garcia-Molina, J. D. Ullman, and J. D. Widom, *Database Systems: The Complete Book*, 2nd edition, Prentice Hall (2008).

[Kifer et al. (2005)] M. Kifer, A. Bernstein, and P. Lewis, *Database Systems: An Application Oriented Approach, Complete Version*, 2nd edition, Addison Wesley (2005).

[O'Neil and O'Neil (2000)] P. O'Neil and E. O'Neil, *Database: Principles, Programming, Performance*, 2nd edition, Morgan Kaufmann (2000).

[Ramakrishnan and Gehrke (2002)] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd edition, McGraw Hill (2002).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.

PART 1

RELATIONAL LANGUAGES

A data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. The relational model uses a collection of tables to represent both data and the relationships among those data. Its conceptual simplicity has led to its widespread adoption; today a vast majority of database products are based on the relational model. The relational model describes data at the logical and view levels, abstracting away low-level details of data storage.

To make data from a relational database available to users, we have to address how users specify requests for retrieving and updating data. Several query languages have been developed for this task, which are covered in this part.

Chapter 2 introduces the basic concepts underlying relational databases, including the coverage of relational algebra—a formal query language that forms the basis for SQL. The language SQL is the most widely used relational query language today and is covered in great detail in this part.

Chapter 3 provides an overview of the SQL query language, including the SQL data definition, the basic structure of SQL queries, set operations, aggregate functions, nested subqueries, and modification of the database.

Chapter 4 provides further details of SQL, including join expressions, views, transactions, integrity constraints that are enforced by the database, and authorization mechanisms that control what access and update actions can be carried out by a user.

Chapter 5 covers advanced topics related to SQL including access to SQL from programming languages, functions, procedures, triggers, recursive queries, and advanced aggregation features.

CHAPTER 2

Introduction to the Relational Model

The relational model remains the primary data model for commercial data-processing applications. It attained its primary position because of its simplicity, which eases the job of the programmer, compared to earlier data models such as the network model or the hierarchical model. It has retained this position by incorporating various new features and capabilities over its half-century of existence. Among those additions are object-relational features such as complex data types and stored procedures, support for XML data, and various tools to support semi-structured data. The relational model's independence from any specific underlying low-level data structures has allowed it to persist despite the advent of new approaches to data storage, including modern column-stores that are designed for large-scale data mining.

In this chapter, we first study the fundamentals of the relational model. A substantial theory exists for relational databases. In Chapter 6 and Chapter 7, we shall examine aspects of database theory that help in the design of relational database schemas, while in Chapter 15 and Chapter 16 we discuss aspects of the theory dealing with efficient processing of queries. In Chapter 27, we study aspects of formal relational languages beyond our basic coverage in this chapter.

2.1 Structure of Relational Databases

A relational database consists of a collection of **tables**, each of which is assigned a unique name. For example, consider the *instructor* table of Figure 2.1, which stores information about instructors. The table has four column headers: *ID*, *name*, *dept_name*, and *salary*. Each row of this table records information about an instructor, consisting of the instructor's *ID*, *name*, *dept_name*, and *salary*. Similarly, the *course* table of Figure 2.2 stores information about courses, consisting of a *course_id*, *title*, *dept_name*, and *credits*, for each course. Note that each instructor is identified by the value of the column *ID*, while each course is identified by the value of the column *course_id*.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 2.1 The *instructor* relation.

Figure 2.3 shows a third table, *prereq*, which stores the prerequisite courses for each course. The table has two columns, *course_id* and *prereq_id*. Each row consists of a pair of course identifiers such that the second course is a prerequisite for the first course.

Thus, a row in the *prereq* table indicates that two courses are *related* in the sense that one course is a prerequisite for the other. As another example, when we consider the table *instructor*, a row in the table can be thought of as representing the relationship

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

Figure 2.2 The *course* relation.

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Figure 2.3 The *prereq* relation.

between a specified *ID* and the corresponding values for *name*, *dept_name*, and *salary* values.

In general, a row in a table represents a *relationship* among a set of values. Since a table is a collection of such relationships, there is a close correspondence between the concept of *table* and the mathematical concept of *relation*, from which the relational data model takes its name. In mathematical terminology, a *tuple* is simply a sequence (or list) of values. A relationship between n values is represented mathematically by an *n-tuple* of values, that is, a tuple with n values, which corresponds to a row in a table.

Thus, in the relational model the term **relation** is used to refer to a table, while the term **tuple** is used to refer to a row. Similarly, the term **attribute** refers to a column of a table.

Examining Figure 2.1, we can see that the relation *instructor* has four attributes: *ID*, *name*, *dept_name*, and *salary*.

We use the term **relation instance** to refer to a specific instance of a relation, that is, containing a specific set of rows. The instance of *instructor* shown in Figure 2.1 has 12 tuples, corresponding to 12 instructors.

In this chapter, we shall be using a number of different relations to illustrate the various concepts underlying the relational data model. These relations represent part of a university. To simplify our presentation, we exclude much of the data an actual university database would contain. We shall discuss criteria for the appropriateness of relational structures in great detail in Chapter 6 and Chapter 7.

The order in which tuples appear in a relation is irrelevant, since a relation is a *set* of tuples. Thus, whether the tuples of a relation are listed in sorted order, as in Figure 2.1, or are unsorted, as in Figure 2.4, does not matter; the relations in the two figures are the same, since both contain the same set of tuples. For ease of exposition, we generally show the relations sorted by their first attribute.

For each attribute of a relation, there is a set of permitted values, called the **domain** of that attribute. Thus, the domain of the *salary* attribute of the *instructor* relation is the set of all possible salary values, while the domain of the *name* attribute is the set of all possible instructor names.

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

Figure 2.4 Unsorted display of the *instructor* relation.

We require that, for all relations r, the domains of all attributes of r be atomic. A domain is **atomic** if elements of the domain are considered to be indivisible units. For example, suppose the table *instructor* had an attribute *phone_number*, which can store a set of phone numbers corresponding to the instructor. Then the domain of *phone_number* would not be atomic, since an element of the domain is a set of phone numbers, and it has subparts, namely, the individual phone numbers in the set.

The important issue is not what the domain itself is, but rather how we use domain elements in our database. Suppose now that the *phone_number* attribute stores a single phone number. Even then, if we split the value from the phone number attribute into a country code, an area code, and a local number, we would be treating it as a non-atomic value. If we treat each phone number as a single indivisible unit, then the attribute *phone_number* would have an atomic domain.

The **null value** is a special value that signifies that the value is unknown or does not exist. For example, suppose as before that we include the attribute *phone_number* in the *instructor* relation. It may be that an instructor does not have a phone number at all, or that the telephone number is unlisted. We would then have to use the null value to signify that the value is unknown or does not exist. We shall see later that null values cause a number of difficulties when we access or update the database, and thus they should be eliminated if at all possible. We shall assume null values are absent initially, and in Section 3.6 we describe the effect of nulls on different operations.

The relatively strict structure of relations results in several important practical advantages in the storage and processing of data, as we shall see. That strict structure is suitable for well-defined and relatively static applications, but it is less suitable for applications where not only data but also the types and structure of those data change over time. A modern enterprise needs to find a good balance between the efficiencies of structured data and those situations where a predetermined structure is limiting.

2.2 Database Schema

When we talk about a database, we must differentiate between the **database schema**, which is the logical design of the database, and the **database instance**, which is a snapshot of the data in the database at a given instant in time.

The concept of a relation corresponds to the programming-language notion of a variable, while the concept of a **relation schema** corresponds to the programming-language notion of type definition.

In general, a relation schema consists of a list of attributes and their corresponding domains. We shall not be concerned about the precise definition of the domain of each attribute until we discuss the SQL language in Chapter 3.

The concept of a relation instance corresponds to the programming-language notion of a value of a variable. The value of a given variable may change with time; similarly the contents of a relation instance may change with time as the relation is updated. In contrast, the schema of a relation does not generally change.

Although it is important to know the difference between a relation schema and a relation instance, we often use the same name, such as *instructor*, to refer to both the schema and the instance. Where required, we explicitly refer to the schema or to the instance, for example "the *instructor* schema," or "an instance of the *instructor* relation." However, where it is clear whether we mean the schema or the instance, we simply use the relation name.

Consider the *department* relation of Figure 2.5. The schema for that relation is:

department (*dept_name*, *building*, *budget*)

Note that the attribute *dept_name* appears in both the *instructor* schema and the *department* schema. This duplication is not a coincidence. Rather, using common attributes in relation schemas is one way of relating tuples of distinct relations. For example, suppose we wish to find the information about all the instructors who work in the Watson building. We look first at the *department* relation to find the *dept_name* of all the departments housed in Watson. Then, for each such department, we look in

dept_name	*building*	*budget*
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Elec. Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000

Figure 2.5 The *department* relation.

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

Figure 2.6 The *section* relation.

the *instructor* relation to find the information about the instructor associated with the corresponding *dept_name*.

Each course in a university may be offered multiple times, across different semesters, or even within a semester. We need a relation to describe each individual offering, or section, of the class. The schema is:

section (*course_id*, *sec_id*, *semester*, *year*, *building*, *room_number*, *time_slot_id*)

Figure 2.6 shows a sample instance of the *section* relation.

We need a relation to describe the association between instructors and the class sections that they teach. The relation schema to describe this association is:

teaches (*ID*, *course_id*, *sec_id*, *semester*, *year*)

Figure 2.7 shows a sample instance of the *teaches* relation.

As you can imagine, there are many more relations maintained in a real university database. In addition to those relations we have listed already, *instructor*, *department*, *course*, *section*, *prereq*, and *teaches*, we use the following relations in this text:

- *student* (*ID*, *name*, *dept_name*, *tot_cred*)
- *advisor* (*s_id*, *i_id*)
- *takes* (*ID*, *course_id*, *sec_id*, *semester*, *year*, *grade*)

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

Figure 2.7 The *teaches* relation.

- *classroom* (*building*, *room_number*, *capacity*)
- *time_slot* (*time_slot_id*, *day*, *start_time*, *end_time*)

2.3 Keys

We must have a way to specify how tuples within a given relation are distinguished. This is expressed in terms of their attributes. That is, the values of the attribute values of a tuple must be such that they can *uniquely identify* the tuple. In other words, no two tuples in a relation are allowed to have exactly the same value for all attributes.[1]

A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example, the *ID* attribute of the relation *instructor* is sufficient to distinguish one instructor tuple from another. Thus, *ID* is a superkey. The *name* attribute of *instructor*, on the other hand, is not a superkey, because several instructors might have the same name.

Formally, let R denote the set of attributes in the schema of relation r. If we say that a subset K of R is a *superkey* for r, we are restricting consideration to instances of relations r in which no two distinct tuples have the same values on all attributes in K. That is, if t_1 and t_2 are in r and $t_1 \neq t_2$, then $t_1.K \neq t_2.K$.

[1] Commercial database systems relax the requirement that a relation is a set and instead allow duplicate tuples. This is discussed further in Chapter 3.

A superkey may contain extraneous attributes. For example, the combination of *ID* and *name* is a superkey for the relation *instructor*. If K is a superkey, then so is any superset of K. We are often interested in superkeys for which no proper subset is a superkey. Such minimal superkeys are called **candidate keys**.

It is possible that several distinct sets of attributes could serve as a candidate key. Suppose that a combination of *name* and *dept_name* is sufficient to distinguish among members of the *instructor* relation. Then, both {*ID*} and {*name*, *dept_name*} are candidate keys. Although the attributes *ID* and *name* together can distinguish *instructor* tuples, their combination, {*ID*, *name*}, does not form a candidate key, since the attribute *ID* alone is a candidate key.

We shall use the term **primary key** to denote a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation. A key (whether primary, candidate, or super) is a property of the entire relation, rather than of the individual tuples. Any two individual tuples in the relation are prohibited from having the same value on the key attributes at the same time. The designation of a key represents a constraint in the real-world enterprise being modeled. Thus, primary keys are also referred to as **primary key constraints**.

It is customary to list the primary key attributes of a relation schema before the other attributes; for example, the *dept_name* attribute of *department* is listed first, since it is the primary key. Primary key attributes are also underlined.

Consider the *classroom* relation:

$$classroom\ (\underline{building}, \underline{room\_number}, capacity)$$

Here the primary key consists of two attributes, *building* and *room_number*, which are underlined to indicate they are part of the primary key. Neither attribute by itself can uniquely identify a classroom, although together they uniquely identify a classroom. Also consider the *time_slot* relation:

$$time\_slot\ (\underline{time\_slot\_id}, \underline{day}, \underline{start\_time}, end\_time)$$

Each section has an associated *time_slot_id*. The *time_slot* relation provides information on which days of the week, and at what times, a particular *time_slot_id* meets. For example, *time_slot_id* 'A' may meet from 8.00 AM to 8.50 AM on Mondays, Wednesdays, and Fridays. It is possible for a time slot to have multiple sessions within a single day, at different times, so the *time_slot_id* and *day* together do not uniquely identify the tuple. The primary key of the *time_slot* relation thus consists of the attributes *time_slot_id*, *day*, and *start_time*, since these three attributes together uniquely identify a time slot for a course.

Primary keys must be chosen with care. As we noted, the name of a person is insufficient, because there may be many people with the same name. In the United States, the social security number attribute of a person would be a candidate key. Since non-U.S. residents usually do not have social security numbers, international enterprises must

generate their own unique identifiers. An alternative is to use some unique combination of other attributes as a key.

The primary key should be chosen such that its attribute values are never, or are very rarely, changed. For instance, the address field of a person should not be part of the primary key, since it is likely to change. Social security numbers, on the other hand, are guaranteed never to change. Unique identifiers generated by enterprises generally do not change, except if two enterprises merge; in such a case the same identifier may have been issued by both enterprises, and a reallocation of identifiers may be required to make sure they are unique.

Figure 2.8 shows the complete set of relations that we use in our sample university schema, with primary-key attributes underlined.

Next, we consider another type of constraint on the contents of relations, called foreign-key constraints. Consider the attribute *dept_name* of the *instructor* relation. It would not make sense for a tuple in *instructor* to have a value for *dept_name* that does not correspond to a department in the *department* relation. Thus, in any database instance, given any tuple, say t_a, from the *instructor* relation, there must be some tuple, say t_b, in the *department* relation such that the value of the *dept_name* attribute of t_a is the same as the value of the primary key, *dept_name*, of t_b.

A **foreign-key constraint** from attribute(s) A of relation r_1 to the primary-key B of relation r_2 states that on any database instance, the value of A for each tuple in r_1 must also be the value of B for some tuple in r_2. Attribute set A is called a **foreign key** from r_1, referencing r_2. The relation r_1 is also called the **referencing relation** of the foreign-key constraint, and r_2 is called the **referenced relation**.

For example, the attribute *dept_name* in *instructor* is a foreign key from *instructor*, referencing *department*; note that *dept_name* is the primary key of *department*. Similarly,

---

classroom(<u>*building*</u>, <u>*room_number*</u>, *capacity*)
department(<u>*dept_name*</u>, *building*, *budget*)
course(<u>*course_id*</u>, *title*, *dept_name*, *credits*)
instructor(<u>*ID*</u>, *name*, *dept_name*, *salary*)
section(<u>*course_id*</u>, <u>*sec_id*</u>, <u>*semester*</u>, <u>*year*</u>, *building*, *room_number*, *time_slot_id*)
teaches(<u>*ID*</u>, <u>*course_id*</u>, <u>*sec_id*</u>, <u>*semester*</u>, <u>*year*</u>)
student(<u>*ID*</u>, *name*, *dept_name*, *tot_cred*)
takes(<u>*ID*</u>, <u>*course_id*</u>, <u>*sec_id*</u>, <u>*semester*</u>, <u>*year*</u>, *grade*)
advisor(<u>*s_ID*</u>, *i_ID*)
time_slot(<u>*time_slot_id*</u>, <u>*day*</u>, <u>*start_time*</u>, *end_time*)
prereq(<u>*course_id*</u>, <u>*prereq_id*</u>)

Figure 2.8 Schema of the university database.

the attributes *building* and *room_number* of the *section* relation together form a foreign key referencing the *classroom* relation.

Note that in a foreign-key constraint, the referenced attribute(s) must be the primary key of the referenced relation. The more general case, a referential-integrity constraint, relaxes the requirement that the referenced attributes form the primary key of the referenced relation.

As an example, consider the values in the *time_slot_id* attribute of the *section* relation. We require that these values must exist in the *time_slot_id* attribute of the *time_slot* relation. Such a requirement is an example of a referential integrity constraint. In general, a **referential integrity constraint** requires that the values appearing in specified attributes of any tuple in the referencing relation also appear in specified attributes of at least one tuple in the referenced relation.

Note that *time_slot* does not form a primary key of the *time_slot* relation, although it is a part of the primary key; thus, we cannot use a foreign-key constraint to enforce the above constraint. In fact, foreign-key constraints are a *special case* of referential integrity constraints, where the referenced attributes form the primary key of the referenced relation. Database systems today typically support foreign-key constraints, but they do not support referential integrity constraints where the referenced attribute is not a primary key.

2.4 Schema Diagrams

A database schema, along with primary key and foreign-key constraints, can be depicted by **schema diagrams**. Figure 2.9 shows the schema diagram for our university organization. Each relation appears as a box, with the relation name at the top in blue and the attributes listed inside the box.

Primary-key attributes are shown underlined. Foreign-key constraints appear as arrows from the foreign-key attributes of the referencing relation to the primary key of the referenced relation. We use a two-headed arrow, instead of a single-headed arrow, to indicate a referential integrity constraint that is not a foreign-key constraints. In Figure 2.9, the line with a two-headed arrow from *time_slot_id* in the *section* relation to *time_slot_id* in the *time_slot* relation represents the referential integrity constraint from *section.time_slot_id* to *time_slot.time_slot_id*.

Many database systems provide design tools with a graphical user interface for creating schema diagrams.[2] We shall discuss a different diagrammatic representation of schemas, called the entity-relationship diagram, at length in Chapter 6; although there are some similarities in appearance, these two notations are quite different, and should not be confused for one another.

[2]The two-headed arrow notation to represent referential integrity constraints has been introduced by us and is not supported by any tool as far as we know; the notations for primary and foreign keys, however, are widely used.

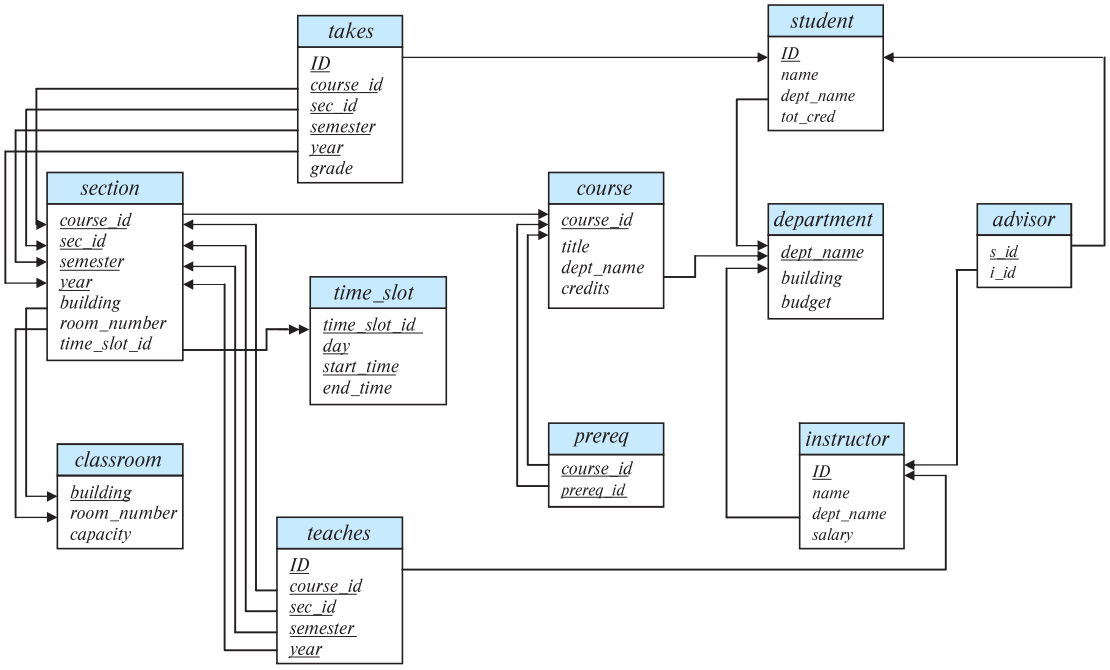


Figure 2.9 Schema diagram for the university database.

2.5 Relational Query Languages

A **query language** is a language in which a user requests information from the database. These languages are usually on a level higher than that of a standard programming language. Query languages can be categorized as imperative, functional, or declarative. In an **imperative query language**, the user instructs the system to perform a specific sequence of operations on the database to compute the desired result; such languages usually have a notion of state variables, which are updated in the course of the computation.

In a **functional query language**, the computation is expressed as the evaluation of functions that may operate on data in the database or on the results of other functions; functions are side-effect free, and they do not update the program state.[3] In a **declarative query language**, the user describes the desired information without giving a specific sequence of steps or function calls for obtaining that information; the desired information is typically described using some form of mathematical logic. It is the job of the database system to figure out how to obtain the desired information.

[3]The term *procedural language* has been used in earlier editions of the book to refer to languages based on procedure invocations, which include functional languages; however, the term is also widely used to refer to imperative languages. To avoid confusion we no longer use the term.

There are a number of "pure" query languages.

- The *relational algebra*, which we describe in Section 2.6, is a functional query language.[4] The relational algebra forms the theoretical basis of the SQL query language.
- The tuple relational calculus and domain relational calculus, which we describe in Chapter 27 (available online) are declarative.

These query languages are terse and formal, lacking the "syntactic sugar" of commercial languages, but they illustrate the fundamental techniques for extracting data from the database.

Query languages used in practice, such as the SQL query language, include elements of the imperative, functional, and declarative approaches. We study the very widely used query language SQL in Chapter 3 through Chapter 5.

2.6 The Relational Algebra

The relational algebra consists of a set of operations that take one or two relations as input and produce a new relation as their result.

Some of these operations, such as the select, project, and rename operations, are called *unary* operations because they operate on one relation. The other operations, such as union, Cartesian product, and set difference, operate on pairs of relations and are, therefore, called *binary* operations.

Although the relational algebra operations form the basis for the widely used SQL query language, database systems do not allow users to write queries in relational algebra. However, there are implementations of relational algebra that have been built for students to practice relational algebra queries. The website of our book, db-book.com, under the link titled Laboratory Material, provides pointers to a few such implementations.

It is worth recalling at this point that since a relation is a set of tuples, relations cannot contain duplicate tuples. In practice, however, tables in database systems are permitted to contain duplicates unless a specific constraint prohibits it. But, in discussing the formal relational algebra, we require that duplicates be eliminated, as is required by the mathematical definition of a set. In Chapter 3 we discuss how relational algebra can be extended to work on multisets, which are sets that can contain duplicates.

[4] Unlike modern functional languages, relational algebra supports only a small number of predefined functions, which define an algebra on relations.

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Figure 2.10 Result of $\sigma_{dept\_name = \text{"Physics"}}(instructor)$.

2.6.1 The Select Operation

The **select** operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ. Thus, to select those tuples of the *instructor* relation where the instructor is in the "Physics" department, we write:

$$\sigma_{dept\_name = \text{"Physics"}}(instructor)$$

If the *instructor* relation is as shown in Figure 2.1, then the relation that results from the preceding query is as shown in Figure 2.10.

We can find all instructors with salary greater than \$90,000 by writing:

$$\sigma_{salary > 90000}(instructor)$$

In general, we allow comparisons using $=$, $\neq$, $<$, $\leq$, $>$, and $\geq$ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives *and* ($\wedge$), *or* ($\vee$), and *not* ($\neg$). Thus, to find the instructors in Physics with a salary greater than \$90,000, we write:

$$\sigma_{dept\_name = \text{"Physics"} \wedge salary > 90000}(instructor)$$

The selection predicate may include comparisons between two attributes. To illustrate, consider the relation *department*. To find all departments whose name is the same as their building name, we can write:

$$\sigma_{dept\_name = building}(department)$$

2.6.2 The Project Operation

Suppose we want to list all instructors' *ID*, *name*, and *salary*, but we do not care about the *dept_name*. The **project** operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated. Projection is denoted by the uppercase Greek letter pi (Π). We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses. We write the query to produce such a list as:

$$\Pi_{ID,\ name,\ salary}(instructor)$$

Figure 2.11 shows the relation that results from this query.

ID	*name*	*salary*
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

Figure 2.11 Result of $\Pi_{ID,\ name,\ salary}(instructor)$.

The basic version of the project operator $\Pi_L(E)$ allows only attribute names to be present in the list L. A generalized version of the operator allows expressions involving attributes to appear in the list L. For example, we could use:

$$\Pi_{ID,name,salary/12}(instructor)$$

to get the monthly salary of each instructor.

2.6.3 Composition of Relational Operations

The fact that the result of a relational operation is itself a relation is important. Consider the more complicated query "Find the names of all instructors in the Physics department." We write:

$$\Pi_{name}\ (\sigma_{dept\_name = \text{"Physics"}}\ (instructor))$$

Notice that, instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

In general, since the result of a relational-algebra operation is of the same type (relation) as its inputs, relational-algebra operations can be composed together into a **relational-algebra expression**. Composing relational-algebra operations into relational-algebra expressions is just like composing arithmetic operations (such as +, −, ∗, and ÷) into arithmetic expressions.

2.6.4 The Cartesian-Product Operation

The **Cartesian-product** operation, denoted by a cross (×), allows us to combine information from any two relations. We write the Cartesian product of relations r_1 and r_2 as $r_1 \times r_2$.

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Figure 2.12 Result of the Cartesian product *instructor* × *teaches*.

A Cartesian product of database relations differs in its definition slightly from the mathematical definition of a Cartesian product of sets. Instead of $r_1 \times r_2$ producing pairs (t_1, t_2) of tuples from r_1 and r_2, the relational algebra concatenates t_1 and t_2 into a single tuple, as shown in Figure 2.12.

Since the same attribute name may appear in the schemas of both r_1 and r_2, we need to devise a naming schema to distinguish between these attributes. We do so here by attaching to an attribute the name of the relation from which the attribute originally came. For example, the relation schema for $r = instructor \times teaches$ is:

(*instructor.ID*, *instructor.name*, *instructor.dept_name*, *instructor.salary*,
teaches.ID, *teaches.course_id*, *teaches.sec_id*, *teaches.semester*, *teaches.year*)

With this schema, we can distinguish *instructor.ID* from *teaches.ID*. For those attributes that appear in only one of the two schemas, we shall usually drop the relation-name prefix. This simplification does not lead to any ambiguity. We can then write the relation schema for r as:

(*instructor.ID*, *name*, *dept_name*, *salary*,
teaches.ID, *course_id*, *sec_id*, *semester*, *year*)

This naming convention *requires* that the relations that are the arguments of the Cartesian-product operation have distinct names. This requirement causes problems in some cases, such as when the Cartesian product of a relation with itself is desired. A similar problem arises if we use the result of a relational-algebra expression in a Cartesian product, since we shall need a name for the relation so that we can refer to the relation's attributes. In Section 2.6.8, we see how to avoid these problems by using the rename operation.

Now that we know the relation schema for $r = instructor \times teaches$, what tuples appear in r? As you may suspect, we construct a tuple of r out of each possible pair of tuples: one from the *instructor* relation (Figure 2.1) and one from the *teaches* relation (Figure 2.7). Thus, r is a large relation, as you can see from Figure 2.12, which includes only a portion of the tuples that make up r.

Assume that we have n_1 tuples in *instructor* and n_2 tuples in *teaches*. Then, there are $n_1 * n_2$ ways of choosing a pair of tuples—one tuple from each relation; so there are $n_1 * n_2$ tuples in r. In particular for our example, for some tuples t in r, it may be that the two ID values, *instructor.ID* and *teaches.ID*, are different.

In general, if we have relations $r_1(R_1)$ and $r_2(R_2)$, then $r_1 \times r_2$ is a relation $r(R)$ whose schema R is the concatenation of the schemas R_1 and R_2. Relation r contains all tuples t for which there is a tuple t_1 in r_1 and a tuple t_2 in r_2 for which t and t_1 have the same value on the attributes in R_1 and t and t_2 have the same value on the attributes in R_2.

2.6.5 The Join Operation

Suppose we want to find the information about all instructors together with the *course_id* of all courses they have taught. We need the information in both the *instructor* relation and the *teaches* relation to compute the required result. The Cartesian product of *instructor* and *teaches* does bring together information from both these relations, but unfortunately the Cartesian product associates every instructor with every course that was taught, regardless of whether that instructor taught that course.

Since the Cartesian-product operation associates *every* tuple of *instructor* with every tuple of *teaches*, we know that if an instructor has taught a course (as recorded in the *teaches* relation), then there is some tuple in $instructor \times teaches$ that contains her name and satisfies $instructor.ID = teaches.ID$. So, if we write:

$$\sigma_{instructor.ID = teaches.ID}(instructor \times teaches)$$

we get only those tuples of $instructor \times teaches$ that pertain to instructors and the courses that they taught.

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018
98345	Kim	Elec. Eng.	80000	98345	EE-181	1	Spring	2017

Figure 2.13 Result of $\sigma_{instructor.ID = teaches.ID}(instructor \times teaches)$.

The result of this expression is shown in Figure 2.13. Observe that instructors Gold, Califieri, and Singh do not teach any course (as recorded in the *teaches* relation), and therefore do not appear in the result.

Note that this expression results in the duplication of the instructor's ID. This can be easily handled by adding a projection to eliminate the column *teaches.ID*.

The *join* operation allows us to combine a selection and a Cartesian product into a single operation.

Consider relations $r(R)$ and $s(S)$, and let θ be a predicate on attributes in the schema $R \cup S$. The **join** operation $r \bowtie_\theta s$ is defined as follows:

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

Thus, $\sigma_{instructor.ID = teaches.ID}(instructor \times teaches)$ can equivalently be written as $instructor \bowtie_{instructor.ID = teaches.ID} teaches$.

2.6.6 Set Operations

Consider a query to find the set of all courses taught in the Fall 2017 semester, the Spring 2018 semester, or both. The information is contained in the *section* relation (Figure 2.6). To find the set of all courses taught in the Fall 2017 semester, we write:

$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Fall”} \wedge year = 2017}\ (section))$$

To find the set of all courses taught in the Spring 2018 semester, we write:

$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Spring”} \wedge year = 2018}\ (section))$$

To answer the query, we need the **union** of these two sets; that is, we need all *course_id*s that appear in either or both of the two relations. We find these data by the binary operation union, denoted, as in set theory, by ∪. So the expression needed is:

$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Fall”} \wedge year=2017}\ (section)) \cup$$
$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Spring”} \wedge year=2018}\ (section))$$

The result relation for this query appears in Figure 2.14. Notice that there are eight tuples in the result, even though there are three distinct courses offered in the Fall 2017 semester and six distinct courses offered in the Spring 2018 semester. Since relations are sets, duplicate values such as CS-101, which is offered in both semesters, are replaced by a single occurrence.

Observe that, in our example, we took the union of two sets, both of which consisted of *course_id* values. In general, for a union operation to make sense:

1. We must ensure that the input relations to the union operation have the same number of attributes; the number of attributes of a relation is referred to as its **arity**.
2. When the attributes have associated types, the types of the *i*th attributes of both input relations must be the same, for each *i*.

Such relations are referred to as **compatible** relations.

For example, it would not make sense to take the union of the *instructor* and *section* relations, since they have different numbers of attributes. And even though the *instructor* and the *student* relations both have arity 4, their 4th attributes, namely, *salary* and *tot_cred*, are of two different types. The union of these two attributes would not make sense in most situations.

The **intersection** operation, denoted by ∩, allows us to find tuples that are in both the input relations. The expression $r \cap s$ produces a relation containing those tuples in

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 2.14 Courses offered in either Fall 2017, Spring 2018, or both semesters.

course_id
CS-101

Figure 2.15 Courses offered in both the Fall 2017 and Spring 2018 semesters.

r as well as in s. As with the union operation, we must ensure that intersection is done between compatible relations.

Suppose that we wish to find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters. Using set intersection, we can write

$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Fall”} \wedge year=2017}\ (section)) \cap \Pi_{course\_id}\ (\sigma_{semester = \text{“Spring”} \wedge year=2018}\ (section))$$

The result relation for this query appears in Figure 2.15.

The **set-difference** operation, denoted by $-$, allows us to find tuples that are in one relation but are not in another. The expression $r - s$ produces a relation containing those tuples in r but not in s.

We can find all the courses taught in the Fall 2017 semester but not in Spring 2018 semester by writing:

$$\Pi_{course\_id}\ (\sigma_{semester = \text{“Fall”} \wedge year=2017}\ (section)) - \Pi_{course\_id}\ (\sigma_{semester = \text{“Spring”} \wedge year=2018}\ (section))$$

The result relation for this query appears in Figure 2.16.

As with the union operation, we must ensure that set differences are taken between compatible relations.

2.6.7 The Assignment Operation

It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables. The **assignment** operation, denoted by $\leftarrow$, works like assignment in a programming language. To illustrate this operation, consider the query to find courses that run in Fall 2017 as well as Spring 2018, which we saw earlier. We could write it as:

course_id
CS-347
PHY-101

Figure 2.16 Courses offered in the Fall 2017 semester but not in Spring 2018 semester.

$$courses\_fall\_2017 \leftarrow \Pi_{course\_id}(\sigma_{semester = \text{“Fall”} \wedge year=2017}(section))$$
$$courses\_spring\_2018 \leftarrow \Pi_{course\_id}(\sigma_{semester = \text{“Spring”} \wedge year=2018}(section))$$
$$courses\_fall\_2017 \cap courses\_spring\_2018$$

The final line above displays the query result. The preceding two lines assign the query result to a temporary relation. The evaluation of an assignment does not result in any relation being displayed to the user. Rather, the result of the expression to the right of the $\leftarrow$ is assigned to the relation variable on the left of the $\leftarrow$. This relation variable may be used in subsequent expressions.

With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query. For relational-algebra queries, assignment must always be made to a temporary relation variable. Assignments to permanent relations constitute a database modification. Note that the assignment operation does not provide any additional power to the algebra. It is, however, a convenient way to express complex queries.

2.6.8 The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful in some cases to give them names; the **rename** operator, denoted by the lowercase Greek letter rho (ρ), lets us do this. Given a relational-algebra expression E, the expression

$$\rho_x(E)$$

returns the result of expression E under the name x.

A relation r by itself is considered a (trivial) relational-algebra expression. Thus, we can also apply the rename operation to a relation r to get the same relation under a new name. Some queries require the same relation to be used more than once in the query; in such cases, the rename operation can be used to give unique names to the different occurrences of the same relation.

A second form of the rename operation is as follows: Assume that a relational-algebra expression E has arity n. Then, the expression

$$\rho_{x(A_1,A_2,\ldots,A_n)}(E)$$

returns the result of expression E under the name x, and with the attributes renamed to $A_1, A_2, \ldots, A_n$. This form of the rename operation can be used to give names to attributes in the results of relational algebra operations that involve expressions on attributes.

To illustrate renaming a relation, we consider the query “Find the ID and name of those instructors who earn more than the instructor whose ID is 12121.” (That’s the instructor Wu in the example table in Figure 2.1.)

There are several strategies for writing this query, but to illustrate the rename operation, our strategy is to compare the salary of each instructor with the salary of the

Note 2.1 OTHER RELATIONAL OPERATIONS

In addition to the relational algebra operations we have seen so far, there are a number of other operations that are commonly used. We summarize them below and describe them in detail later, along with equivalent SQL constructs.

The aggregation operation allows a function to be computed over the set of values returned by a query. These functions include average, sum, min, and max, among others. The operation allows also for these aggregations to be performed after splitting the set of values into groups, for example, by computing the average salary in each department. We study the aggregation operation in more detail in Section 3.7 (Note 3.2 on page 97).

The *natural join* operation replaces the predicate θ in $\bowtie_\theta$ with an implicit predicate that requires equality over those attributes that appear in the schemas of both the left and right relations. This is notationally convenient but poses risks for queries that are reused and thus might be used after a relation's schema is changed. It is covered in Section 4.1.1.

Recall that when we computed the join of *instructor* and *teaches*, instructors who have not taught any course do not appear in the join result. The *outer join* operation allows for the retention of such tuples in the result by inserting nulls for the missing values. It is covered in Section 4.1.3 (Note 4.1 on page 136).

instructor with ID 12121. The difficulty here is that we need to reference the *instructor* relation once to get the salary of each instructor and then a second time to get the salary of instructor 12121; and we want to do all this in one expression. The rename operator allows us to do this using different names for each referencing of the *instructor* relation. In this example, we shall use the name *i* to refer to our scan of the *instructor* relation in which we are seeking those that will be part of the answer, and *w* to refer to the scan of the *instructor* relation to obtain the salary of instructor 12121:

$$\Pi_{i.ID,i.name}\ ((\sigma_{i.salary\ >\ w.salary}(\rho_i(instructor)\ \times\ \sigma_{w.id=12121}(\rho_w\ (instructor)))))$$

The rename operation is not strictly required, since it is possible to use a positional notation for attributes. We can name attributes of a relation implicitly by using a positional notation, where \$1, \$2, … refer to the first attribute, the second attribute, and so on. The positional notation can also be used to refer to attributes of the results of relational-algebra operations. However, the positional notation is inconvenient for humans, since the position of the attribute is a number, rather than an easy-to-remember attribute name. Hence, we do not use the positional notation in this textbook.

2.6.9 Equivalent Queries

Note that there is often more than one way to write a query in relational algebra. Consider the following query, which finds information about courses taught by instructors in the Physics department:

$$\sigma_{dept\_name = \text{“Physics”}}(instructor \bowtie_{instructor.ID = teaches.ID} teaches)$$

Now consider an alternative query:

$$(\sigma_{dept\_name = \text{“Physics”}}(instructor)) \bowtie_{instructor.ID = teaches.ID} teaches$$

Note the subtle difference between the two queries: in the first query, the selection that restricts *dept_name* to Physics is applied after the join of *instructor* and *teaches* has been computed, whereas in the second query, the selection that restricts *dept_name* to Physics is applied to *instructor*, and the join operation is applied subsequently.

Although the two queries are not identical, they are in fact **equivalent**; that is, they give the same result on any database.

Query optimizers in database systems typically look at what result an expression computes and find an efficient way of computing that result, rather than following the exact sequence of steps specified in the query. The algebraic structure of relational algebra makes it easy to find efficient but equivalent alternative expressions, as we will see in Chapter 16.

2.7 Summary

- The relational data model is based on a collection of tables. The user of the database system may query these tables, insert new tuples, delete tuples, and update (modify) tuples. There are several languages for expressing these operations.
- The schema of a relation refers to its logical design, while an instance of the relation refers to its contents at a point in time. The schema of a database and an instance of a database are similarly defined. The schema of a relation includes its attributes, and optionally the types of the attributes and constraints on the relation such as primary and foreign-key constraints.
- A superkey of a relation is a set of one or more attributes whose values are guaranteed to identify tuples in the relation uniquely. A candidate key is a minimal superkey, that is, a set of attributes that forms a superkey, but none of whose subsets is a superkey. One of the candidate keys of a relation is chosen as its primary key.
- A foreign-key constraint from attribute(s) A of relation r_1 to the primary-key B of relation r_2 states that the value of A for each tuple in r_1 must also be the value of B for some tuple in r_2. The relation r_1 is called the referencing relation, and r_2 is called the referenced relation.

- A schema diagram is a pictorial depiction of the schema of a database that shows the relations in the database, their attributes, and primary keys and foreign keys.
- The relational query languages define a set of operations that operate on tables and output tables as their results. These operations can be combined to get expressions that express desired queries.
- The relational algebra provides a set of operations that take one or more relations as input and return a relation as an output. Practical query languages such as SQL are based on the relational algebra, but they add a number of useful syntactic features.
- The relational algebra defines a set of algebraic operations that operate on tables, and output tables as their results. These operations can be combined to get expressions that express desired queries. The algebra defines the basic operations used within relational query languages like SQL.

Review Terms

- Table
- Relation
- Tuple
- Attribute
- Relation instance
- Domain
- Atomic domain
- Null value
- Database schema
- Database instance
- Relation schema
- Keys
  - Superkey
  - Candidate key
  - Primary key
  - Primary key constraints
- Foreign-key constraint
  - Referencing relation
  - Referenced relation
- Referential integrity constraint
- Schema diagram
- Query language types
  - Imperative
  - Functional
  - Declarative
- Relational algebra
- Relational-algebra expression
- Relational-algebra operations
  - Select σ
  - Project Π
  - Cartesian product $\times$
  - Join $\bowtie$
  - Union $\cup$
  - Set difference $-$
  - Set intersection $\cap$
  - Assignment $\leftarrow$
  - Rename ρ

employee (*person_name*, *street*, *city*)
works (*person_name*, *company_name*, *salary*)
company (*company_name*, *city*)

Figure 2.17 Employee database.

Practice Exercises

2.1 Consider the employee database of Figure 2.17. What are the appropriate primary keys?

2.2 Consider the foreign-key constraint from the *dept_name* attribute of *instructor* to the *department* relation. Give examples of inserts and deletes to these relations that can cause a violation of the foreign-key constraint.

2.3 Consider the *time_slot* relation. Given that a particular time slot can meet more than once in a week, explain why *day* and *start_time* are part of the primary key of this relation, while *end_time* is not.

2.4 In the instance of *instructor* shown in Figure 2.1, no two instructors have the same name. From this, can we conclude that *name* can be used as a superkey (or primary key) of *instructor*?

2.5 What is the result of first performing the Cartesian product of *student* and *advisor*, and then performing a selection operation on the result with the predicate *s_id* = ID? (Using the symbolic notation of relational algebra, this query can be written as $\sigma_{s\_id=ID}(student \times advisor)$.)

2.6 Consider the employee database of Figure 2.17. Give an expression in the relational algebra to express each of the following queries:

a. Find the name of each employee who lives in city "Miami".

b. Find the name of each employee whose salary is greater than $100000.

c. Find the name of each employee who lives in "Miami" and whose salary is greater than $100000.

2.7 Consider the bank database of Figure 2.18. Give an expression in the relational algebra for each of the following queries:

a. Find the name of each branch located in "Chicago".

b. Find the ID of each borrower who has a loan in branch "Downtown".

branch(*branch_name*, *branch_city, assets*)
customer (*ID*, *customer_name*, *customer_street, customer_city*)
loan (*loan_number*, *branch_name, amount*)
borrower (*ID*, *loan_number*)
account (*account_number*, *branch_name, balance*)
depositor (*ID*, *account_number*)

Figure 2.18 Bank database.

2.8 Consider the employee database of Figure 2.17. Give an expression in the relational algebra to express each of the following queries:

a. Find the ID and name of each employee who does not work for "BigBank".

b. Find the ID and name of each employee who earns at least as much as every employee in the database.

2.9 The **division operator** of relational algebra, "÷", is defined as follows. Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$; that is, every attribute of schema S is also in schema R. Given a tuple t, let $t[S]$ denote the projection of tuple t on the attributes in S. Then $r \div s$ is a relation on schema $R - S$ (that is, on the schema containing all attributes of schema R that are not in schema S). A tuple t is in $r \div s$ if and only if both of two conditions hold:

- t is in $\Pi_{R-S}(r)$
- For every tuple t_s in s, there is a tuple t_r in r satisfying both of the following:

  a. $t_r[S] = t_s[S]$

  b. $t_r[R - S] = t$

Given the above definition:

a. Write a relational algebra expression using the division operator to find the IDs of all students who have taken all Comp. Sci. courses. (Hint: project *takes* to just ID and *course_id*, and generate the set of all Comp. Sci. *course_id*s using a select expression, before doing the division.)

b. Show how to write the above query in relational algebra, without using division. (By doing so, you would have shown how to define the division operation using the other relational algebra operations.)

Exercises

2.10 Describe the differences in meaning between the terms *relation* and *relation schema*.

2.11 Consider the *advisor* relation shown in the schema diagram in Figure 2.9, with *s_id* as the primary key of *advisor*. Suppose a student can have more than one advisor. Then, would *s_id* still be a primary key of the *advisor* relation? If not, what should the primary key of *advisor* be?

2.12 Consider the bank database of Figure 2.18. Assume that branch names and customer names uniquely identify branches and customers, but loans and accounts can be associated with more than one customer.

a. What are the appropriate primary keys?

b. Given your choice of primary keys, identify appropriate foreign keys.

2.13 Construct a schema diagram for the bank database of Figure 2.18.

2.14 Consider the employee database of Figure 2.17. Give an expression in the relational algebra to express each of the following queries:

a. Find the ID and name of each employee who works for "BigBank".

b. Find the ID, name, and city of residence of each employee who works for "BigBank".

c. Find the ID, name, street address, and city of residence of each employee who works for "BigBank" and earns more than $10000.

d. Find the ID and name of each employee in this database who lives in the same city as the company for which she or he works.

2.15 Consider the bank database of Figure 2.18. Give an expression in the relational algebra for each of the following queries:

a. Find each loan number with a loan amount greater than $10000.

b. Find the ID of each depositor who has an account with a balance greater than $6000.

c. Find the ID of each depositor who has an account with a balance greater than $6000 at the "Uptown" branch.

2.16 List two reasons why null values might be introduced into a database.

2.17 Discuss the relative merits of imperative, functional, and declarative languages.

2.18 Write the following queries in relational algebra, using the university schema.

a. Find the ID and name of each instructor in the Physics department.

b. Find the ID and name of each instructor in a department located in the building "Watson".

c. Find the ID and name of each student who has taken at least one course in the "Comp. Sci." department.

d. Find the ID and name of each student who has taken at least one course section in the year 2018.

e. Find the ID and name of each student who has not taken any course section in the year 2018.

Further Reading

E. F. Codd of the IBM San Jose Research Laboratory proposed the relational model in the late 1960s ([Codd (1970)]). In that paper, Codd also introduced the original definition of relational algebra. This work led to the prestigious ACM Turing Award to Codd in 1981 ([Codd (1982)]).

After E. F. Codd introduced the relational model, an expansive theory developed around the relational model pertaining to schema design and the expressive power of various relational languages. Several classic texts cover relational database theory, including [Maier (1983)] (which is available free, online), and [Abiteboul et al. (1995)].

Codd's original paper inspired several research projects that were formed in the mid to late 1970s with the goal of constructing practical relational database systems, including System R at the IBM San Jose Research Laboratory, Ingres at the University of California at Berkeley, and Query-by-Example at the IBM T. J. Watson Research Center. The Oracle database was developed commercially at the same time.

Many relational database products are now commercially available. These include IBM's DB2 and Informix, Oracle, Microsoft SQL Server, and Sybase and HANA from SAP. Popular open-source relational database systems include MySQL and PostgreSQL. Hive and Spark are widely used systems that support parallel execution of queries across large numbers of computers.

Bibliography

[Abiteboul et al. (1995)] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison Wesley (1995).

[Codd (1970)] E. F. Codd, "A Relational Model for Large Shared Data Banks", *Communications of the ACM*, Volume 13, Number 6 (1970), pages 377–387.

[Codd (1982)] E. F. Codd, "The 1981 ACM Turing Award Lecture: Relational Database: A Practical Foundation for Productivity", *Communications of the ACM*, Volume 25, Number 2 (1982), pages 109–117.

[Maier (1983)] D. Maier, *The Theory of Relational Databases*, Computer Science Press (1983).

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.