

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:

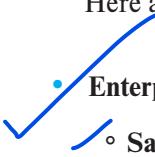
- are highly valuable,
- are relatively large, and
- are accessed by multiple users and applications, often at the same time.

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 various 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 independence**.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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

<i>dept_name</i>	<i>building</i>	<i>budget</i>
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

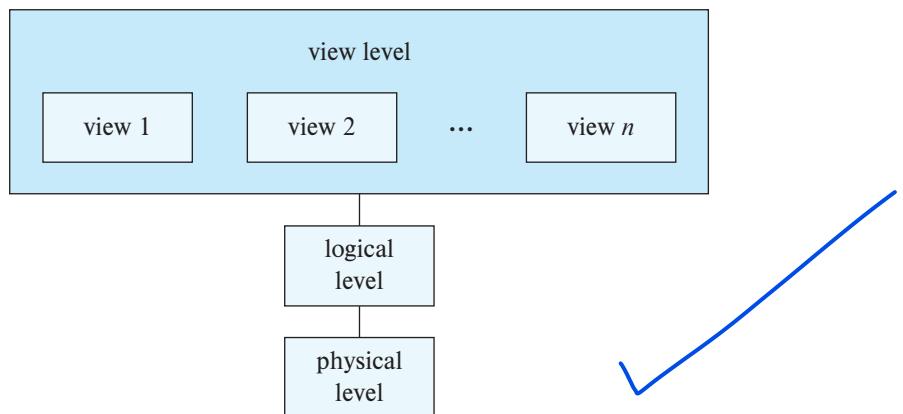


Figure 1.2 The three levels of data abstraction.

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*.

¹The 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

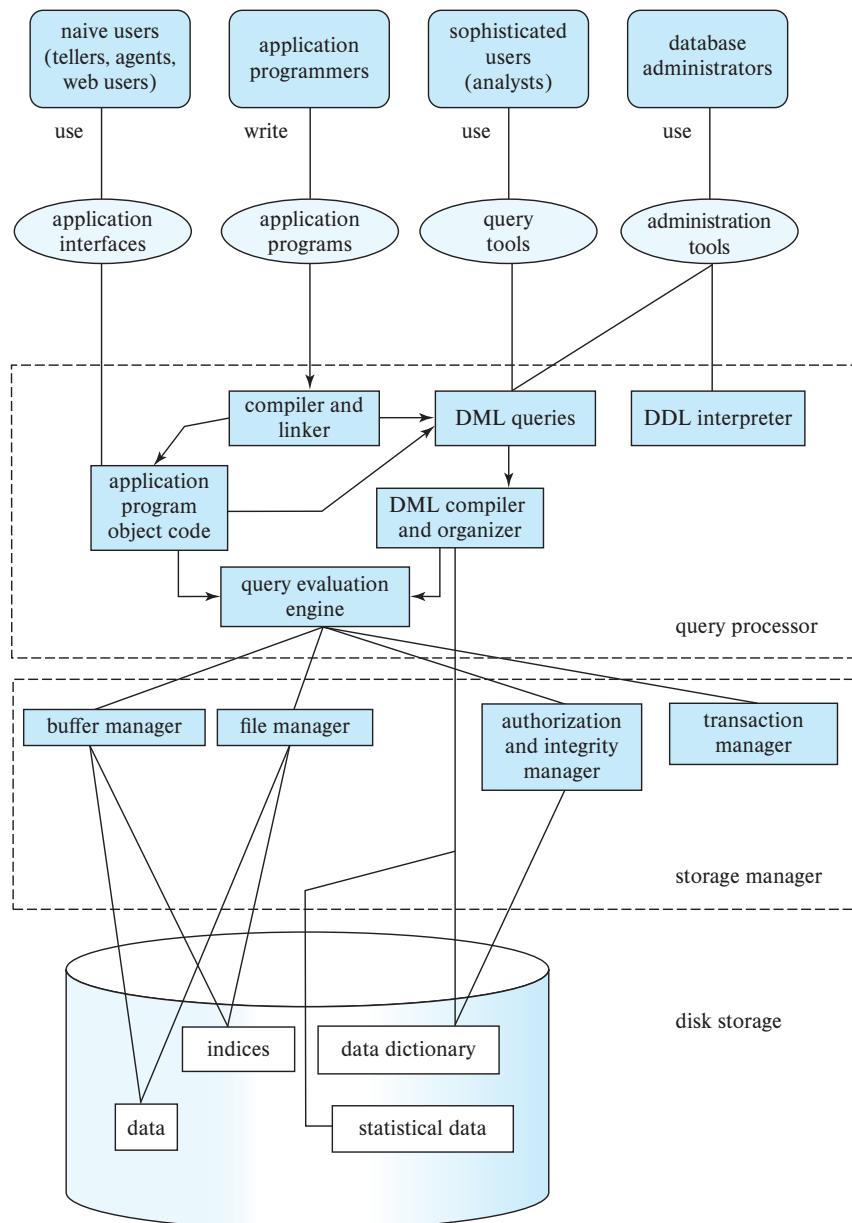


Figure 1.3 System structure.

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 backend. 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.

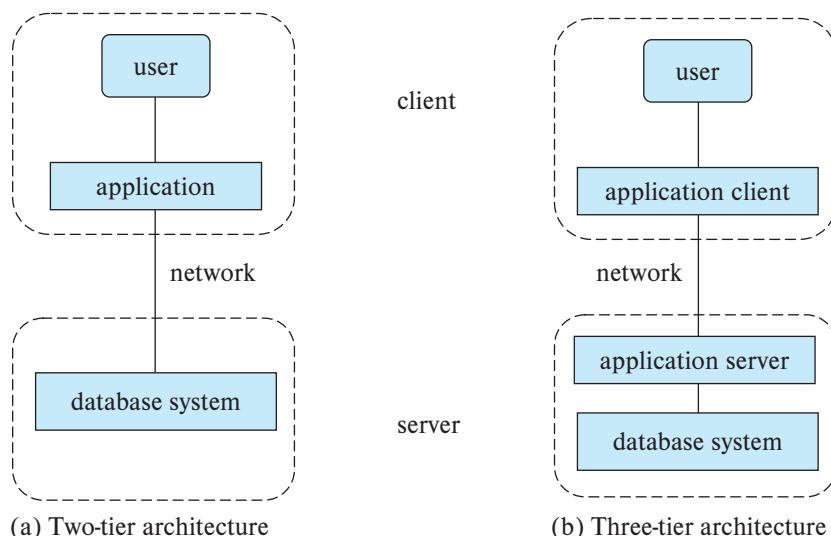


Figure 1.4 Two-tier and three-tier architectures.

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 non-procedural 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. RÃ©, 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.

CHAPTER 1



Introduction

Practice Exercises

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

Answer:

Two disadvantages associated with database systems are listed below.

- a. Setup of the database system requires more knowledge, money, skills, and time.
- b. The complexity of the database may result in poor performance.

- 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.

Answer:

- a. Executing an action in the DDL results in the creation of an object in the database; in contrast, a programming language type declaration is simply an abstraction used in the program.
- b. Database DDLs allow consistency constraints to be specified, which programming language type systems generally do not allow. These include domain constraints and referential integrity constraints.
- c. Database DDLs support authorization, giving different access rights to different users. Programming language type systems do not provide such protection (at best, they protect attributes in a class from being accessed by methods in another class).
- d. Programming language type systems are usually much richer than the SQL type system. Most databases support only basic types such as different types of numbers and strings, although some databases do support some complex types such as arrays and objects.

- e. A database DDL is focused on specifying types of attributes of relations; in contrast, a programming language allows objects and collections of objects to be created.

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

Answer:

Six major steps in setting up a database for a particular enterprise are:

- Define the high-level requirements of the enterprise (this step generates a document known as the system requirements specification.)
- Define a model containing all appropriate types of data and data relationships.
- Define the integrity constraints on the data.
- Define the physical level.
- For each known problem to be solved on a regular basis (e.g., tasks to be carried out by clerks or web users), define a user interface to carry out the task, and write the necessary application programs to implement the user interface.
- Create/initialize the database.

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.

Answer:

- **Data redundancy and inconsistency.** This would be relevant to metadata to some extent, although not to the actual video data, which are not updated. There are very few relationships here, and none of them can lead to redundancy.
- **Difficulty in accessing data.** If video data are only accessed through a few predefined interfaces, as is done in video sharing sites today, this will not be a problem. However, if an organization needs to find video data based on specific search conditions (beyond simple keyword queries), if metadata were stored in files it would be hard to find relevant data without writing application programs. Using a database would be important for the task of finding data.
- **Data isolation.** Since data are not usually updated, but instead newly created, data isolation is not a major issue. Even the task of keeping track of

who has viewed what videos is (conceptually) append only, again making isolation not a major issue. However, if authorization is added, there may be some issues of concurrent updates to authorization information.

- **Integrity problems.** It seems unlikely there are significant integrity constraints in this application, except for primary keys. If the data are distributed, there may be issues in enforcing primary key constraints. Integrity problems are probably not a major issue.
- **Atomicity problems.** When a video is uploaded, metadata about the video and the video should be added atomically, otherwise there would be an inconsistency in the data. An underlying recovery mechanism would be required to ensure atomicity in the event of failures.
- **Concurrent-access anomalies.** Since data are not updated, concurrent access anomalies would be unlikely to occur.
- **Security problems.** These would be an issue if the system supported authorization.

- 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.

Answer:

Queries used in the web are specified by providing a list of keywords with no specific syntax. The result is typically an ordered list of URLs, along with snippets of information about the content of the URLs. In contrast, database queries have a specific syntax allowing complex queries to be specified. And in the relational world the result of a query is always a table.

Databases and Database Users

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

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

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

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

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

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

1.1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.2 An Example

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

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

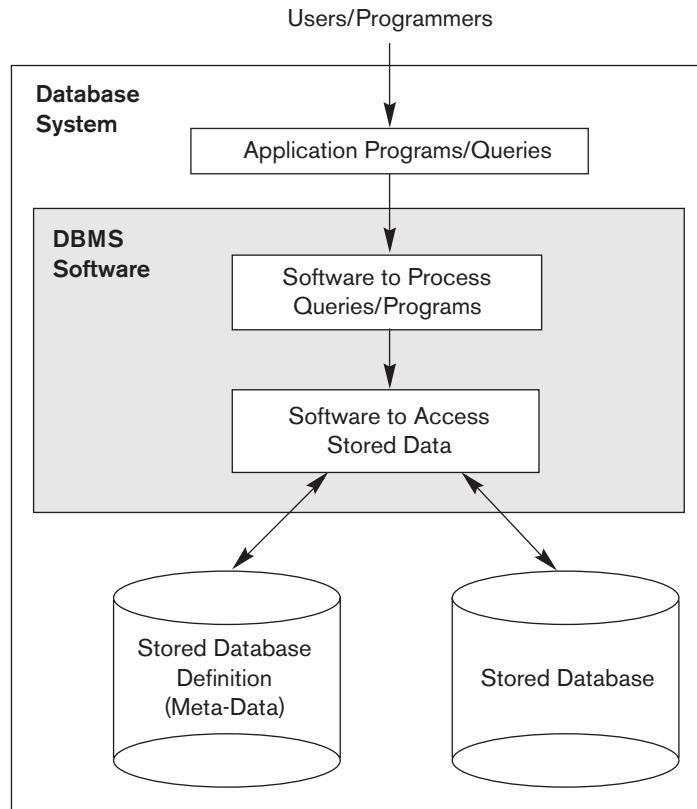


Figure 1.1
A simplified database system environment.

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

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

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

STUDENT

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

COURSE

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

SECTION

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

GRADE_REPORT

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

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2

A database that stores student and course information.

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

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

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

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

Examples of updates include the following:

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

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

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

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

1.3 Characteristics of the Database Approach

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

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

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

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

1.3.1 Self-Describing Nature of a Database System

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

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

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

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

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

RELATIONS

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

Figure 1.3

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

COLUMNS

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

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

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

1.3.2 Insulation between Programs and Data, and Data Abstraction

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

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

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

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

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

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

Figure 1.4

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

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

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

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

1.3.3 Support of Multiple Views of the Data

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

1.3.4 Sharing of Data and Multiuser Transaction Processing

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

TRANSCRIPT

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

(a)

COURSE_PREREQUISITES

Course_name	Course_number	Prerequisites
Database	CS3380	CS3320
		MATH2410
Data Structures	CS3320	CS1310

(b)

Figure 1.5

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

(b) The COURSE_PREREQUISITES view.

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

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

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

1.4 Actors on the Scene

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

1.4.1 Database Administrators

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

1.4.2 Database Designers

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

1.4.3 End Users

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

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

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

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

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

1.4.4 **System Analysts and Application Programmers (Software Engineers)**

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

1.5 Workers behind the Scene

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

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

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

1.6 Advantages of Using the DBMS Approach

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

1.6.1 Controlling Redundancy

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

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

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

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

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

Figure 1.6

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

GRADE_REPORT					
Student_number	Student_name	Section_identifier	Course_number	Grade	
17	Smith	112	MATH2410	B	
17	Smith	119	CS1310	C	
8	Brown	85	MATH2410	A	
8	Brown	92	CS1310	A	
8	Brown	102	CS3320	B	
8	Brown	135	CS3380	A	

(a)

GRADE_REPORT					
Student_number	Student_name	Section_identifier	Course_number	Grade	
17	Brown	112	MATH2410	B	

(b)

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

1.6.2 Restricting Unauthorized Access

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

1.6.3 Providing Persistent Storage for Program Objects

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

The persistent storage of program objects and data structures is an important function of database systems. Traditional database systems often suffered from the so-called **impedance mismatch problem**, since the data structures provided by the DBMS were incompatible with the programming language's data structures.

Object-oriented database systems typically offer data structure compatibility with one or more object-oriented programming languages.

1.6.4 Providing Storage Structures and Search Techniques for Efficient Query Processing

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

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

1.6.5 Providing Backup and Recovery

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

1.6.6 Providing Multiple User Interfaces

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

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

1.6.7 Representing Complex Relationships among Data

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

1.6.8 Enforcing Integrity Constraints

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

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

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

1.6.9 Permitting Inferencing and Actions Using Rules and Triggers

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

1.6.10 Additional Implications of Using the Database Approach

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

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

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

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

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

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

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

1.7 A Brief History of Database Applications

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

1.7.1 Early Database Applications Using Hierarchical and Network Systems

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

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

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

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

1.7.2 Providing Data Abstraction and Application Flexibility with Relational Databases

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

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

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

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

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

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

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

1.7.5 Extending Database Capabilities for New Applications

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

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

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

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

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

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

1.7.6 Emergence of Big Data Storage Systems and NOSQL Databases

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

1.8 When Not to Use a DBMS

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

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

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

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

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

1.9 Summary

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

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

Review Questions

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

Exercises

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

1.14. Consider Figure 1.2.

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

Selected Bibliography

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



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*.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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

<i>course_id</i>	<i>title</i>	<i>dept_name</i>	<i>credits</i>
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.

<i>course_id</i>	<i>prereq_id</i>
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.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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

<i>dept_name</i>	<i>building</i>	<i>budget</i>
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.

<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>building</i>	<i>room_number</i>	<i>time_slot_id</i>
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)*

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
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 $t_1.K \neq t_2.K$.

¹Commercial 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 (*building*, *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 (*time_slot_id*, *day*, *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(building, room_number, capacity)
department(dept_name, building, budget)
course(course_id, title, dept_name, credits)
instructor(ID, name, dept_name, salary)
section(course_id, sec_id, semester, year, building, room_number, time_slot_id)
teaches(ID, course_id, sec_id, semester, year)
student(ID, name, dept_name, tot_cred)
takes(ID, course_id, sec_id, semester, year, grade)
advisor(s_ID, i_ID)
time_slot(time_slot_id, day, start_time, end_time)
prereq(course_id, prereq_id)

```

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 constraint. 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.

²The 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.

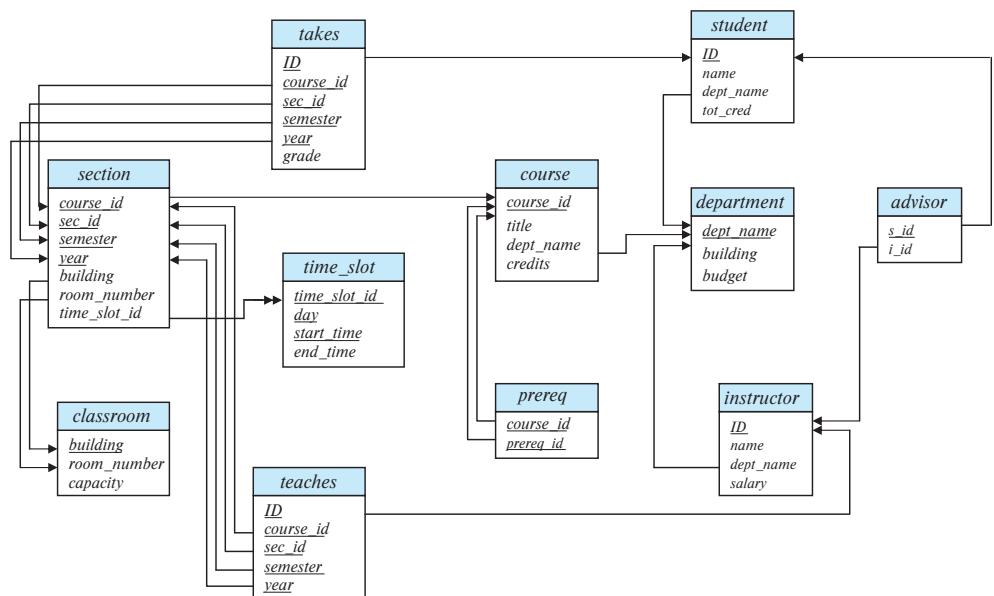


Figure 2.9 Schema diagram for the university database.

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.

³The 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.

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

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

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

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:

$$\sigma_{dept_name = "Physics"}(instructor)$$

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:

$$\sigma_{dept_name = "Physics" \wedge salary > 90000}(instructor)$$

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.

<i>ID</i>	<i>name</i>	<i>salary</i>
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:

$$\Pi_{name} (\sigma_{dept_name = "Physics"}(instructor))$$

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 \div) into arithmetic expressions.

2.6.4 The Cartesian-Product Operation

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

<i>instructor.ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
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 $\text{instructor} \times \text{teaches}$.

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 $r = \text{instructor} \times \text{teaches}$ is:

$$(\text{instructor.ID}, \text{instructor.name}, \text{instructor.dept_name}, \text{instructor.salary}, \\ \text{teaches.ID}, \text{teaches.course_id}, \text{teaches.sec_id}, \text{teaches.semester}, \text{teaches.year})$$

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 $r = \text{instructor} \times \text{teaches}$, 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:

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

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

<i>instructor.ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
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:

$$\Pi_{course_id} (\sigma_{semester = "Fall" \wedge year = 2017} (section))$$

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

$$\Pi_{course_id} (\sigma_{semester = "Spring" \wedge year = 2018} (section))$$

To answer the query, we need the **union** of these two sets; that is, we need all *course_ids* 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 \cup . So the expression needed is:

$$\begin{aligned} \Pi_{\text{course_id}} (\sigma_{\text{semester} = \text{"Fall"} \wedge \text{year} = 2017} (\text{section})) \cup \\ \Pi_{\text{course_id}} (\sigma_{\text{semester} = \text{"Spring"} \wedge \text{year} = 2018} (\text{section})) \end{aligned}$$

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

<i>course_id</i>
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

$$\Pi_{course_id} (\sigma_{semester = "Fall" \wedge year = 2017} (section)) \cap \Pi_{course_id} (\sigma_{semester = "Spring" \wedge year = 2018} (section))$$

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:

$$\Pi_{course_id} (\sigma_{semester = "Fall" \wedge year = 2017} (section)) - \Pi_{course_id} (\sigma_{semester = "Spring" \wedge year = 2018} (section))$$

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.

$$\begin{aligned}
 courses_fall_2017 &\leftarrow \Pi_{course_id}(\sigma_{semester} = "Fall" \wedge year = 2017) \text{ (section)} \\
 courses_spring_2018 &\leftarrow \Pi_{course_id}(\sigma_{semester} = "Spring" \wedge year = 2018) \text{ (section)} \\
 courses_fall_2017 \cap courses_spring_2018
 \end{aligned}$$

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

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

returns the result of expression E under the name x , and with the attributes renamed to A_1, A_2, \dots, 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_0 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:
- Find the ID and name of each employee who does not work for “BigBank”.
 - 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, “ \div ”, 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:
 - $t_r[S] = t_s[S]$
 - $t_r[R - S] = t$

Given the above definition:

- 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_ids* using a select expression, before doing the division.)
- 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.
- What are the appropriate primary keys?
 - 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:
- Find the ID and name of each employee who works for “BigBank”.
 - Find the ID, name, and city of residence of each employee who works for “BigBank”.
 - Find the ID, name, street address, and city of residence of each employee who works for “BigBank” and earns more than \$10000.
 - 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:
- Find each loan number with a loan amount greater than \$10000.
 - Find the ID of each depositor who has an account with a balance greater than \$6000.
 - 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.
- 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 Nešvadba/Shutterstock.

CHAPTER **2**



Introduction to the Relational Model

Practice Exercises

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

Answer:

The appropriate primary keys are shown below:

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

- 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.

Answer:

- Inserting a tuple:

(10111, Ostrom, Economics, 110000)

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

Figure 2.17 Employee database.

into the *instructor* table, where the *department* table does not have the department Economics, would violate the foreign-key constraint.

- Deleting the tuple:

(Biology, Watson, 90000)

from the *department* table, where at least one student or instructor tuple has *dept_name* as Biology, would violate 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.

Answer:

The attributes *day* and *start_time* are part of the primary key since a particular class will most likely meet on several different days and may even meet more than once in a day. However, *end_time* is not part of the primary key since a particular class that starts at a particular time on a particular day cannot end at more than one time.

- 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*?

Answer:

No. For this possible instance of the *instructor* table the names are unique, but in general this may not always be the case (unless the university has a rule that two instructors cannot have the same name, which is a rather unlikely scenario).

- 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 $\sigma_{s_id=ID}(student \times advisor)$.)

Answer:

The result attributes include all attribute values of *student* followed by all attributes of *advisor*. The tuples in the result are as follows: For each student who has an advisor, the result has a row containing that student's attributes, followed by an *s_id* attribute identical to the student's ID attribute, followed by the *i_id* attribute containing the ID of the student's advisor.

Students who do not have an advisor will not appear in the result. A student who has more than one advisor will appear a corresponding number of times in the result.

- 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".

$\underbrace{\text{branch}(\text{branch_name}, \text{branch_city}, \text{assets})}$
 $\text{customer}(\text{ID}, \text{customer_name}, \text{customer_street}, \text{customer_city})$
 $\underbrace{\text{loan}(\text{loan_number}, \text{branch_name}, \text{amount})}$
 $\underbrace{\text{borrower}(\text{ID}, \text{loan_number})}$
 $\text{account}(\text{account_number}, \text{branch_name}, \text{balance})$
 $\text{depositor}(\text{ID}, \text{account_number})$

Figure 2.18 Bank database.

- 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.

Answer:

- a. $\Pi_{\text{person_name}} (\sigma_{\text{city} = \text{“Miami”}} (\text{employee}))$
- b. $\Pi_{\text{person_name}} (\sigma_{\text{salary} > 100000} (\text{employee} \bowtie \text{works}))$
- c. $\Pi_{\text{person_name}} (\sigma_{\text{city} = \text{“Miami”} \wedge \text{salary} > 100000} (\text{employee} \bowtie \text{works}))$

- 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”.

Answer:

- a. $\Pi_{\text{branch_name}} (\sigma_{\text{branch_city} = \text{“Chicago”}} (\text{branch}))$
- b. $\Pi_{\text{ID}} (\sigma_{\text{branch_name} = \text{“Downtown”}} (\text{borrower} \bowtie_{\text{borrower.loan_number} = \text{loan.loan_number}} \text{loan}))$

- 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.

Answer:

- a. To find employees who do not work for BigBank, we first find all those who *do* work for BigBank. Those are exactly the employees *not* part of the

desired result. We then use set difference to find the set of all employees minus those employees that should not be in the result.

$$\begin{aligned} & \Pi_{ID,person_name}(employee) - \\ & \Pi_{ID,person_name} \\ & (employee \bowtie_{employee.ID=works.ID} (\sigma_{company_name='BigBank'}(works))) \end{aligned}$$

- b. We use the same approach as in part *a* by first finding those employees who do not earn the highest salary, or, said differently, for whom some other employee earns more. Since this involves comparing two employee salary values, we need to reference the *employee* relation twice and therefore use renaming.

$$\begin{aligned} & \Pi_{ID,person_name}(employee) - \\ & \Pi_{A.ID,A.person_name}(\rho_A(employee)) \bowtie_{A.salary < B.salary} \rho_B(employee)) \end{aligned}$$

- 2.9** The **division operator** of relational algebra, “ \div ”, 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_ids* 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.)

Answer:

- a. $\Pi_{ID}(\Pi_{ID,course_id}(takes) \div \Pi_{course_id}(\sigma_{dept_name='Comp. Sci.'}(course)))$
- b. The required expression is as follows:

$$\begin{aligned}
r &\leftarrow \Pi_{ID, course_id}(takes) \\
s &\leftarrow \Pi_{course_id}(\sigma_{dept_name='Comp. Sci.'}(course)) \\
\Pi_{ID}(takes) &- \Pi_{ID}((\Pi_{ID}(takes) \times s) - r)
\end{aligned}$$

In general, let $r(R)$ and $s(S)$ be given, with $S \subseteq R$. Then we can express the division operation using basic relational algebra operations as follows:

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

To see that this expression is true, we observe that $\Pi_{R-S}(r)$ gives us all tuples t that satisfy the first condition of the definition of division. The expression on the right side of the set difference operator

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

serves to eliminate those tuples that fail to satisfy the second condition of the definition of division. Let us see how it does so. Consider $\Pi_{R-S}(r) \times s$. This relation is on schema R , and pairs every tuple in $\Pi_{R-S}(r)$ with every tuple in s . The expression $\Pi_{R-S,S}(r)$ merely reorders the attributes of r .

Thus, $(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)$ gives us those pairs of tuples from $\Pi_{R-S}(r)$ and s that do not appear in r . If a tuple t_j is in

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

then there is some tuple t_s in s that does not combine with tuple t_j to form a tuple in r . Thus, t_j holds a value for attributes $R - S$ that does not appear in $r \div s$. It is these values that we eliminate from $\Pi_{R-S}(r)$.

CHAPTER 3



Introduction to SQL

In this chapter, as well as in Chapter 4 and Chapter 5, we study the most widely used database query language, SQL.

Although we refer to the SQL language as a “query language,” it can do much more than just query a database. It can define the structure of the data, modify data in the database, and specify security constraints.

It is not our intention to provide a complete users’ guide for SQL. Rather, we present SQL’s fundamental constructs and concepts. Individual implementations of SQL may differ in details or may support only a subset of the full language.

We strongly encourage you to try out the SQL queries that we describe here on an actual database. See the Tools section at the end of this chapter for tips on what database systems you could use, and how to create the schema, populate sample data, and execute your queries.

3.1

Overview of the SQL Query Language

IBM developed the original version of SQL, originally called Sequel, as part of the System R project in the early 1970s. The Sequel language has evolved since then, and its name has changed to SQL (Structured Query Language). Many products now support the SQL language. SQL has clearly established itself as *the* standard relational database language.

In 1986, the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) published an SQL standard, called SQL-86. ANSI published an extended standard for SQL, SQL-89, in 1989. The next version of the standard was SQL-92 standard, followed by SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, and most recently SQL:2016.

The SQL language has several parts:

- **Data-definition language (DDL).** The SQL DDL provides commands for defining relation schemas, deleting relations, and modifying relation schemas.

- **Data-manipulation language (DML).** The SQL DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- **Integrity.** The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy. Updates that violate integrity constraints are disallowed.
- **View definition.** The SQL DDL includes commands for defining views.
- **Transaction control.** SQL includes commands for specifying the beginning and end points of transactions.
- **Embedded SQL and dynamic SQL.** Embedded and dynamic SQL define how SQL statements can be embedded within general-purpose programming languages, such as C, C++, and Java.
- **Authorization.** The SQL DDL includes commands for specifying access rights to relations and views.

In this chapter, we present a survey of basic DML and the DDL features of SQL. Features described here have been part of the SQL standard since SQL-92.

In Chapter 4, we provide a more detailed coverage of the SQL query language, including (a) various join expressions, (b) views, (c) transactions, (d) integrity constraints, (e) type system, and (f) authorization.

In Chapter 5, we cover more advanced features of the SQL language, including (a) mechanisms to allow accessing SQL from a programming language, (b) SQL functions and procedures, (c) triggers, (d) recursive queries, (e) advanced aggregation features, and (f) several features designed for data analysis.

Although most SQL implementations support the standard features we describe here, there are differences between implementations. Most implementations support some nonstandard features while omitting support for some of the more advanced and more recent features. In case you find that some language features described here do not work on the database system that you use, consult the user manuals for your database system to find exactly what features it supports.

3.2

SQL Data Definition

The set of relations in a database are specified using a data-definition language (DDL). The SQL DDL allows specification of not only a set of relations, but also information about each relation, including:

- The schema for each relation.
- The types of values associated with each attribute.

- The integrity constraints.
- The set of indices to be maintained for each relation.
- The security and authorization information for each relation.
- The physical storage structure of each relation on disk.

We discuss here basic schema definition and basic types; we defer discussion of the other SQL DDL features to Chapter 4 and Chapter 5.

3.2.1 Basic Types

The SQL standard supports a variety of built-in types, including:

- **char(*n*)**: A fixed-length character string with user-specified length *n*. The full form, **character**, can be used instead.
- **varchar(*n*)**: A variable-length character string with user-specified maximum length *n*. The full form, **character varying**, is equivalent.
- **int**: An integer (a finite subset of the integers that is machine dependent). The full form, **integer**, is equivalent.
- **smallint**: A small integer (a machine-dependent subset of the integer type).
- **numeric(*p,d*)**: A fixed-point number with user-specified precision. The number consists of *p* digits (plus a sign), and *d* of the *p* digits are to the right of the decimal point. Thus, **numeric(3,1)** allows 44.5 to be stored exactly, but neither 444.5 nor 0.32 can be stored exactly in a field of this type.
- **real, double precision**: Floating-point and double-precision floating-point numbers with machine-dependent precision.
- **float(*n*)**: A floating-point number with precision of at least *n* digits.

Additional types are covered in Section 4.5.

Each type may include a special value called the **null** value. A null value indicates an absent value that may exist but be unknown or that may not exist at all. In certain cases, we may wish to prohibit null values from being entered, as we shall see shortly.

The **char** data type stores fixed-length strings. Consider, for example, an attribute *A* of type **char(10)**. If we stored a string “Avi” in this attribute, seven spaces are appended to the string to make it 10 characters long. In contrast, if attribute *B* were of type **varchar(10)**, and we stored “Avi” in attribute *B*, no spaces would be added. When comparing two values of type **char**, if they are of different lengths, extra spaces are automatically attached to the shorter one to make them the same size before comparison.

When comparing a **char** type with a **varchar** type, one may expect extra spaces to be added to the **varchar** type to make the lengths equal, before comparison; however, this may or may not be done, depending on the database system. As a result, even if

the same value “Avi” is stored in the attributes A and B above, a comparison $A=B$ may return false. We recommend you always use the **varchar** type instead of the **char** type to avoid these problems.

SQL also provides the **nvarchar** type to store multilingual data using the Unicode representation. However, many databases allow Unicode (in the UTF-8 representation) to be stored even in **varchar** types.

3.2.2 Basic Schema Definition

We define an SQL relation by using the **create table** command. The following command creates a relation *department* in the database:



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

The relation created above has three attributes, *dept_name*, which is a character string of maximum length 20, *building*, which is a character string of maximum length 15, and *budget*, which is a number with 12 digits in total, two of which are after the decimal point. The **create table** command also specifies that the *dept_name* attribute is the primary key of the *department* relation.

The general form of the **create table** command is:

```
create table r
  (A1 D1,
   A2 D2,
   ...,
   An Dn,
   <integrity-constraint1>,
   ...,
   <integrity-constraintk>);
```

where *r* is the name of the relation, each A_i is the name of an attribute in the schema of relation *r*, and D_i is the domain of attribute A_i ; that is, D_i specifies the type of attribute A_i along with optional constraints that restrict the set of allowed values for A_i .

The semicolon shown at the end of the **create table** statements, as well as at the end of other SQL statements later in this chapter, is optional in many SQL implementations.

SQL supports a number of different integrity constraints. In this section, we discuss only a few of them:

- **primary key** ($A_{j_1}, A_{j_2}, \dots, A_{j_m}$): The **primary-key** specification says that attributes $A_{j_1}, A_{j_2}, \dots, A_{j_m}$ form the primary key for the relation. The primary-key attributes

are required to be *nonnull* and *unique*; that is, no tuple can have a null value for a primary-key attribute, and no two tuples in the relation can be equal on all the primary-key attributes. Although the primary-key specification is optional, it is generally a good idea to specify a primary key for each relation.

 **foreign key** ($A_{k_1}, A_{k_2}, \dots, A_{k_n}$) **references** s : The **foreign key** specification says that the values of attributes ($A_{k_1}, A_{k_2}, \dots, A_{k_n}$) for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation s .

Figure 3.1 presents a partial SQL DDL definition of the university database we use in the text. The definition of the *course* table has a declaration “**foreign key** (*dept_name*) **references** *department*”. This foreign-key declaration specifies that for each course tuple, the department name specified in the tuple must exist in the primary key attribute (*dept_name*) of the *department* relation. Without this constraint, it is possible for a course to specify a nonexistent department name. Figure 3.1 also shows foreign-key constraints on tables *section*, *instructor* and *teaches*. Some database systems, including MySQL, require an alternative syntax, “**foreign key** (*dept_name*) **references** *department*(*dept_name*)”, where the referenced attributes in the referenced table are listed explicitly.

- **not null**: The **not null** constraint on an attribute specifies that the null value is not allowed for that attribute; in other words, the constraint excludes the null value from the domain of that attribute. For example, in Figure 3.1, the **not null** constraint on the *name* attribute of the *instructor* relation ensures that the name of an instructor cannot be null.

More details on the foreign-key constraint, as well as on other integrity constraints that the **create table** command may include, are provided later, in Section 4.4.

SQL prevents any update to the database that violates an integrity constraint. For example, if a newly inserted or modified tuple in a relation has null values for any primary-key attribute, or if the tuple has the same value on the primary-key attributes as does another tuple in the relation, SQL flags an error and prevents the update. Similarly, an insertion of a *course* tuple with a *dept_name* value that does not appear in the *department* relation would violate the foreign-key constraint on *course*, and SQL prevents such an insertion from taking place.

A newly created relation is empty initially. Inserting tuples into a relation, updating them, and deleting them are done by data manipulation statements **insert**, **update**, and **delete**, which are covered in Section 3.9.

To remove a relation from an SQL database, we use the **drop table** command. The **drop table** command deletes all information about the dropped relation from the database. The command

```
drop table  $r$ ;
```

is a more drastic action than

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

create table course
  (course_id      varchar (7),
   title          varchar (50),
   dept_name      varchar (20),
   credits        numeric (2,0),
   primary key (course_id),
   foreign key (dept_name) references department);

create table instructor
  (ID              varchar (5),
   name            varchar (20) not null,
   dept_name       varchar (20),
   salary          numeric (8,2),
   primary key (ID),
   foreign key (dept_name) references department);

create table section
  (course_id      varchar (8),
   sec_id          varchar (8),
   semester        varchar (6),
   year            numeric (4,0),
   building        varchar (15),
   room_number     varchar (7),
   time_slot_id    varchar (4),
   primary key (course_id, sec_id, semester, year),
   foreign key (course_id) references course);

create table teaches
  (ID              varchar (5),
   course_id       varchar (8),
   sec_id          varchar (8),
   semester        varchar (6),
   year            numeric (4,0),
   primary key (ID, course_id, sec_id, semester, year),
   foreign key (course_id, sec_id, semester, year) references section,
   foreign key (ID) references instructor);
```

Figure 3.1 SQL data definition for part of the university database.

```
delete from r;
```

The latter retains relation r , but deletes all tuples in r . The former deletes not only all tuples of r , but also the schema for r . After r is dropped, no tuples can be inserted into r unless it is re-created with the **create table** command.

We use the **alter table** command to add attributes to an existing relation. All tuples in the relation are assigned *null* as the value for the new attribute. The form of the **alter table** command is

```
alter table r add A D;
```

where r is the name of an existing relation, A is the name of the attribute to be added, and D is the type of the added attribute. We can drop attributes from a relation by the command

```
alter table r drop A;
```

where r is the name of an existing relation, and A is the name of an attribute of the relation. Many database systems do not support dropping of attributes, although they will allow an entire table to be dropped.

3.3 Basic Structure of SQL Queries

The basic structure of an SQL query consists of three clauses: **select**, **from**, and **where**. A query takes as its input the relations listed in the **from** clause, operates on them as specified in the **where** and **select** clauses, and then produces a relation as the result. We introduce the SQL syntax through examples, and we describe the general structure of SQL queries later.

3.3.1 Queries on a Single Relation

Let us consider a simple query using our university example, “Find the names of all instructors.” Instructor names are found in the *instructor* relation, so we put that relation in the **from** clause. The instructor’s name appears in the *name* attribute, so we put that in the **select** clause.

```
select name  
from instructor;
```

The result is a relation consisting of a single attribute with the heading *name*. If the *instructor* relation is as shown in Figure 2.1, then the relation that results from the preceding query is shown in Figure 3.2.

<i>name</i>
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 3.2 Result of “select *name* from *instructor*”.

Now consider another query, “Find the department names of all instructors,” which can be written as:

```
select dept_name
      from instructor;
```

Since more than one instructor can belong to a department, a department name could appear more than once in the *instructor* relation. The result of the above query is a relation containing the department names, shown in Figure 3.3.

In the formal, mathematical definition of the relational model, a relation is a set. Thus, duplicate tuples would never appear in relations. In practice, duplicate elimination is time-consuming. Therefore, SQL allows duplicates in database relations as well as in the results of SQL expressions.¹ Thus, the preceding SQL query lists each department name once for every tuple in which it appears in the *instructor* relation.

In those cases where we want to force the elimination of duplicates, we insert the keyword **distinct** after **select**. We can rewrite the preceding query as:

```
select distinct dept_name
      from instructor;
```

if we want duplicates removed. The result of the above query would contain each department name at most once.

¹ Any database relation whose schema includes a primary-key declaration cannot contain duplicate tuples, since they would violate the primary-key constraint.

<i>dept_name</i>
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 3.3 Result of “select *dept_name* from *instructor*”.

SQL allows us to use the keyword **all** to specify explicitly that duplicates are not removed:

```
select all dept_name
      from instructor;
```

Since duplicate retention is the default, we shall not use **all** in our examples. To ensure the elimination of duplicates in the results of our example queries, we shall use **distinct** whenever it is necessary.

The **select** clause may also contain arithmetic expressions involving the operators **+**, **-**, *****, and **/** operating on constants or attributes of tuples. For example, the query:

```
select ID, name, dept_name, salary * 1.1
      from instructor;
```

returns a relation that is the same as the *instructor* relation, except that the attribute *salary* is multiplied by 1.1. This shows what would result if we gave a 10% raise to each instructor; note, however, that it does not result in any change to the *instructor* relation.

SQL also provides special data types, such as various forms of the *date* type, and allows several arithmetic functions to operate on these types. We discuss this further in Section 4.5.1.

The **where** clause allows us to select only those rows in the result relation of the **from** clause that satisfy a specified predicate. Consider the query “Find the names of all instructors in the Computer Science department who have salary greater than \$70,000.” This query can be written in SQL as:

<i>name</i>
Katz
Brandt

Figure 3.4 Result of “Find the names of all instructors in the Computer Science department who have salary greater than \$70,000.”

```
select name
  from instructor
 where dept_name = 'Comp. Sci.' and salary > 70000;
```

If the *instructor* relation is as shown in Figure 2.1, then the relation that results from the preceding query is shown in Figure 3.4.

SQL allows the use of the logical connectives **and**, **or**, and **not** in the **where** clause. The operands of the logical connectives can be expressions involving the comparison operators $<$, \leq , $>$, \geq , $=$, and \neq . SQL allows us to use the comparison operators to compare strings and arithmetic expressions, as well as special types, such as date types.

We shall explore other features of **where** clause predicates later in this chapter.

3.3.2 Queries on Multiple Relations

So far our example queries were on a single relation. Queries often need to access information from multiple relations. We now study how to write such queries.

As an example, suppose we want to answer the query “Retrieve the names of all instructors, along with their department names and department building name.”

Looking at the schema of the relation *instructor*, we realize that we can get the department name from the attribute *dept_name*, but the department building name is present in the attribute *building* of the relation *department*. To answer the query, each tuple in the *instructor* relation must be matched with the tuple in the *department* relation whose *dept_name* value matches the *dept_name* value of the *instructor* tuple.

In SQL, to answer the above query, we list the relations that need to be accessed in the **from** clause and specify the matching condition in the **where** clause. The above query can be written in SQL as

```
select name, instructor.dept_name, building
  from instructor, department
 where instructor.dept_name = department.dept_name;
```

If the *instructor* and *department* relations are as shown in Figure 2.1 and Figure 2.5 respectively, then the result of this query is shown in Figure 3.5.

Note that the attribute *dept_name* occurs in both the relations *instructor* and *department*, and the relation name is used as a prefix (in *instructor.dept_name*, and *de-*

<i>name</i>	<i>dept_name</i>	<i>building</i>
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

Figure 3.5 The result of “Retrieve the names of all instructors, along with their department names and department building name.”

partment.dept_name) to make clear to which attribute we are referring. In contrast, the attributes *name* and *building* appear in only one of the relations and therefore do not need to be prefixed by the relation name.

This naming convention *requires* that the relations that are present in the **from** clause have distinct names. This requirement causes problems in some cases, such as when information from two different tuples in the same relation needs to be combined. In Section 3.4.1, we see how to avoid these problems by using the rename operation.

We now consider the general case of SQL queries involving multiple relations. As we have seen earlier, an SQL query can contain three types of clauses, the **select** clause, the **from** clause, and the **where** clause. The role of each clause is as follows:

- The **select** clause is used to list the attributes desired in the result of a query.
- The **from** clause is a list of the relations to be accessed in the evaluation of the query.
- The **where** clause is a predicate involving attributes of the relation in the **from** clause.

A typical SQL query has the form:

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P;
```

Each A_i represents an attribute, and each r_i a relation. P is a predicate. If the **where** clause is omitted, the predicate P is **true**.

Although the clauses must be written in the order **select**, **from**, **where**, the easiest way to understand the operations specified by the query is to consider the clauses in operational order: first **from**, then **where**, and then **select**.²

The **from** clause by itself defines a Cartesian product of the relations listed in the clause. It is defined formally in terms of relational algebra, but it can also be understood as an iterative process that generates tuples for the result relation of the **from** clause.

```

for each tuple  $t_1$  in relation  $r_1$ 
  for each tuple  $t_2$  in relation  $r_2$ 
    ...
      for each tuple  $t_m$  in relation  $r_m$ 
        Concatenate  $t_1, t_2, \dots, t_m$  into a single tuple  $t$ 
        Add  $t$  into the result relation
    
```

The result relation has all attributes from all the relations in the **from** clause. Since the same attribute name may appear in both r_i and r_j , as we saw earlier, we prefix the name of the relation from which the attribute originally came, before the attribute name.

For example, the relation schema for the Cartesian product of relations *instructor* and *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 as:

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

To illustrate, consider the *instructor* relation in Figure 2.1 and the *teaches* relation in Figure 2.7. Their Cartesian product is shown in Figure 3.6, which includes only a portion of the tuples that make up the Cartesian product result.

The Cartesian product by itself combines tuples from *instructor* and *teaches* that are unrelated to each other. Each tuple in *instructor* is combined with *every* tuple in *teaches*, even those that refer to a different instructor. The result can be an extremely large relation, and it rarely makes sense to create such a Cartesian product.

²In practice, SQL may convert the expression into an equivalent form that can be processed more efficiently. However, we shall defer concerns about efficiency to Chapter 15 and Chapter 16.

<i>instructor.ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
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 3.6 The Cartesian product of the *instructor* relation with the *teaches* relation.

Instead, the predicate in the **where** clause is used to restrict the combinations created by the Cartesian product to those that are meaningful for the desired answer. We would likely want a query involving *instructor* and *teaches* to combine a particular tuple *t* in *instructor* with only those tuples in *teaches* that refer to the same instructor to which *t* refers. That is, we wish only to match *teaches* tuples with *instructor* tuples that have the same *ID* value. The following SQL query ensures this condition and outputs the instructor name and course identifiers from such matching tuples.

```
select name, course_id
  from instructor, teaches
 where instructor.ID= teaches.ID;
```



<i>name</i>	<i>course_id</i>
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

Figure 3.7 Result of “For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught.”

Note that the preceding query outputs only instructors who have taught some course. Instructors who have not taught any course are not output; if we wish to output such tuples, we could use an operation called the *outer join*, which is described in Section 4.1.3.

If the *instructor* relation is as shown in Figure 2.1 and the *teaches* relation is as shown in Figure 2.7, then the relation that results from the preceding query is shown in Figure 3.7. Observe that instructors Gold, Califieri, and Singh, who have not taught any course, do not appear in Figure 3.7.

If we wished to find only instructor names and course identifiers for instructors in the Computer Science department, we could add an extra predicate to the **where** clause, as shown below.

```
select name, course_id
  from instructor, teaches
 where instructor.ID=teaches.ID and instructor.dept_name = 'Comp. Sci.';
```

Note that since the *dept_name* attribute occurs only in the *instructor* relation, we could have used just *dept_name*, instead of *instructor.dept_name* in the above query.

In general, the meaning of an SQL query can be understood as follows:

1. Generate a Cartesian product of the relations listed in the **from** clause.
2. Apply the predicates specified in the **where** clause on the result of Step 1.

3. For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the **select** clause.

This sequence of steps helps make clear what the result of an SQL query should be, *not* how it should be executed. A real implementation of SQL would not execute the query in this fashion; it would instead optimize evaluation by generating (as far as possible) only elements of the Cartesian product that satisfy the **where** clause predicates. We study such implementation techniques in Chapter 15 and Chapter 16.

When writing queries, you should be careful to include appropriate **where** clause conditions. If you omit the **where** clause condition in the preceding SQL query, it will output the Cartesian product, which could be a huge relation. For the example *instructor* relation in Figure 2.1 and the example *teaches* relation in Figure 2.7, their Cartesian product has $12 * 13 = 156$ tuples—more than we can show in the text! To make matters worse, suppose we have a more realistic number of instructors than we show in our sample relations in the figures, say 200 instructors. Let's assume each instructor teaches three courses, so we have 600 tuples in the *teaches* relation. Then the preceding iterative process generates $200 * 600 = 120,000$ tuples in the result.

3.4

Additional Basic Operations

A number of additional basic operations are supported in SQL.

3.4.1 The Rename Operation

Consider again the query that we used earlier:

```
select name, course_id  
      from instructor, teaches  
     where instructor.ID= teaches.ID;
```

The result of this query is a relation with the following attributes:

name, course_id

The names of the attributes in the result are derived from the names of the attributes in the relations in the **from** clause.

We cannot, however, always derive names in this way, for several reasons: First, two relations in the **from** clause may have attributes with the same name, in which case an attribute name is duplicated in the result. Second, if we use an arithmetic expression in the **select** clause, the resultant attribute does not have a name. Third, even if an attribute name can be derived from the base relations as in the preceding example, we may want to change the attribute name in the result. Hence, SQL provides a way of renaming the attributes of a result relation. It uses the **as clause**, taking the form:

Note 3.1 SQL AND MULTISSET RELATIONAL ALGEBRA - PART 1

There is a close connection between relational algebra operations and SQL operations. One key difference is that, unlike the relational algebra, SQL allows duplicates. The SQL standard defines how many copies of each tuple are there in the output of a query, which depends, in turn, on how many copies of tuples are present in the input relations.

To model this behavior of SQL, a version of relational algebra, called the **multiset relational algebra**, is defined to work on multisets: sets that may contain duplicates. The basic operations in the multiset relational algebra are defined as follows:

1. If there are c_1 copies of tuple t_1 in r_1 , and t_1 satisfies selection σ_0 , then there are c_1 copies of t_1 in $\sigma_0(r_1)$.
2. For each copy of tuple t_1 in r_1 , there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$, where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1 .
3. If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2 , there are $c_1 * c_2$ copies of the tuple $t_1 \cdot t_2$ in $r_1 \times r_2$.

For example, suppose that relations r_1 with schema (A, B) and r_2 with schema (C) are the following multisets: $r_1 = \{(1, a), (2, a)\}$ and $r_2 = \{(2), (3), (3)\}$. Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$, whereas $\Pi_B(r_1) \times r_2$ would be:

$$\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$$

Now consider a basic SQL query of the form:

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P
```

Each A_i represents an attribute, and each r_i a relation. P is a predicate. If the **where** clause is omitted, the predicate P is **true**. The query is equivalent to the multiset relational-algebra expression:

$$\Pi_{A_1, A_2, \dots, A_n}(\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

The relational algebra *select* operation corresponds to the SQL **where** clause, not to the SQL **select** clause; the difference in meaning is an unfortunate historical fact. We discuss the representation of more complex SQL queries in Note 3.2 on page 97.

The relational-algebra representation of SQL queries helps to formally define the meaning of the SQL program. Further, database systems typically translate SQL queries into a lower-level representation based on relational algebra, and they perform query optimization and query evaluation using this representation.

old-name as new-name

The **as** clause can appear in both the **select** and **from** clauses.³

For example, if we want the attribute name *name* to be replaced with the name *instructor_name*, we can rewrite the preceding query as:

```
select name as instructor_name, course_id
  from instructor, teaches
 where instructor.ID= teaches.ID;
```

The **as** clause is particularly useful in renaming relations. One reason to rename a relation is to replace a long relation name with a shortened version that is more convenient to use elsewhere in the query. To illustrate, we rewrite the query “**For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught.**”

```
select T.name, S.course_id
  from instructor as T, teaches as S
 where T.ID= S.ID;
```

Another reason to rename a relation is a case where we wish to compare tuples in the same relation. We then need to take the Cartesian product of a relation with itself and, without renaming, it becomes impossible to distinguish one tuple from the other. Suppose that we want to write the query “**Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.**” We can write the SQL expression:

```
select distinct T.name
  from instructor as T, instructor as S
 where T.salary > S.salary and S.dept_name = 'Biology';
```

Observe that we could not use the notation *instructor.salary*, since it would not be clear which reference to *instructor* is intended.

In the above query, *T* and *S* can be thought of as copies of the relation *instructor*, but more precisely, they are declared as aliases, that is, as alternative names, for the relation *instructor*. An identifier, such as *T* and *S*, that is used to rename a relation is referred to as a **correlation name** in the SQL standard, but it is also commonly referred to as a **table alias**, or a **correlation variable**, or a **tuple variable**.

³Early versions of SQL did not include the keyword **as**. As a result, some implementations of SQL, notably Oracle, do not permit the keyword **as** in the **from** clause. In Oracle, “*old-name as new-name*” is written instead as “*old-name new-name*” in the **from** clause. The keyword **as** is permitted for renaming attributes in the **select** clause, but it is optional and may be omitted in Oracle.

Note that a better way to phrase the previous query in English would be “Find the names of all instructors who earn more than the lowest paid instructor in the Biology department.” Our original wording fits more closely with the SQL that we wrote, but the latter wording is more intuitive, and it can in fact be expressed directly in SQL as we shall see in Section 3.8.2.

3.4.2 String Operations

SQL specifies strings by enclosing them in single quotes, for example, 'Computer'. A single quote character that is part of a string can be specified by using two single quote characters; for example, the string “It’s right” can be specified by 'It's right'.

The SQL standard specifies that the equality operation on strings is case sensitive; as a result, the expression “comp. sci.’ = ‘Comp. Sci.’” evaluates to false. However, some database systems, such as MySQL and SQL Server, do not distinguish uppercase from lowercase when matching strings; as a result, ““comp. sci.’ = ‘Comp. Sci.’” would evaluate to true on these systems. This default behavior can, however, be changed, either at the database level or at the level of specific attributes.

SQL also permits a variety of functions on character strings, such as concatenating (using “||”), extracting substrings, finding the length of strings, converting strings to uppercase (using the function **upper(s)** where *s* is a string) and lowercase (using the function **lower(s)**), removing spaces at the end of the string (using **trim(s)**), and so on. There are variations on the exact set of string functions supported by different database systems. See your database system’s manual for more details on exactly what string functions it supports.

Pattern matching can be performed on strings using the operator **like**. We describe patterns by using two special characters:

- Percent (%): The % character matches any substring.
- Underscore (_): The _ character matches any character.

Patterns are case sensitive;⁴ that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

- 'Intro%' matches any string beginning with “Intro”.
- '%Comp%' matches any string containing “Comp” as a substring, for example, 'Intro. to Computer Science', and 'Computational Biology'.
- '___' matches any string of exactly three characters.
- '___%' matches any string of at least three characters.

⁴Except for MySQL, or with the **ilike** operator in PostgreSQL, where patterns are case insensitive.

SQL expresses patterns by using the **like** comparison operator. Consider the query “Find the names of all departments whose building name includes the substring ‘Watson.’” This query can be written as:

```
select dept_name
from department
where building like '%Watson%';
```

For patterns to include the special pattern characters (that is, % and _), SQL allows the specification of an escape character. The escape character is used immediately before a special pattern character to indicate that the special pattern character is to be treated like a normal character. We define the escape character for a **like** comparison using the **escape keyword**. To illustrate, consider the following patterns, which use a backslash (\) as the escape character:

- like 'ab\%cd%' **escape '\'** matches all strings beginning with “ab%cd”.
- like 'ab\\cd%' **escape '\'** matches all strings beginning with “ab\cd”.

SQL allows us to search for mismatches instead of matches by using the **not like** comparison operator. Some implementations provide variants of the **like** operation that do not distinguish lower- and uppercase.

Some SQL implementations, notably PostgreSQL, offer a **similar to** operation that provides more powerful pattern matching than the **like** operation; the syntax for specifying patterns is similar to that used in Unix regular expressions.

3.4.3 Attribute Specification in the Select Clause

The asterisk symbol “ * ” can be used in the **select** clause to denote “all attributes.” Thus, the use of *instructor.** in the **select** clause of the query:

```
select instructor.*
from instructor, teaches
where instructor.ID= teaches.ID;
```

indicates that all attributes of *instructor* are to be selected. A **select** clause of the form **select *** indicates that all attributes of the result relation of the **from** clause are selected.

3.4.4 Ordering the Display of Tuples

SQL offers the user some control over the order in which tuples in a relation are displayed. The **order by** clause causes the tuples in the result of a query to appear in sorted order. To list in alphabetic order all instructors in the Physics department, we write:

```

select name
from instructor
where dept_name = 'Physics'
order by name;

```

By default, the **order by** clause lists items in ascending order. To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order. Furthermore, ordering can be performed on multiple attributes. Suppose that we wish to list the entire *instructor* relation in descending order of *salary*. If several instructors have the same salary, we order them in ascending order by name. We express this query in SQL as follows:

```

select *
from instructor
order by salary desc, name asc;

```

3.4.5 Where-Clause Predicates

SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value. If we wish to find the names of instructors with salary amounts between \$90,000 and \$100,000, we can use the **between** comparison to write:

```

select name
from instructor
where salary between 90000 and 100000;

```

instead of:

```

select name
from instructor
where salary <= 100000 and salary >= 90000;

```

Similarly, we can use the **not between** comparison operator.

SQL permits us to use the notation (v_1, v_2, \dots, v_n) to denote a tuple of arity n containing values v_1, v_2, \dots, v_n ; the notation is called a *row constructor*. The comparison operators can be used on tuples, and the ordering is defined lexicographically. For example, $(a_1, a_2) \leq (b_1, b_2)$ is true if $a_1 \leq b_1$ and $a_2 \leq b_2$; similarly, the two tuples are equal if all their attributes are equal. Thus, the SQL query:

```

select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID and dept_name = 'Biology';

```

<i>course_id</i>
CS-101
CS-347
PHY-101

Figure 3.8 The *c1* relation, listing courses taught in Fall 2017.

can be rewritten as follows:⁵

```
select name, course_id
from instructor, teaches
where (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```

3.5

Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the mathematical set operations \cup , \cap , and $-$. We shall now construct queries involving the **union**, **intersect**, and **except** operations over two sets.

- The set of all courses taught in the Fall 2017 semester:

```
select course_id
from section
where semester = 'Fall' and year= 2017;
```

- The set of all courses taught in the Spring 2018 semester:

```
select course_id
from section
where semester = 'Spring' and year= 2018;
```

In our discussion that follows, we shall refer to the relations obtained as the result of the preceding queries as *c1* and *c2*, respectively, and show the results when these queries are run on the *section* relation of Figure 2.6 in Figure 3.8 and Figure 3.9. Observe that *c2* contains two tuples corresponding to *course_id* CS-319, since two sections of the course were offered in Spring 2018.

⁵Although it is part of the SQL-92 standard, some SQL implementations, notably Oracle, do not support this syntax.

<i>course_id</i>
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

Figure 3.9 The *c2* relation, listing courses taught in Spring 2018.

3.5.1 The Union Operation

To find the set of all courses taught either in Fall 2017 or in Spring 2018, or both, we write the following query. Note that the parentheses we include around each **select**-**from**-**where** statement below are optional but useful for ease of reading; some databases do not allow the use of the parentheses, in which case they may be dropped.



```
(select course_id
  from section
  where semester = 'Fall' and year= 2017)
union
(select course_id
  from section
  where semester = 'Spring' and year= 2018);
```

The **union** operation automatically eliminates duplicates, unlike the **select** clause. Thus, using the *section* relation of Figure 2.6, where two sections of CS-319 are offered in Spring 2018, and a section of CS-101 is offered in the Fall 2017 as well as in the Spring 2018 semesters, CS-101 and CS-319 appear only once in the result, shown in Figure 3.10.

If we want to retain all duplicates, we must write **union all** in place of **union**:

```
(select course_id
  from section
  where semester = 'Fall' and year= 2017)
union all
(select course_id
  from section
  where semester = 'Spring' and year= 2018);
```

The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both *c1* and *c2*. So, in the above query, each of CS-319 and CS-101 would

<i>course_id</i>
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 3.10 The result relation for *c1 union c2*.

be listed twice. As a further example, if it were the case that four sections of ECE-101 were taught in the Fall 2017 semester and two sections of ECE-101 were taught in the Spring 2018 semester, then there would be six tuples with ECE-101 in the result.

3.5.2 The Intersect Operation

To find the set of all courses taught in both the Fall 2017 and Spring 2018, we write:

```
(select course_id
  from section
  where semester = 'Fall' and year = 2017)
intersect
(select course_id
  from section
  where semester = 'Spring' and year = 2018);
```

The result relation, shown in Figure 3.11, contains only one tuple with CS-101. The **intersect** operation automatically eliminates duplicates.⁶ For example, if it were the case that four sections of ECE-101 were taught in the Fall 2017 semester and two sections of ECE-101 were taught in the Spring 2018 semester, then there would be only one tuple with ECE-101 in the result.

<i>course_id</i>
CS-101

Figure 3.11 The result relation for *c1 intersect c2*.

⁶MySQL does not implement the **intersect** operation; a work-around is to use subqueries as discussed in Section 3.8.1.

<i>course_id</i>
CS-347
PHY-101

Figure 3.12 The result relation for $c1$ except $c2$.

If we want to retain all duplicates, we must write **intersect all** in place of **intersect**:

```
(select course_id
  from section
  where semester = 'Fall' and year= 2017)
intersect all
(select course_id
  from section
  where semester = 'Spring' and year= 2018);
```

The number of duplicate tuples that appear in the result is equal to the minimum number of duplicates in both $c1$ and $c2$. For example, if four sections of ECE-101 were taught in the Fall 2017 semester and two sections of ECE-101 were taught in the Spring 2018 semester, then there would be two tuples with ECE-101 in the result.

3.5.3 The Except Operation

To find all courses taught in the Fall 2017 semester but not in the Spring 2018 semester, we write:

```
(select course_id
  from section
  where semester = 'Fall' and year= 2017)
except
(select course_id
  from section
  where semester = 'Spring' and year= 2018);
```

The result of this query is shown in Figure 3.12. Note that this is exactly relation $c1$ of Figure 3.8 except that the tuple for CS-101 does not appear. The **except** operation ⁷ outputs all tuples from its first input that do not occur in the second input; that is, it

⁷Some SQL implementations, notably Oracle, use the keyword **minus** in place of **except**, while Oracle 12c uses the keywords **multiset except** in place of **except all**. MySQL does not implement it at all; a work-around is to use subqueries as discussed in Section 3.8.1.

performs set difference. The operation automatically eliminates duplicates in the inputs before performing set difference. For example, if four sections of ECE-101 were taught in the Fall 2017 semester and two sections of ECE-101 were taught in the Spring 2018 semester, the result of the **except** operation would not have any copy of ECE-101.

If we want to retain duplicates, we must write **except all** in place of **except**:

```
(select course_id
  from section
  where semester = 'Fall' and year= 2017)
except all
(select course_id
  from section
  where semester = 'Spring' and year= 2018);
```

The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies in $c1$ minus the number of duplicate copies in $c2$, provided that the difference is positive. Thus, if four sections of ECE-101 were taught in the Fall 2017 semester and two sections of ECE-101 were taught in Spring 2018, then there are two tuples with ECE-101 in the result. If, however, there were two or fewer sections of ECE-101 in the Fall 2017 semester and two sections of ECE-101 in the Spring 2018 semester, there is no tuple with ECE-101 in the result.

3.6

Null Values

Null values present special problems in relational operations, including arithmetic operations, comparison operations, and set operations.

The result of an arithmetic expression (involving, for example, $+$, $-$, $*$, or $/$) is null if any of the input values is null. For example, if a query has an expression $r.A + 5$, and $r.A$ is null for a particular tuple, then the expression result must also be null for that tuple.

Comparisons involving nulls are more of a problem. For example, consider the comparison “ $1 < \text{null}$ ”. It would be wrong to say this is true since we do not know what the null value represents. But it would likewise be wrong to claim this expression is false; if we did, “**not** ($1 < \text{null}$)” would evaluate to true, which does not make sense. SQL therefore treats as **unknown** the result of any comparison involving a null value (other than predicates **is null** and **is not null**, which are described later in this section). This creates a third logical value in addition to *true* and *false*.

Since the predicate in a **where** clause can involve Boolean operations such as **and**, **or**, and **not** on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value **unknown**.

- 
 • **and:** The result of *true and unknown* is *unknown*, *false and unknown* is *false*, while *unknown and unknown* is *unknown*.
- **or:** The result of *true or unknown* is *true*, *false or unknown* is *unknown*, while *unknown or unknown* is *unknown*.
- **not:** The result of **not** *unknown* is *unknown*.

You can verify that if $r.A$ is null, then “ $1 < r.A$ ” as well as “**not** ($1 < r.A$)” evaluate to unknown.

If the **where** clause predicate evaluates to either **false** or **unknown** for a tuple, that tuple is not added to the result.

SQL uses the special keyword **null** in a predicate to test for a null value. Thus, to find all instructors who appear in the *instructor* relation with null values for *salary*, we write:

```
select name
  from instructor
 where salary is null;
```

The predicate **is not null** succeeds if the value on which it is applied is not null.

SQL allows us to test whether the result of a comparison is unknown, rather than true or false, by using the clauses **is unknown** and **is not unknown**.⁸ For example,

```
select name
  from instructor
 where salary > 10000 is unknown;
```

When a query uses the **select distinct** clause, duplicate tuples must be eliminated. For this purpose, when comparing values of corresponding attributes from two tuples, the values are treated as identical if either both are non-null and equal in value, or both are null. Thus, two copies of a tuple, such as $\{('A',\text{null}), ('A',\text{null})\}$, are treated as being identical, even if some of the attributes have a null value. Using the **distinct** clause then retains only one copy of such identical tuples. Note that the treatment of null above is different from the way nulls are treated in predicates, where a comparison “*null=null*” would return unknown, rather than true.

The approach of treating tuples as identical if they have the same values for all attributes, even if some of the values are null, is also used for the set operations union, intersection, and except.

⁸The **is unknown** and **is not unknown** constructs are not supported by several databases.

3.7

Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five standard built-in aggregate functions:⁹

- Average: **avg**
- Minimum: **min**
- Maximum: **max**
- Total: **sum**
- Count: **count**

The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

3.7.1 Basic Aggregation

Consider the query “Find the average salary of instructors in the Computer Science department.” We write this query as follows:

```
select avg (salary)
  from instructor
 where dept_name = 'Comp. Sci.';
```

The result of this query is a relation with a single attribute containing a single tuple with a numerical value corresponding to the average salary of instructors in the Computer Science department. The database system may give an awkward name to the result relation attribute that is generated by aggregation, consisting of the text of the expression; however, we can give a meaningful name to the attribute by using the **as** clause as follows:

```
select avg (salary) as avg_salary
  from instructor
 where dept_name = 'Comp. Sci.';
```

In the *instructor* relation of Figure 2.1, the salaries in the Computer Science department are \$75,000, \$65,000, and \$92,000. The average salary is $\$232,000/3 = \$77,333.33$.

Retaining duplicates is important in computing an average. Suppose the Computer Science department adds a fourth instructor whose salary happens to be \$75,000. If du-

⁹Most implementations of SQL offer a number of additional aggregate functions.

plices were eliminated, we would obtain the wrong answer ($\$232,000/4 = \$58,000$) rather than the correct answer of \$76,750.

There are cases where we must eliminate duplicates before computing an aggregate function. If we do want to eliminate duplicates, we use the keyword **distinct** in the aggregate expression. An example arises in the query “Find the total number of instructors who teach a course in the Spring 2018 semester.” In this case, an instructor counts only once, regardless of the number of course sections that the instructor teaches. The required information is contained in the relation *teaches*, and we write this query as follows:

```
select count (distinct ID)
from teaches
where semester = 'Spring' and year = 2018;
```

Because of the keyword **distinct** preceding *ID*, even if an instructor teaches more than one course, she is counted only once in the result.

We use the aggregate function **count** frequently to count the number of tuples in a relation. The notation for this function in SQL is **count (*)**. Thus, to find the number of tuples in the *course* relation, we write

```
select count (*)
from course;
```

SQL does not allow the use of **distinct** with **count (*)**. It is legal to use **distinct** with **max** and **min**, even though the result does not change. We can use the keyword **all** in place of **distinct** to specify duplicate retention, but since **all** is the default, there is no need to do so.

3.7.2 Aggregation with Grouping

There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this in SQL using the **group by** clause. The attribute or attributes given in the **group by** clause are used to form groups. Tuples with the same value on all attributes in the **group by** clause are placed in one group.

As an illustration, consider the query “Find the average salary in each department.” We write this query as follows:

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name;
```

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

Figure 3.13 Tuples of the *instructor* relation, grouped by the *dept_name* attribute.

Figure 3.13 shows the tuples in the *instructor* relation grouped by the *dept_name* attribute, which is the first step in computing the query result. The specified aggregate is computed for each group, and the result of the query is shown in Figure 3.14.

In contrast, consider the query “Find the average salary of all instructors.” We write this query as follows:

```
select avg (salary)
from instructor;
```

In this case the **group by** clause has been omitted, so the entire relation is treated as a single group.

<i>dept_name</i>	<i>avg_salary</i>
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Figure 3.14 The result relation for the query “Find the average salary in each department”.

As another example of aggregation on groups of tuples, consider the query “Find the number of instructors in each department who teach a course in the Spring 2018 semester.” Information about which instructors teach which course sections in which semester is available in the *teaches* relation. However, this information has to be joined with information from the *instructor* relation to get the department name of each instructor. Thus, we write this query as follows:

```
select dept_name, count (distinct ID) as instr_count
from instructor, teaches
where instructor.ID = teaches.ID and
      semester = 'Spring' and year = 2018
group by dept_name;
```

The result is shown in Figure 3.15.

When an SQL query uses grouping, it is important to ensure that the only attributes that appear in the **select** statement without being aggregated are those that are present in the **group by** clause. In other words, any attribute that is not present in the **group by** clause may appear in the **select** clause only as an argument to an aggregate function, otherwise the query is treated as erroneous. For example, the following query is erroneous since *ID* does not appear in the **group by** clause, and yet it appears in the **select** clause without being aggregated:

```
/* erroneous query */
select dept_name, ID, avg (salary)
from instructor
group by dept_name;
```

In the preceding query, each instructor in a particular group (defined by *dept_name*) can have a different *ID*, and since only one tuple is output for each group, there is no unique way of choosing which *ID* value to output. As a result, such cases are disallowed by SQL.

The preceding query also illustrates a comment written in SQL by enclosing text in “*/* */*”; the same comment could have also been written as “*-- erroneous query*”.

dept_name	instr_count
Comp. Sci.	3
Finance	1
History	1
Music	1

Figure 3.15 The result relation for the query “Find the number of instructors in each department who teach a course in the Spring 2018 semester.”

<i>dept_name</i>	<i>avg_salary</i>
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Figure 3.16 The result relation for the query “Find the average salary of instructors in those departments where the average salary is more than \$42,000.”

3.7.3 The Having Clause

At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than \$42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL. SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used in the **having** clause. We express this query in SQL as follows:

```
select dept_name, avg (salary) as avg_salary
  from instructor
 group by dept_name
 having avg (salary) > 42000;
```

The result is shown in Figure 3.16.

As was the case for the **select** clause, any attribute that is present in the **having** clause without being aggregated must appear in the **group by** clause, otherwise the query is erroneous.

The meaning of a query containing aggregation, **group by**, or **having** clauses is defined by the following sequence of operations:

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.
2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause.
3. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group.

4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed.
5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query “For each course section offered in 2017, find the average total credits (*tot_cred*) of all students enrolled in the section, if the section has at least 2 students.”

```
select course_id, semester, year, sec_id, avg (tot_cred)
from student, takes
where student.ID = takes.ID and year = 2017
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

Note that all the required information for the preceding query is available from the relations *takes* and *student*, and that although the query pertains to sections, a join with *section* is not needed.

3.7.4 Aggregation with Null and Boolean Values

Null values, when they exist, complicate the processing of aggregate operators. For example, assume that some tuples in the *instructor* relation have a null value for *salary*. Consider the following query to total all salary amounts:

```
select sum (salary)
from instructor;
```

The values to be summed in the preceding query include null values, since we assumed that some tuples have a null value for *salary*. Rather than say that the overall sum is itself *null*, the SQL standard says that the **sum** operator should ignore *null* values in its input.

In general, aggregate functions treat nulls according to the following rule: All aggregate functions except **count (*)** ignore null values in their input collection. As a result of null values being ignored, the collection of values may be empty. The **count** of an empty collection is defined to be 0, and all other aggregate operations return a value of null when applied on an empty collection. The effect of null values on some of the more complicated SQL constructs can be subtle.

A **Boolean** data type that can take values **true**, **false**, and **unknown** was introduced in SQL:1999. The aggregate functions **some** and **every** can be applied on a collection of Boolean values, and compute the disjunction (**or**) and conjunction (**and**), respectively, of the values.

Note 3.2 SQL AND MULTISSET RELATIONAL ALGEBRA - PART 2

As we saw earlier in Note 3.1 on page 80, the SQL **select**, **from**, and **where** clauses can be represented in the multiset relational algebra, using the multiset versions of the select, project, and Cartesian product operations.

The relational algebra union, intersection, and set difference (\cup , \cap , and $-$) operations can also be extended to the multiset relational algebra in a similar way, following the corresponding definitions of **union all**, **intersect all**, and **except all** in SQL, which we saw in Section 3.5; the SQL **union**, **intersect**, and **except** correspond to the set version of \cup , \cap , and $-$.

The extended relational algebra aggregate operation γ permits the use of aggregate functions on relation attributes. (The symbol \mathcal{G} is also used to represent the aggregate operation and was used in earlier editions of the book.) The operation $\text{dept_name} \gamma_{\text{average}(\text{salary})}(\text{instructor})$ groups the *instructor* relation on the *dept_name* attribute and computes the average salary for each group, as we saw earlier in Section 3.7.2. The subscript on the left side may be omitted, resulting in the entire input relation being in a single group. Thus, $\gamma_{\text{average}(\text{salary})}(\text{instructor})$ computes the average salary of all instructors. The aggregated values do not have an attribute name; they can be given a name either by using the rename operator ρ or for convenience using the following syntax:

$$\text{dept_name} \gamma_{\text{average}(\text{salary})} \text{ as avg_salary}(\text{instructor})$$

More complex SQL queries can also be rewritten in relational algebra. For example, the query:

```
select A1, A2, sum(A3)
from r1, r2, ..., rm
where P
group by A1, A2 having count(A4) > 2
```

is equivalent to:

$$t1 \leftarrow \sigma_P(r_1 \times r_2 \times \dots \times r_m) \\
\Pi_{A_1, A_2, \text{Sum}A_3}(\sigma_{\text{count}A_4 > 2}(A_1, A_2 \gamma_{\text{sum}(A_3)} \text{ as } \text{Sum}A_3, \text{count}(A_4) \text{ as } \text{count}A_4(t1)))$$

Join expressions in the **from** clause can be written using equivalent join expressions in relational algebra; we leave the details as an exercise for the reader. However, subqueries in the **where** or **select** clause cannot be rewritten into relational algebra in such a straightforward manner, since there is no relational algebra operation equivalent to the subquery construct. Extensions of relational algebra have been proposed for this task, but they are beyond the scope of this book.

3.8 Nested Subqueries

SQL provides a mechanism for nesting subqueries. A subquery is a **select-from-where** expression that is nested within another query. A common use of subqueries is to perform tests for set membership, make set comparisons, and determine set cardinality by nesting subqueries in the **where** clause. We study such uses of nested subqueries in the **where** clause in Section 3.8.1 through Section 3.8.4. In Section 3.8.5, we study nesting of subqueries in the **from** clause. In Section 3.8.7, we see how a class of subqueries called scalar subqueries can appear wherever an expression returning a value can occur.

3.8.1 Set Membership

SQL allows testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership.

As an illustration, reconsider the query “Find all the courses taught in both the Fall 2017 and Spring 2018 semesters.” Earlier, we wrote such a query by intersecting two sets: the set of courses taught in Fall 2017 and the set of courses taught in Spring 2018. We can take the alternative approach of finding all courses that were taught in Fall 2017 and that are also members of the set of courses taught in Spring 2018. This formulation generates the same results as the previous one did, but it leads us to write our query using the **in** connective of SQL. We begin by finding all courses taught in Spring 2018, and we write the subquery:

```
(select course_id
  from section
  where semester = 'Spring' and year= 2018)
```

We then need to find those courses that were taught in the Fall 2017 and that appear in the set of courses obtained in the subquery. We do so by nesting the subquery in the **where** clause of an outer query. The resulting query is:

```
select distinct course_id
  from section
  where semester = 'Fall' and year= 2017 and
        course_id in (select course_id
                       from section
                       where semester = 'Spring' and year= 2018);
```

Note that we need to use **distinct** here because the **intersect** operation removes duplicates by default.

This example shows that it is possible to write the same query several ways in SQL. This flexibility is beneficial, since it allows a user to think about the query in the way

that seems most natural. We shall see that there is a substantial amount of redundancy in SQL.

We use the **not in** construct in a way similar to the **in** construct. For example, to find all the courses taught in the Fall 2017 semester but not in the Spring 2018 semester, which we expressed earlier using the **except** operation, we can write:

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
course_id not in (select course_id
from section
where semester = 'Spring' and year= 2018);
```

The **in** and **not in** operators can also be used on enumerated sets. The following query selects the names of instructors whose names are neither “Mozart” nor “Einstein”.

```
select distinct name
from instructor
where name not in ('Mozart', 'Einstein');
```

In the preceding examples, we tested membership in a one-attribute relation. It is also possible to test for membership in an arbitrary relation in SQL. For example, we can write the query “find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 110011” as follows:

```
select count (distinct ID)
from takes
where (course_id, sec_id, semester, year) in (select course_id, sec_id, semester, year
from teaches
where teaches.ID= '10101');
```

Note, however, that some SQL implementations do not support the row construction syntax “(*course_id*, *sec_id*, *semester*, *year*)” used above. We will see alternative ways of writing this query in Section 3.8.3.

3.8.2 Set Comparison

As an example of the ability of a nested subquery to compare sets, consider the query “Find the names of all instructors whose salary is greater than at least one instructor in the Biology department.” In Section 3.4.1, we wrote this query as follows:

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept_name = 'Biology';
```

SQL does, however, offer an alternative style for writing the preceding query. The phrase “greater than at least one” is represented in SQL by `> some`. This construct allows us to rewrite the query in a form that resembles closely our formulation of the query in English.

```
select name
from instructor
where salary > some (select salary
from instructor
where dept_name = 'Biology');
```

The subquery:

```
(select salary
from instructor
where dept_name = 'Biology')
```

generates the set of all salary values of all instructors in the Biology department. The `> some` comparison in the `where` clause of the outer `select` is true if the `salary` value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology.

SQL also allows `< some`, `<= some`, `>= some`, `= some`, and `<> some` comparisons. As an exercise, verify that `= some` is identical to `in`, whereas `<> some` is *not* the same as `not in`.¹⁰

Now we modify our query slightly. Let us find the names of all instructors that have a salary value greater than that of each instructor in the Biology department. The construct `> all` corresponds to the phrase “greater than all.” Using this construct, we write the query as follows:

```
select name
from instructor
where salary > all (select salary
from instructor
where dept_name = 'Biology');
```

As it does for `some`, SQL also allows `< all`, `<= all`, `>= all`, `= all`, and `<> all` comparisons. As an exercise, verify that `<> all` is identical to `not in`, whereas `= all` is *not* the same as `in`.

¹⁰The keyword `any` is synonymous to `some` in SQL. Early versions of SQL allowed only `any`. Later versions added the alternative `some` to avoid the linguistic ambiguity of the word *any* in English.

As another example of set comparisons, consider the query “Find the departments that have the highest average salary.” We begin by writing a query to find all average salaries, and then nest it as a subquery of a larger query that finds those departments for which the average salary is greater than or equal to all average salaries:

```
select dept_name
  from instructor
 group by dept_name
 having avg (salary) >= all (select avg (salary)
                                from instructor
                                group by dept_name);
```

3.8.3 Test for Empty Relations

SQL includes a feature for testing whether a subquery has any tuples in its result. The `exists` construct returns the value `true` if the argument subquery is nonempty. Using the `exists` construct, we can write the query “Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester” in still another way:

```
select course_id
  from section as S
 where semester = 'Fall' and year= 2017 and
 exists (select *
           from section as T
           where semester = 'Spring' and year= 2018 and
           S.course_id= T.course_id);
```

The above query also illustrates a feature of SQL where a **correlation name** from an outer query (*S* in the above query), can be used in a subquery in the `where` clause. A subquery that uses a correlation name from an outer query is called a **correlated subquery**.

In queries that contain subqueries, a scoping rule applies for correlation names. In a subquery, according to the rule, it is legal to use only correlation names defined in the subquery itself or in any query that contains the subquery. If a correlation name is defined both locally in a subquery and globally in a containing query, the local definition applies. This rule is analogous to the usual scoping rules used for variables in programming languages.

We can test for the nonexistence of tuples in a subquery by using the `not exists` construct. We can use the `not exists` construct to simulate the set containment (that is, superset) operation: We can write “relation *A* contains relation *B*” as “`not exists (B except A)`.” (Although it is not part of the current SQL standards, the `contains` operator was present in some early relational systems.) To illustrate the `not exists` operator,

consider the query “Find all students who have taken all courses offered in the Biology department.” Using the **except** construct, we can write the query as follows:

```
select S.ID, S.name
from student as S
where not exists ((select course_id
from course
where dept_name = 'Biology')
except
(select T.course_id
from takes as T
where S.ID = T.ID));
```

Here, the subquery:

```
(select course_id
from course
where dept_name = 'Biology')
```

finds the set of all courses offered in the Biology department. The subquery:

```
(select T.course_id
from takes as T
where S.ID = T.ID)
```

finds all the courses that student *S.ID* has taken. Thus, the outer **select** takes each student and tests whether the set of all courses that the student has taken contains the set of all courses offered in the Biology department.

We saw in Section 3.8.1, an SQL query to “find the total number of (distinct) students who have taken course sections taught by the instructor with *ID* 110011”. That query used a tuple constructor syntax that is not supported by some databases. An alternative way to write the query, using the **exists** construct, is as follows:

```
select count (distinct ID)
from takes
where exists (select course_id, sec_id, semester, year
from teaches
where teaches.ID= '10101'
and takes.course_id = teaches.course_id
and takes.sec_id = teaches.sec_id
and takes.semester = teaches.semester
and takes.year = teaches.year
);
);
```

3.8.4 Test for the Absence of Duplicate Tuples

SQL includes a Boolean function for testing whether a subquery has duplicate tuples in its result. The **unique** construct¹¹ returns the value **true** if the argument subquery contains no duplicate tuples. Using the **unique** construct, we can write the query “Find all courses that were offered at most once in 2017” as follows:

```
select T.course_id
from course as T
where unique (select R.course_id
               from section as R
               where T.course_id= R.course_id and
                     R.year = 2017);
```

Note that if a course were not offered in 2017, the subquery would return an empty result, and the **unique** predicate would evaluate to true on the empty set.

An equivalent version of this query not using the **unique** construct is:

```
select T.course_id
from course as T
where 1 >= (select count(R.course_id)
               from section as R
               where T.course_id= R.course_id and
                     R.year = 2017);
```

We can test for the existence of duplicate tuples in a subquery by using the **not unique** construct. To illustrate this construct, consider the query “Find all courses that were offered at least twice in 2017” as follows:

```
select T.course_id
from course as T
where not unique (select R.course_id
                   from section as R
                   where T.course_id= R.course_id and
                         R.year = 2017);
```

Formally, the **unique** test on a relation is defined to fail if and only if the relation contains two distinct tuples t_1 and t_2 such that $t_1 = t_2$. Since the test $t_1 = t_2$ fails if any of the fields of t_1 or t_2 are null, it is possible for **unique** to be true even if there are multiple copies of a tuple, as long as at least one of the attributes of the tuple is null.

¹¹This construct is not yet widely implemented.

3.8.5 Subqueries in the From Clause

SQL allows a subquery expression to be used in the **from** clause. The key concept applied here is that any **select-from-where** expression returns a relation as a result and, therefore, can be inserted into another **select-from-where** anywhere that a relation can appear.

Consider the query “Find the average instructors’ salaries of those departments where the average salary is greater than \$42,000.” We wrote this query in Section 3.7 by using the **having** clause. We can now rewrite this query, without using the **having** clause, by using a subquery in the **from** clause, as follows:

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
       from instructor
       group by dept_name)
where avg_salary > 42000;
```

The subquery generates a relation consisting of the names of all departments and their corresponding average instructors’ salaries. The attributes of the subquery result can be used in the outer query, as can be seen in the above example.

Note that we do not need to use the **having** clause, since the subquery in the **from** clause computes the average salary, and the predicate that was in the **having** clause earlier is now in the **where** clause of the outer query.

We can give the subquery result relation a name, and rename the attributes, using the **as** clause, as illustrated below.

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
       from instructor
       group by dept_name)
       as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

The subquery result relation is named *dept_avg*, with the attributes *dept_name* and *avg_salary*.

Nested subqueries in the **from** clause are supported by most but not all SQL implementations. Note that some SQL implementations, notably MySQL and PostgreSQL, require that each subquery relation in the **from** clause must be given a name, even if the name is never referenced; Oracle allows a subquery result relation to be given a name (with the keyword **as** omitted) but does not allow renaming of attributes of the relation. An easy workaround for that is to do the attribute renaming in the **select** clause of the subquery; in the above query, the **select** clause of the subquery would be replaced by

```
select dept_name, avg(salary) as avg_salary
```

and

`“as dept_avg (dept_name, avg_salary)”`

would be replaced by

`“as dept_avg”.`

As another example, suppose we wish to find the maximum across all departments of the total of all instructors' salaries in each department. The **having** clause does not help us in this task, but we can write this query easily by using a subquery in the **from** clause, as follows:

```
select max (tot_salary)
from (select dept_name, sum(salary)
      from instructor
      group by dept_name) as dept_total (dept_name, tot_salary);
```

We note that nested subqueries in the **from** clause cannot use correlation variables from other relations in the same **from** clause. However, the SQL standard, starting with SQL:2003, allows a subquery in the **from** clause that is prefixed by the **lateral** keyword to access attributes of preceding tables or subqueries in the same **from** clause. For example, if we wish to print the names of each instructor, along with their salary and the average salary in their department, we could write the query as follows:

```
select name, salary, avg_salary
from instructor I1, lateral (select avg(salary) as avg_salary
                           from instructor I2
                           where I2.dept_name= I1.dept_name);
```

Without the **lateral** clause, the subquery cannot access the correlation variable *I1* from the outer query. Only the more recent implementations of SQL support the **lateral** clause.

3.8.6 The With Clause

The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs. Consider the following query, which finds those departments with the maximum budget.

```
with max_budget (value) as
      (select max(budget)
       from department)
select budget
      from department, max_budget
     where department.budget = max_budget.value;
```

The **with** clause in the query defines the temporary relation *max_budget* containing the results of the subquery defining the relation. The relation is available for use only within later parts of the same query.¹² The **with** clause, introduced in SQL:1999, is supported by many, but not all, database systems.

We could have written the preceding query by using a nested subquery in either the **from** clause or the **where** clause. However, using nested subqueries would have made the query harder to read and understand. The **with** clause makes the query logic clearer; it also permits this temporary relation to be used in multiple places within a query.

For example, suppose we want to find all departments where the total salary is greater than the average of the total salary at all departments. We can write the query using the **with** clause as follows.

```
with dept_total (dept_name, value) as
    (select dept_name, sum(salary)
     from instructor
     group by dept_name),
dept_total_avg(value) as
    (select avg(value)
     from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value;
```

We can create an equivalent query without the **with** clause, but it would be more complicated and harder to understand. You can write the equivalent query as an exercise.

3.8.7 Scalar Subqueries

SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called **scalar subqueries**. For example, a subquery can be used in the **select** clause as illustrated in the following example that lists all departments along with the number of instructors in each department:

```
select dept_name,
    (select count(*)
     from instructor
     where department.dept_name = instructor.dept_name)
    as num_instructors
from department;
```

¹²The SQL evaluation engine may not physically create the relation and is free to compute the overall query result in alternative ways, as long as the result of the query is the same as if the relation had been created.

The subquery in this example is guaranteed to return only a single value since it has a **count(*)** aggregate without a **group by**. The example also illustrates the usage of correlation variables, that is, attributes of relations in the **from** clause of the outer query, such as *department.dept_name* in the above example.

Scalar subqueries can occur in **select**, **where**, and **having** clauses. Scalar subqueries may also be defined without aggregates. It is not always possible to figure out at compile time if a subquery can return more than one tuple in its result; if the result has more than one tuple when the subquery is executed, a run-time error occurs.

Note that technically the type of a scalar subquery result is still a relation, even if it contains a single tuple. However, when a scalar subquery is used in an expression where a value is expected, SQL implicitly extracts the value from the single attribute of the single tuple in the relation and returns that value.

3.8.8 Scalar Without a From Clause

Certain queries require a calculation but no reference to any relation. Similarly, certain queries may have subqueries that contain a **from** clause without the top-level query needing a **from** clause.

As an example, suppose we wish to find the average number of sections taught (regardless of year or semester) per instructor, with sections taught by multiple instructors counted once per instructor. We need to count the number of tuples in *teaches* to find the total number of sections taught and count the number of tuples in *instructor* to find the number of instructors. Then a simple division gives us the desired result. One might write this as:

```
(select count (*) from teaches) / (select count (*) from instructor);
```

While this is legal in some systems, others will report an error due to the lack of a **from** clause.¹³ In the latter case, a special dummy relation called, for example, *dual* can be created, containing a single tuple. This allows the preceding query to be written as:

```
select (select count (*) from teaches) / (select count (*) from instructor)
      from dual;
```

Oracle provides a predefined relation called *dual*, containing a single tuple, for uses such as the above (the relation has a single attribute, which is not relevant for our purposes); you can create an equivalent relation if you use any other database.

Since the above queries divide one integer by another, the result would, on most databases, be an integer, which would result in loss of precision. If you wish to get the result as a floating point number, you could multiply one of the two subquery results by 1.0 to convert it to a floating point number, before the division operation is performed.

¹³This construct is legal, for example, in SQL Server, but not legal, for example, in Oracle.

Note 3.3 SQL AND MULTISER RELATIONAL ALGEBRA - PART 3

Unlike the SQL set and aggregation operations that we studied earlier in this chapter, SQL subqueries do not have directly equivalent operations in the relational algebra. Most SQL queries involving subqueries can be rewritten in a way that does not require the use of subqueries, and thus they have equivalent relational algebra expressions.

Rewriting to relational algebra can benefit from two extended relational algebra operations called *semijoin*, denoted \bowtie , and *antijoin*, denoted $\overline{\bowtie}$, which are supported internally by many database implementations (the symbol \triangleright is sometimes used in place of $\overline{\bowtie}$ to denote antijoin). For example, given relations r and s , $r \bowtie_{r.A=s.B} s$ outputs all tuples in r that have at least one tuple in s whose $s.B$ attribute value matches that tuples $r.A$ attribute value. Conversely, $r \overline{\bowtie}_{r.A=s.B} s$ outputs all tuples in r that have do not have any such matching tuple in s . These operators can be used to rewrite many subqueries that use the **exists** and **not exists** connectives.

Semijoin and antijoin can be expressed using other relational algebra operations, so they do not add any expressive power, but they are nevertheless quite useful in practice since they can be implemented very efficiently.

However, the process of rewriting SQL queries that contain subqueries is in general not straightforward. Database system implementations therefore extend the relational algebra by allowing σ and Π operators to invoke subqueries in their predicates and projection lists.

3.9 Modification of the Database

We have restricted our attention until now to the extraction of information from the database. Now, we show how to add, remove, or change information with SQL.

3.9.1 Deletion

A **delete** request is expressed in much the same way as a query. We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by:

```
delete from r  
where P;
```

where P represents a predicate and r represents a relation. The **delete** statement first finds all tuples t in r for which $P(t)$ is true, and then deletes them from r . The **where** clause can be omitted, in which case all tuples in r are deleted.

Note that a **delete** command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation. The predicate in the **where** clause may be as complex as a **select** command's **where** clause. At the other extreme, the **where** clause may be empty. The request:

 **delete from** *instructor*;

deletes all tuples from the *instructor* relation. The *instructor* relation itself still exists, but it is empty.

Here are examples of SQL delete requests:

- Delete all tuples in the *instructor* relation pertaining to instructors in the Finance department.

```
delete from instructor
where dept_name = 'Finance';
```

- Delete all instructors with a salary between \$13,000 and \$15,000.

```
delete from instructor
where salary between 13000 and 15000;
```

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

```
delete from instructor
where dept_name in (select dept_name
from department
where building = 'Watson');
```

This **delete** request first finds all departments located in Watson and then deletes all *instructor* tuples pertaining to those departments.

Note that, although we may delete tuples from only one relation at a time, we may reference any number of relations in a **select-from-where** nested in the **where** clause of a **delete**. The **delete** request can contain a nested **select** that references the relation from which tuples are to be deleted. For example, suppose that we want to delete the records of all instructors with salary below the average at the university. We could write:

```
delete from instructor
where salary < (select avg (salary)
from instructor);
```

The **delete** statement first tests each tuple in the relation *instructor* to check whether the salary is less than the average salary of instructors in the university. Then, all tuples that pass the test—that is, represent an instructor with a lower-than-average salary—are deleted. Performing all the tests before performing any deletion is important—if some tuples are deleted before other tuples have been tested, the average salary may change, and the final result of the **delete** would depend on the order in which the tuples were processed!

3.9.2 Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. The attribute values for inserted tuples must be members of the corresponding attribute's domain. Similarly, tuples inserted must have the correct number of attributes.

The simplest **insert** statement is a request to insert one tuple. Suppose that we wish to insert the fact that there is a course CS-437 in the Computer Science department with title “Database Systems” and four credit hours. We write:

```
insert into course
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

In this example, the values are specified in the order in which the corresponding attributes are listed in the relation schema. For the benefit of users who may not remember the order of the attributes, SQL allows the attributes to be specified as part of the **insert** statement. For example, the following SQL **insert** statements are identical in function to the preceding one:

```
insert into course (course_id, title, dept_name, credits)
    values ('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

```
insert into course (title, course_id, credits, dept_name)
    values ('Database Systems', 'CS-437', 4, 'Comp. Sci.');
```

More generally, we might want to insert tuples on the basis of the result of a query. Suppose that we want to make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of \$18,000. We write:

```
insert into instructor
    select ID, name, dept_name, 18000
        from student
    where dept_name = 'Music' and tot_cred > 144;
```

Instead of specifying a tuple as we did earlier in this section, we use a **select** to specify a set of tuples. SQL evaluates the **select** statement first, giving a set of tuples that is then inserted into the *instructor* relation. Each tuple has an *ID*, a *name*, a *dept_name* (Music), and a salary of \$18,000.

It is important that the system evaluate the **select** statement fully before it performs any insertions. If it were to carry out some insertions while the **select** statement was being evaluated, a request such as:

```
insert into student
select *
from student;
```

might insert an infinite number of tuples, if the primary key constraint on *student* were absent. Without the primary key constraint, the request would insert the first tuple in *student* again, creating a second copy of the tuple. Since this second copy is part of *student* now, the **select** statement may find it, and a third copy would be inserted into *student*. The **select** statement may then find this third copy and insert a fourth copy, and so on, forever. Evaluating the **select** statement completely before performing insertions avoids such problems. Thus, the above **insert** statement would simply duplicate every tuple in the *student* relation if the relation did not have a primary key constraint.

Our discussion of the **insert** statement considered only examples in which a value is given for every attribute in inserted tuples. It is possible for inserted tuples to be given values on only some attributes of the schema. The remaining attributes are assigned a null value denoted by *null*. Consider the request:

```
insert into student
values ('3003', 'Green', 'Finance', null);
```

The tuple inserted by this request specified that a student with *ID* “3003” is in the Finance department, but the *tot_cred* value for this student is not known.

Most relational database products have special “bulk loader” utilities to insert a large set of tuples into a relation. These utilities allow data to be read from formatted text files, and they can execute much faster than an equivalent sequence of **insert** statements.

3.9.3 Updates

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple. For this purpose, the **update** statement can be used. As we could for **insert** and **delete**, we can choose the tuples to be updated by using a query.

Suppose that annual salary increases are being made, and salaries of all instructors are to be increased by 5 percent. We write:

 **update** *instructor*
set *salary* = *salary* * 1.05;

The preceding update statement is applied once to each of the tuples in the *instructor* relation.

If a salary increase is to be paid only to instructors with a salary of less than \$70,000, we can write:

```
update instructor  

set salary = salary * 1.05  

where salary < 70000;
```

In general, the **where** clause of the **update** statement may contain any construct legal in the **where** clause of the **select** statement (including nested **selects**). As with **insert** and **delete**, a nested **select** within an **update** statement may reference the relation that is being updated. As before, SQL first tests all tuples in the relation to see whether they should be updated, and it carries out the updates afterward. For example, we can write the request “Give a 5 percent salary raise to instructors whose salary is less than average” as follows:

```
update instructor  

set salary = salary * 1.05  

where salary < (select avg (salary)  

from instructor);
```

Let us now suppose that all instructors with salary over \$100,000 receive a 3 percent raise, whereas all others receive a 5 percent raise. We could write two **update** statements:

```
update instructor  

set salary = salary * 1.03  

where salary > 100000;
```

```
update instructor  

set salary = salary * 1.05  

where salary <= 100000;
```

Note that the order of the two **update** statements is important. If we changed the order of the two statements, an instructor with a salary just under \$100,000 would receive a raise of over 8 percent.

SQL provides a **case** construct that we can use to perform both updates with a single **update** statement, avoiding the problem with the order of updates.

```
update instructor
set salary = case
    when salary <= 100000 then salary * 1.05
    else salary * 1.03
end
```

The general form of the case statement is as follows:

```
case
    when pred1 then result1
    when pred2 then result2
    ...
    when predn then resultn
    else result0
end
```

The operation returns *result_i*, where *i* is the first of *pred₁*, *pred₂*, ..., *pred_n* that is satisfied; if none of the predicates is satisfied, the operation returns *result₀*. Case statements can be used in any place where a value is expected.

Scalar subqueries are useful in SQL update statements, where they can be used in the **set** clause. We illustrate this using the *student* and *takes* relations that we introduced in Chapter 2. Consider an update where we set the *tot_cred* attribute of each *student* tuple to the sum of the credits of courses successfully completed by the student. We assume that a course is successfully completed if the student has a grade that is neither 'F' nor null. To specify this update, we need to use a subquery in the **set** clause, as shown below:

```
update student
set tot_cred = (
    select sum(credits)
    from takes, course
    where student.ID = takes.ID and
        takes.course_id = course.course_id and
        takes.grade <> 'F' and
        takes.grade is not null);
```

In case a student has not successfully completed any course, the preceding statement would set the *tot_cred* attribute value to null. To set the value to 0 instead, we could use another **update** statement to replace null values with 0; a better alternative is to replace the clause "select sum(*credits*)" in the preceding subquery with the following **select** clause using a **case** expression:

```
select case
    when sum(credits) is not null then sum(credits)
    else 0
end
```

Many systems support a **coalesce** function, which we describe in more detail later, in Section 4.5.2, which provides a concise way of replacing nulls by other values. In the above example, we could have used **coalesce(sum(credits), 0)** instead of the **case** expression; this expression would return the aggregate result **sum(credits)** if it is not null, and 0 otherwise.

3.10 Summary

- SQL is the most influential commercially marketed relational query language. The SQL language has several parts:
 - **Data-definition language** (DDL), which provides commands for defining relation schemas, deleting relations, and modifying relation schemas.
 - **Data-manipulation language** (DML), which includes a query language and commands to insert tuples into, delete tuples from, and modify tuples in the database.
- The SQL data-definition language is used to create relations with specified schemas. In addition to specifying the names and types of relation attributes, SQL also allows the specification of integrity constraints such as primary-key constraints and foreign-key constraints.
- SQL includes a variety of language constructs for queries on the database. These include the **select**, **from**, and **where** clauses.
- SQL also provides mechanisms to rename both attributes and relations, and to order query results by sorting on specified attributes.
- SQL supports basic set operations on relations, including **union**, **intersect**, and **except**, which correspond to the mathematical set operations \cup , \cap , and $-$.
- SQL handles queries on relations containing null values by adding the truth value “unknown” to the usual truth values of true and false.
- SQL supports aggregation, including the ability to divide a relation into groups, applying aggregation separately on each group. SQL also supports set operations on groups.
- SQL supports nested subqueries in the **where** and **from** clauses of an outer query. It also supports scalar subqueries wherever an expression returning a value is permitted.
- SQL provides constructs for updating, inserting, and deleting information.

Review Terms

- Data-definition language
- Data-manipulation language
- Database schema
- Database instance
- Relation schema
- Relation instance
- Primary key
- Foreign key
 - Referencing relation
 - Referenced relation
- Null value
- Query language
- SQL query structure
 - **select** clause
 - **from** clause
 - **where** clause
- Multiset relational algebra
- **as** clause
- **order by** clause
- Table alias
- Correlation name (correlation variable, tuple variable)
- Set operations
 - **union**
 - **intersect**
 - **except**
- Aggregate functions
 - **avg, min, max, sum, count**
 - **group by**
 - **having**
- Nested subqueries
- Set comparisons
 - {<, <=, >, >=} { **some, all** }
 - **exists**
 - **unique**
- **lateral** clause
- **with** clause
- Scalar subquery
- Database modification
 - Delete
 - Insert
 - Update

Practice Exercises

- 3.1** Write the following queries in SQL, using the university schema. (We suggest you actually run these queries on a database, using the sample data that we provide on the web site of the book, db-book.com. Instructions for setting up a database, and loading sample data, are provided on the above web site.)
- a. Find the titles of courses in the Comp. Sci. department that have 3 credits.
 - b. Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.

- c. Find the highest salary of any instructor.
 - d. Find all instructors earning the highest salary (there may be more than one with the same salary).
 - e. Find the enrollment of each section that was offered in Fall 2017.
 - f. Find the maximum enrollment, across all sections, in Fall 2017.
 - g. Find the sections that had the maximum enrollment in Fall 2017.
- 3.2** Suppose you are given a relation *grade_points*(*grade*, *points*) that provides a conversion from letter grades in the *takes* relation to numeric scores; for example, an “A” grade could be specified to correspond to 4 points, an “A–” to 3.7 points, a “B+” to 3.3 points, a “B” to 3 points, and so on. The grade points earned by a student for a course offering (section) is defined as the number of credits for the course multiplied by the numeric points for the grade that the student received. Given the preceding relation, and our university schema, write each of the following queries in SQL. You may assume for simplicity that no *takes* tuple has the *null* value for *grade*.
- a. Find the total grade points earned by the student with ID '12345', across all courses taken by the student.
 - b. Find the grade point average (*GPA*) for the above student, that is, the total grade points divided by the total credits for the associated courses.
 - c. Find the ID and the grade-point average of each student.
 - d. Now reconsider your answers to the earlier parts of this exercise under the assumption that some grades might be *null*. Explain whether your solutions still work and, if not, provide versions that handle *nulls* properly.
- 3.3** Write the following inserts, deletes, or updates in SQL, using the university schema.
- a. Increase the salary of each instructor in the Comp. Sci. department by 10%.
 - b. Delete all courses that have never been offered (i.e., do not occur in the *section* relation).
 - c. Insert every student whose *tot_cred* attribute is greater than 100 as an instructor in the same department, with a salary of \$10,000.
- 3.4** Consider the insurance database of Figure 3.17, where the primary keys are underlined. Construct the following SQL queries for this relational database.
- a. Find the total number of people who owned cars that were involved in accidents in 2017.

```

person (driver_id, name, address)
car (license_plate, model, year)
accident (report_number, year, location)
owns (driver_id, license_plate)
participated (report_number, license_plate, driver_id, damage_amount)

```

Figure 3.17 Insurance database

- b. Delete all year-2010 cars belonging to the person whose ID is '12345'.
- 3.5** Suppose that we have a relation $\text{marks}(ID, score)$ and we wish to assign grades to students based on the score as follows: grade *F* if $score < 40$, grade *C* if $40 \leq score < 60$, grade *B* if $60 \leq score < 80$, and grade *A* if $80 \leq score$. Write SQL queries to do the following:
- a. Display the grade for each student, based on the *marks* relation.
 - b. Find the number of students with each grade.
- 3.6** The SQL **like** operator is case sensitive (in most systems), but the **lower()** function on strings can be used to perform case-insensitive matching. To show how, write a query that finds departments whose names contain the string “sci” as a substring, regardless of the case.
- 3.7** Consider the SQL query

```

select p.a1
from p, r1, r2
where p.a1 = r1.a1 or p.a1 = r2.a1

```

Under what conditions does the preceding query select values of *p.a1* that are either in *r1* or in *r2*? Examine carefully the cases where either *r1* or *r2* may be empty.

- 3.8** Consider the bank database of Figure 3.18, where the primary keys are underlined. Construct the following SQL queries for this relational database.
- a. Find the ID of each customer of the bank who has an account but not a loan.
 - b. Find the ID of each customer who lives on the same street and in the same city as customer '12345'.
 - c. Find the name of each branch that has at least one customer who has an account in the bank and who lives in “Harrison”.

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 3.18 Banking database.

- 3.9** Consider the relational database of Figure 3.19, where the primary keys are underlined. Give an expression in SQL for each of the following queries.
- Find the ID, name, and city of residence of each employee who works for “First Bank Corporation”.
 - Find the ID, name, and city of residence of each employee who works for “First Bank Corporation” and earns more than \$10000.
 - Find the ID of each employee who does not work for “First Bank Corporation”.
 - Find the ID of each employee who earns more than every employee of “Small Bank Corporation”.
 - Assume that companies may be located in several cities. Find the name of each company that is located in every city in which “Small Bank Corporation” is located.
 - Find the name of the company that has the most employees (or companies, in the case where there is a tie for the most).
 - Find the name of each company whose employees earn a higher salary, on average, than the average salary at “First Bank Corporation”.

employee (ID, person_name, street, city)
works (ID, company_name, salary)
company (company_name, city)
manages (ID, manager_id)

Figure 3.19 Employee database.

- 3.10** Consider the relational database of Figure 3.19. Give an expression in SQL for each of the following:
- Modify the database so that the employee whose ID is '12345' now lives in "Newtown".
 - Give each manager of "First Bank Corporation" a 10 percent raise unless the salary becomes greater than \$100000; in such cases, give only a 3 percent raise.

Exercises

- 3.11** Write the following queries in SQL, using the university schema.
- Find the ID and name of each student who has taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.
 - Find the ID and name of each student who has not taken any course offered before 2017.
 - For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.
 - Find the lowest, across all departments, of the per-department maximum salary computed by the preceding query.
- 3.12** Write the SQL statements using the university schema to perform the following operations:
- Create a new course "CS-001", titled "Weekly Seminar", with 0 credits.
 - Create a section of this course in Fall 2017, with *sec_id* of 1, and with the location of this section not yet specified.
 - Enroll every student in the Comp. Sci. department in the above section.
 - Delete enrollments in the above section where the student's ID is 12345.
 - Delete the course CS-001. What will happen if you run this **delete** statement without first deleting offerings (sections) of this course?
 - Delete all *takes* tuples corresponding to any section of any course with the word "advanced" as a part of the title; ignore case when matching the word with the title.
- 3.13** Write SQL DDL corresponding to the schema in Figure 3.17. Make any reasonable assumptions about data types, and be sure to declare primary and foreign keys.

- 3.14** Consider the insurance database of Figure 3.17, where the primary keys are underlined. Construct the following SQL queries for this relational database.
- Find the number of accidents involving a car belonging to a person named “John Smith”.
 - Update the damage amount for the car with license_plate “AABB2000” in the accident with report number “AR2197” to \$3000.
- 3.15** Consider the bank database of Figure 3.18, where the primary keys are underlined. Construct the following SQL queries for this relational database.
- Find each customer who has an account at *every* branch located in “Brooklyn”.
 - Find the total sum of all loan amounts in the bank.
 - Find the names of all branches that have assets greater than those of at least one branch located in “Brooklyn”.
- 3.16** Consider the employee database of Figure 3.19, where the primary keys are underlined. Give an expression in SQL for each of the following queries.
- Find ID and name of each employee who lives in the same city as the location of the company for which the employee works.
 - Find ID and name of each employee who lives in the same city and on the same street as does her or his manager.
 - Find ID and name of each employee who earns more than the average salary of all employees of her or his company.
 - Find the company that has the smallest payroll.
- 3.17** Consider the employee database of Figure 3.19. Give an expression in SQL for each of the following queries.
- Give all employees of “First Bank Corporation” a 10 percent raise.
 - Give all managers of “First Bank Corporation” a 10 percent raise.
 - Delete all tuples in the *works* relation for employees of “Small Bank Corporation”.
- 3.18** Give an SQL schema definition for the employee database of Figure 3.19. Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema. Include any foreign-key constraints that might be appropriate.
- 3.19** List two reasons why null values might be introduced into the database.
- 3.20** Show that, in SQL, `<>` **all** is identical to **not in**.

member(memb_no, name)
book(isbn, title, authors, publisher)
borrowed(memb_no, isbn, date)

Figure 3.20 Library database.

- 3.21** Consider the library database of Figure 3.20. Write the following queries in SQL.
- Find the member number and name of each member who has borrowed at least one book published by “McGraw-Hill”.
 - Find the member number and name of each member who has borrowed every book published by “McGraw-Hill”.
 - For each publisher, find the member number and name of each member who has borrowed more than five books of that publisher.
 - Find the average number of books borrowed per member. Take into account that if a member does not borrow any books, then that member does not appear in the *borrowed* relation at all, but that member still counts in the average.

- 3.22** Rewrite the **where** clause

where unique (select title from course)

without using the **unique** construct.

- 3.23** Consider the query:

```

with dept_total (dept_name, value) as
  (select dept_name, sum(salary)
   from instructor
   group by dept_name),
 dept_total_avg(value) as
  (select avg(value)
   from dept_total)
 select dept_name
   from dept_total, dept_total_avg
  where dept_total.value >= dept_total_avg.value;
```

Rewrite this query without using the **with** construct.

- 3.24** Using the university schema, write an SQL query to find the name and ID of those Accounting students advised by an instructor in the Physics department.

- 3.25 Using the university schema, write an SQL query to find the names of those departments whose budget is higher than that of Philosophy. List them in alphabetic order.
- 3.26 Using the university schema, use SQL to do the following: For each student who has retaken a course at least twice (i.e., the student has taken the course at least three times), show the course ID and the student's ID.
Please display your results in order of course ID and do not display duplicate rows.
- 3.27 Using the university schema, write an SQL query to find the IDs of those students who have retaken at least three distinct courses at least once (i.e., the student has taken the course at least two times).
- 3.28 Using the university schema, write an SQL query to find the names and IDs of those instructors who teach every course taught in his or her department (i.e., every course that appears in the *course* relation with the instructor's department name). Order result by name.
- 3.29 Using the university schema, write an SQL query to find the name and ID of each History student whose name begins with the letter 'D' and who has *not* taken at least five Music courses.
- 3.30 Consider the following SQL query on the university schema:

```
select avg(salary) - (sum(salary) / count(*))
from instructor
```

We might expect that the result of this query is zero since the average of a set of numbers is defined to be the sum of the numbers divided by the number of numbers. Indeed this is true for the example *instructor* relation in Figure 2.1. However, there are other possible instances of that relation for which the result would *not* be zero. Give one such instance, and explain why the result would not be zero.

- 3.31 Using the university schema, write an SQL query to find the ID and name of each instructor who has never given an A grade in any course she or he has taught. (Instructors who have never taught a course trivially satisfy this condition.)
- 3.32 Rewrite the preceding query, but also ensure that you include only instructors who have given at least one other non-null grade in some course.
- 3.33 Using the university schema, write an SQL query to find the ID and title of each course in Comp. Sci. that has had at least one section with afternoon hours (i.e., ends at or after 12:00). (You should eliminate duplicates if any.)
- 3.34 Using the university schema, write an SQL query to find the number of students in each section. The result columns should appear in the order "courseid, secid, year, semester, num". You do not need to output sections with 0 students.

- 3.35** Using the university schema, write an SQL query to find section(s) with maximum enrollment. The result columns should appear in the order “courseid, secid, year, semester, num”. (It may be convenient to use the *with* construct.)

Tools

A number of relational database systems are available commercially, including IBM DB2, IBM Informix, Oracle, SAP Adaptive Server Enterprise (formerly Sybase), and Microsoft SQL Server. In addition several open-source database systems can be downloaded and used free of charge, including PostgreSQL and MySQL (free except for certain kinds of commercial use). Some commercial vendors offer free versions of their systems with certain use limitations. These include Oracle Express edition, Microsoft SQL Server Express, and IBM DB2 Express-C.

The sql.js database is version of the embedded SQL database SQLite which can be run directly in a web browser, allowing SQL commands to be executed directly in the browser. All data are temporary and vanishes when you close the browser, but it can be useful for learning SQL; be warned that the subset of SQL that is supported by sql.js and SQLite is considerably smaller than what is supported by other databases. An SQL tutorial using sql.js as the execution engine is hosted at www.w3schools.com/sql.

The web site of our book, db-book.com, provides a significant amount of supporting material for the book. By following the link on the site titled Laboratory Material, you can get access to the following:

- Instructions on how to set up and access some popular database systems, including sql.js (which you can run in your browser), MySQL, and PostgreSQL.
- SQL schema definitions for the University schema.
- SQL scripts for loading sample datasets.
- Tips on how to use the XData system, developed at IIT Bombay, to test queries for correctness by executing them on multiple datasets generated by the system; and, for instructors, tips on how to use XData to automate SQL query grading.
- Get tips on SQL variations across different databases.

Support for different SQL features varies by databases, and most databases also support some non-standard extensions to SQL. Read the system manuals to understand the exact SQL features that a database supports.

Most database systems provide a command line interface for submitting SQL commands. In addition, most databases also provide graphical user interfaces (GUIs), which simplify the task of browsing the database, creating and submitting queries, and administering the database. For PostgreSQL, the pgAdmin tool provides GUI functionality, while for MySQL, phpMyAdmin provides GUI functionality. Oracle provides

Oracle SQL Developer, while Microsoft SQL Server comes with the SQL Server Management Studio.

The NetBeans IDEs SQLEditor provides a GUI front end which works with a number of different database systems, but with limited functionality, while the Eclipse IDE supports similar functionality through the Data Tools Platform (DTP). Commercial IDEs that support SQL access across multiple database platforms include Embarcadero's RAD Studio and Aqua Data Studio.

Further Reading

The original Sequel language that became SQL is described in [Chamberlin et al. (1976)].

The most important SQL reference is likely to be the online documentation provided by the vendor or the particular database system you are using. That documentation will identify any features that deviate from the SQL standard features presented in this chapter. Here are links to the SQL reference manuals for the current (as of 2018) versions of some of the popular databases.

- MySQL 8.0: dev.mysql.com/doc/refman/8.0/en/
- Oracle 12c: docs.oracle.com/database/121/SQLRF/
- PostgreSQL: www.postgresql.org/docs/current/static/sql.html
- SQLite: www.sqlite.org/lang.html
- SQL Server: docs.microsoft.com/en-us/sql/t-sql

Bibliography

[Chamberlin et al. (1976)] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", *IBM Journal of Research and Development*, Volume 20, Number 6 (1976), pages 560–575.

Credits

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

CHAPTER 3



Introduction to SQL

Practice Exercises

- 3.1 Write the following queries in SQL, using the university schema. (We suggest you actually run these queries on a database, using the sample data that we provide on the web site of the book, db-book.com. Instructions for setting up a database, and loading sample data, are provided on the above web site.)
- Find the titles of courses in the Comp. Sci. department that have 3 credits.
 - Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.
 - Find the highest salary of any instructor.
 - Find all instructors earning the highest salary (there may be more than one with the same salary).
 - Find the enrollment of each section that was offered in Fall 2017.
 - Find the maximum enrollment, across all sections, in Fall 2017.
 - Find the sections that had the maximum enrollment in Fall 2017.

Answer:

- Find the titles of courses in the Comp. Sci. department that have 3 credits.

```
select    title
from      course
where     dept_name = 'Comp. Sci.' and credits = 3
```

- Find the IDs of all students who were taught by an instructor named Einstein; make sure there are no duplicates in the result.
This query can be answered in several different ways. One way is as follows.

```
select distinct takes.ID
from takes, instructor, teaches
where takes.course_id = teaches.course_id and
takes.sec_id = teaches.sec_id and
takes.semester = teaches.semester and
takes.year = teaches.year and
teaches.id = instructor.id and
instructor.name = 'Einstein'
```

- c. Find the highest salary of any instructor.

```
select max(salary)
from instructor
```

- d. Find all instructors earning the highest salary (there may be more than one with the same salary).

```
select ID, name
from instructor
where salary = (select max(salary) from instructor)
```

- e. Find the enrollment of each section that was offered in Fall 2017.

```
select course_id, sec_id,
(select count(ID)
from takes
where takes.year = section.year
and takes.semester = section.semester
and takes.course_id = section.course_id
and takes.sec_id = section.sec_id)
as enrollment
from section
where semester = 'Fall'
and year = 2017
```

Note that if the result of the subquery is empty, the aggregate function **count** returns a value of 0.

One way of writing the query might appear to be:

```

select      takes.course_id, takes.sec_id, count(ID)
from        section, takes
where       takes.course_id = section.course_id
              and takes.sec_id = section.sec_id
              and takes.semester = section.semester
              and takes.year = section.year
              and takes.semester = 'Fall'
              and takes.year = 2017
group by    takes.course_id, takes.sec_id

```

But note that if a section does not have any students taking it, it would not appear in the result. One way of ensuring such a section appears with a count of 0 is to use the **outer join** operation, covered in Chapter 4.

- f. Find the maximum enrollment, across all sections, in Fall 2017.
One way of writing this query is as follows:

```

select max(enrollment)
from  (select count(ID) as enrollment
         from   section, takes
         where  takes.year = section.year
                 and takes.semester = section.semester
                 and takes.course_id = section.course_id
                 and takes.sec_id = section.sec_id
                 and takes.semester = 'Fall'
                 and takes.year = 2017
         group by takes.course_id, takes.sec_id)

```

As an alternative to using a nested subquery in the **from** clause, it is possible to use a **with** clause, as illustrated in the answer to the next part of this question.

A subtle issue in the above query is that if no section had any enrollment, the answer would be empty, not 0. We can use the alternative using a subquery, from the previous part of this question, to ensure the count is 0 in this case.

- g. Find the sections that had the maximum enrollment in Fall 2017.
The following answer uses a **with** clause, simplifying the query.

```

with sec_enrollment as (
    select takes.course_id, takes.sec_id, count(ID) as enrollment
    from section, takes
    where takes.year = section.year
        and takes.semester = section.semester
        and takes.course_id = section.course_id
        and takes.sec_id = section.sec_id
        and takes.semester = 'Fall'
        and takes.year = 2017
    group by takes.course_id, takes.sec_id)
select course_id, sec_id
from sec_enrollment
where enrollment = (select max(enrollment) from sec_enrollment)

```

It is also possible to write the query without the **with** clause, but the subquery to find enrollment would get repeated twice in the query.

While not incorrect to add **distinct** in the **count**, it is not necessary in light of the primary key constraint on *takes*.

-  3.2 Suppose you are given a relation *grade_points(grade, points)* that provides a conversion from letter grades in the *takes* relation to numeric scores; for example, an "A" grade could be specified to correspond to 4 points, an "A−" to 3.7 points, a "B+" to 3.3 points, a "B" to 3 points, and so on. The grade points earned by a student for a course offering (section) is defined as the number of credits for the course multiplied by the numeric points for the grade that the student received.

 Given the preceding relation, and our university schema, write each of the following queries in SQL. You may assume for simplicity that no *takes* tuple has the *null* value for *grade*.

- Find the total grade points earned by the student with ID '12345', across all courses taken by the student.
- Find the grade point average (*GPA*) for the above student, that is, the total grade points divided by the total credits for the associated courses.
- Find the ID and the grade-point average of each student.
- Now reconsider your answers to the earlier parts of this exercise under the assumption that some grades might be null. Explain whether your solutions still work and, if not, provide versions that handle nulls properly.

Answer:

- Find the total grade-points earned by the student with ID '12345', across all courses taken by the student.

```

select sum(credits * points)
from takes, course, grade_points
where takes.grade = grade_points.grade
      and takes.course_id = course.course_id
      and ID = '12345'

```

In the above query, a student who has not taken any course would not have any tuples, whereas we would expect to get 0 as the answer. One way of fixing this problem is to use the **outer join** operation, which we study later in Chapter 4. Another way to ensure that we get 0 as the answer is via the following query:

```

(select sum(credits * points)
from takes, course, grade_points
where takes.grade = grade_points.grade
      and takes.course_id = course.course_id
      and ID= '12345')
union
(select 0
from student
where ID = '12345' and
      not exists ( select * from takes where ID = '12345') )

```

- b. Find the grade point average (*GPA*) for the above student, that is, the total grade-points divided by the total credits for the associated courses.

```

select sum(credits * points)/sum(credits) as GPA
from takes, course, grade_points
where takes.grade = grade_points.grade
      and takes.course_id = course.course_id
      and ID= '12345'

```

As before, a student who has not taken any course would not appear in the above result; we can ensure that such a student appears in the result by using the modified query from the previous part of this question. However, an additional issue in this case is that the sum of credits would also be 0, resulting in a divide-by-zero condition. In fact, the only meaningful way of defining the *GPA* in this case is to define it as *null*. We can ensure that such a student appears in the result with a null *GPA* by adding the following **union** clause to the above query.

```

union
(select null as GPA
from student
where ID = '12345' and
      not exists ( select * from takes where ID = '12345') )

```

- c. Find the ID and the grade-point average of each student.

```
select ID, sum(credits * points)/sum(credits) as GPA
from takes, course, grade_points
where takes.grade = grade_points.grade
and takes.course_id = course.course_id
group by ID
```

Again, to handle students who have not taken any course, we would have to add the following **union** clause:

```
union
(select ID, null as GPA
from student
where not exists ( select * from takes where takes.ID = student.ID))
```

- d. Now reconsider your answers to the earlier parts of this exercise under the assumption that some grades might be null. Explain whether your solutions still work and, if not, provide versions that handle nulls properly. The queries listed above all include a test of equality on *grade* between *grade_points* and *takes*. Thus, for any *takes* tuple with a *null* grade, that student's course would be eliminated from the rest of the computation of the result. As a result, the credits of such courses would be eliminated also, and thus the queries would return the correct answer even if some grades are null.

- 3.3** Write the following inserts, deletes, or updates in SQL, using the university schema.

- Increase the salary of each instructor in the Comp. Sci. department by 10%.
- Delete all courses that have never been offered (i.e., do not occur in the *section* relation).
- Insert every student whose *tot_cred* attribute is greater than 100 as an instructor in the same department, with a salary of \$10,000.

Answer:

- Increase the salary of each instructor in the Comp. Sci. department by 10%.

```
update instructor
set salary = salary * 1.10
where dept_name = 'Comp. Sci.'
```

- Delete all courses that have never been offered (that is, do not occur in the *section* relation).

```

person (driver_id, name, address)
car (license_plate, model, year)
accident (report_number, year, location)
owns (driver_id, license_plate)
participated (report_number, license_plate, driver_id, damage_amount)

```

Figure 3.17 Insurance database

```

delete from course
where course_id not in
    (select course_id from section)

```

- c. Insert every student whose *tot_cred* attribute is greater than 100 as an instructor in the same department, with a salary of \$10,000.

```

insert into instructor
select ID, name, dept_name, 10000
from student
where tot_cred > 100

```

- 3.4** Consider the insurance database of Figure 3.17, where the primary keys are underlined. Construct the following SQL queries for this relational database.

- Find the total number of people who owned cars that were involved in accidents in 2017.
- Delete all year-2010 cars belonging to the person whose ID is '12345'.

Answer:

- Find the total number of people who owned cars that were involved in accidents in 2017.

Note: This is not the same as the total number of accidents in 2017. We must count people with several accidents only once. Furthermore, note that the question asks for owners, and it might be that the owner of the car was not the driver actually involved in the accident.

```

select count (distinct person.driver_id)
from accident, participated, person, owns
where accident.report_number = participated.report_number
and owns.driver_id = person.driver_id
and owns.license_plate = participated.license_plate
and year = 2017

```

- b. Delete all year-2010 cars belonging to the person whose ID is '12345'.

```
delete car
where year = 2010 and license_plate in
  (select license_plate
   from owns o
   where o.driver_id = '12345')
```

Note: The *owns*, *accident* and *participated* records associated with the deleted cars still exist.

- 3.5** Suppose that we have a relation *marks*(*ID*, *score*) and we wish to assign grades to students based on the score as follows: grade *F* if *score* < 40, grade *C* if $40 \leq \text{score} < 60$, grade *B* if $60 \leq \text{score} < 80$, and grade *A* if $80 \leq \text{score}$. Write SQL queries to do the following:

- a. Display the grade for each student, based on the *marks* relation.
- b. Find the number of students with each grade.

Answer:

- a. Display the grade for each student, based on the *marks* relation.

```
select ID,
  case
    when score < 40 then 'F'
    when score < 60 then 'C'
    when score < 80 then 'B'
    else 'A'
  end
from marks
```

- b. Find the number of students with each grade.

```

with      grades as
(
select    ID,
case
when score < 40 then 'F'
when score < 60 then 'C'
when score < 80 then 'B'
else 'A'
end as grade
from      marks
)
select    grade, count(ID)
from      grades
group by grade

```

As an alternative, the **with** clause can be removed, and instead the definition of *grades* can be made a subquery of the main query.

- 3.6** The SQL **like** operator is case sensitive (in most systems), but the **lower()** function on strings can be used to perform case-insensitive matching. To show how, write a query that finds departments whose names contain the string “sci” as a substring, regardless of the case.

Answer:

```

select dept_name
from department
where lower(dept_name) like '%sci%'

```

- 3.7** Consider the SQL query

```

select p.a1
from p, r1, r2
where p.a1 = r1.a1 or p.a1 = r2.a1

```

Under what conditions does the preceding query select values of *p.a1* that are either in *r1* or in *r2*? Examine carefully the cases where either *r1* or *r2* may be empty.

Answer:

The query selects those values of *p.a1* that are equal to some value of *r1.a1* or *r2.a1* if and only if both *r1* and *r2* are non-empty. If one or both of *r1* and *r2* are empty, the Cartesian product of *p*, *r1* and *r2* is empty, hence the result of the query is empty. If *p* itself is empty, the result is empty.

- 3.8** Consider the bank database of Figure 3.18, where the primary keys are underlined. Construct the following SQL queries for this relational database.

```

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 3.18 Banking database.

- Find the ID of each customer of the bank who has an account but not a loan.
- Find the ID of each customer who lives on the same street and in the same city as customer '12345'.
- Find the name of each branch that has at least one customer who has an account in the bank and who lives in "Harrison".

Answer:

- Find the ID of each customer of the bank who has an account but not a loan.

```

(select ID
from depositor)
except
(select ID
from borrower)

```

- Find the ID of each customer who lives on the same street and in the same city as customer '12345'.

```

select F.ID
from customer as F, customer as S
where F.customer_street = S.customer_street
and F.customer_city = S.customer_city
and S.customer_id = '12345'

```

- Find the name of each branch that has at least one customer who has an account in the bank and who lives in "Harrison".

```

select distinct branch_name
from account, depositor, customer
where customer.id = depositor.id
        and depositor.account_number = account.account_number
        and customer.city = 'Harrison'

```

- 3.9** Consider the relational database of Figure 3.19, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

- a. Find the ID, name, and city of residence of each employee who works for “First Bank Corporation”.
- b. Find the ID, name, and city of residence of each employee who works for “First Bank Corporation” and earns more than \$10000.
- c. Find the ID of each employee who does not work for “First Bank Corporation”.
- d. Find the ID of each employee who earns more than every employee of “Small Bank Corporation”.
- e. Assume that companies may be located in several cities. Find the name of each company that is located in every city in which “Small Bank Corporation” is located.
- f. Find the name of the company that has the most employees (or companies, in the case where there is a tie for the most).
- g. Find the name of each company whose employees earn a higher salary, on average, than the average salary at “First Bank Corporation”.

Answer:

- a. Find the ID, name, and city of residence of each employee who works for “First Bank Corporation”.

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

Figure 3.19 Employee database.

```
select e.ID, e.person_name, city
from employee as e, works as w
where w.company_name = 'First Bank Corporation' and
      w.ID = e.ID
```

- b. Find the ID, name, and city of residence of each employee who works for “First Bank Corporation” and earns more than \$10000.

```
select *
from employee
where ID in
  (select ID
   from works
   where company_name = 'First Bank Corporation' and salary > 10000)
```

This could be written also in the style of the answer to part a.

- c. Find the ID of each employee who does not work for “First Bank Corporation”.

```
select ID
from works
where company_name <> 'First Bank Corporation'
```

If one allows people to appear in *employee* without appearing also in *works*, the solution is slightly more complicated. An outer join as discussed in Chapter 4 could be used as well.

```
select ID
from employee
where ID not in
  (select ID
   from works
   where company_name = 'First Bank Corporation')
```

- d. Find the ID of each employee who earns more than every employee of “Small Bank Corporation”.

```
select ID
from works
where salary > all
  (select salary
   from works
   where company_name = 'Small Bank Corporation')
```

If people may work for several companies and we wish to consider the *total* earnings of each person, the problem is more complex. But note that the

fact that ID is the primary key for *works* implies that this cannot be the case.

- e. Assume that companies may be located in several cities. Find the name of each company that is located in every city in which “Small Bank Corporation” is located.

```
select S.company_name
from company as S
where not exists ((select city
    from company
    where company_name = 'Small Bank Corporation')
except
    (select city
    from company as T
    where S.company_name = T.company_name))
```

- f. Find the name of the company that has the most employees (or companies, in the case where there is a tie for the most).

```
select company_name
from works
group by company_name
having count (distinct ID) >= all
    (select count (distinct ID)
    from works
    group by company_name)
```

- g. Find the name of each company whose employees earn a higher salary, on average, than the average salary at “First Bank Corporation”.

```
select company_name
from works
group by company_name
having avg (salary) > (select avg (salary)
    from works
    where company_name = 'First Bank Corporation')
```

- 3.10** Consider the relational database of Figure 3.19. Give an expression in SQL for each of the following:

- a. Modify the database so that the employee whose ID is '12345' now lives in “Newtown”.
- b. Give each manager of “First Bank Corporation” a 10 percent raise unless the salary becomes greater than \$100000; in such cases, give only a 3 percent raise.

Answer:

- a. Modify the database so that the employee whose ID is '12345' now lives in "Newtown".

```
update employee
set city = 'Newtown'
where ID = '12345'
```

- b. Give each manager of "First Bank Corporation" a 10 percent raise unless the salary becomes greater than \$100000; in such cases, give only a 3 percent raise.

```
update works T
set T.salary = T.salary * 1.03
where T.ID in (select manager_id
                from manages)
                and T.salary * 1.1 > 100000
                and T.company_name = 'First Bank Corporation'
```

```
update works T
set T.salary = T.salary * 1.1
where T.ID in (select manager_id
                from manages)
                and T.salary * 1.1 <= 100000
                and T.company_name = 'First Bank Corporation'
```

The above updates would give different results if executed in the opposite order. We give below a safer solution using the **case** statement.

```
update works T
set T.salary = T.salary *
(case
    when (T.salary * 1.1 > 100000) then 1.03
    else 1.1
end)
where T.ID in (select manager_id
                from manages) and
                T.company_name = 'First Bank Corporation'
```

CHAPTER 4



Intermediate SQL

In this chapter, we continue our study of SQL. We consider more complex forms of SQL queries, view definition, transactions, integrity constraints, more details regarding SQL data definition, and authorization.

4.1 Join Expressions

In all of the example queries we used in Chapter 3 (except when we used set operations), we combined information from multiple relations using the Cartesian product operator. In this section, we introduce a number of “join” operations that allow the programmer to write some queries in a more natural way and to express some queries that are difficult to do with only the Cartesian product.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Figure 4.1 The *student* relation.

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	<i>null</i>

Figure 4.2 The *takes* relation.

All the examples used in this section involve the two relations *student* and *takes*, shown in Figure 4.1 and Figure 4.2, respectively. Observe that the attribute *grade* has a value *null* for the student with *ID* 98988, for the course BIO-301, section 1, taken in Summer 2018. The null value indicates that the grade has not been awarded yet.

4.1.1 The Natural Join

Consider the following SQL query, which computes for each student the set of courses a student has taken:

```
select name, course_id
  from student, takes
 where student.ID = takes.ID;
```

Note that this query outputs only students who have taken some course. Students who have not taken any course are not output.

Note that in the *student* and *takes* table, the matching condition required *student.ID* to be equal to *takes.ID*. These are the only attributes in the two relations that have the same name. In fact, this is a common case; that is, the matching condition in the **from** clause most often requires all attributes with matching names to be equated.

To make the life of an SQL programmer easier for this common case, SQL supports an operation called the *natural join*, which we describe below. In fact, SQL supports several other ways in which information from two or more relations can be **joined** together. We have already seen how a Cartesian product along with a **where** clause predicate can be used to join information from multiple relations. Other ways of joining information from multiple relations are discussed in Section 4.1.2 through Section 4.1.4.

The **natural join** operation operates on two relations and produces a relation as the result. Unlike the Cartesian product of two relations, which concatenates each tuple of the first relation with every tuple of the second, natural join considers only those pairs of tuples with the same value on those attributes that appear in the schemas of both relations. So, going back to the example of the relations *student* and *takes*, computing:

```
student natural join takes
```

considers only those pairs of tuples where both the tuple from *student* and the tuple from *takes* have the same value on the common attribute, *ID*.

The resulting relation, shown in Figure 4.3, has only 22 tuples, the ones that give information about a student and a course that the student has actually taken. Notice that we do not repeat those attributes that appear in the schemas of both relations; rather they appear only once. Notice also the order in which the attributes are listed: first the attributes common to the schemas of both relations, second those attributes unique to the schema of the first relation, and finally, those attributes unique to the schema of the second relation.

Earlier we wrote the query “For all students in the university who have taken some course, find their names and the course ID of all courses they took” as:

```
select name, course_id  
from student, takes  
where student.ID = takes.ID;
```

This query can be written more concisely using the natural-join operation in SQL as:

```
select name, course_id  
from student natural join takes;
```

Both of the above queries generate the same result.¹

¹For notational symmetry, SQL allows the Cartesian product, which we have denoted with a comma, to be denoted by the keywords **cross join**. Thus, “**from student, takes**” could be expressed equivalently as “**from student cross join takes**”.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	<i>null</i>

Figure 4.3 The natural join of the *student* relation with the *takes* relation.

The result of the natural join operation is a relation. Conceptually, expression “*student natural join takes*” in the **from** clause is replaced by the relation obtained by evaluating the natural join.² The **where** and **select** clauses are then evaluated on this relation, as we saw in Section 3.3.2.

A **from** clause in an SQL query can have multiple relations combined using natural join, as shown here:

```
select A1, A2, ..., An
  from r1 natural join r2 natural join ... natural join rm
    where P;
```

More generally, a **from** clause can be of the form

²As a consequence, it may not be possible in some systems to use attribute names containing the original relation names, for instance, *student.ID* or *takes.ID*, to refer to attributes in the natural join result. While some systems allow it, others don't, and some allow it for all attributes except the join attributes (i.e., those that appear in both relation schemas). We can, however, use attribute names such as *name* and *course_id* without the relation names.

from E_1, E_2, \dots, E_n

where each E_i can be a single relation or an expression involving natural joins. For example, suppose we wish to answer the query “List the names of students along with the titles of courses that they have taken.” The query can be written in SQL as follows:

```
select name, title
from student natural join takes, course
where takes.course_id = course.course_id;
```

The natural join of *student* and *takes* is first computed, as we saw earlier, and a Cartesian product of this result with *course* is computed, from which the **where** clause extracts only those tuples where the course identifier from the join result matches the course identifier from the *course* relation. Note that *takes.course_id* in the **where** clause refers to the *course_id* field of the natural join result, since this field, in turn, came from the *takes* relation.

In contrast, the following SQL query does *not* compute the same result:

```
select name, title
from student natural join takes natural join course;
```

To see why, note that the natural join of *student* and *takes* contains the attributes (*ID*, *name*, *dept_name*, *tot_cred*, *course_id*, *sec_id*), while the *course* relation contains the attributes (*course_id*, *title*, *dept_name*, *credits*). As a result, the natural join would require that the *dept_name* attribute values from the two relations be the same in addition to requiring that the *course_id* values be the same. This query would then omit all (student name, course title) pairs where the student takes a course in a department other than the student’s own department. The previous query, on the other hand, correctly outputs such pairs.

To provide the benefit of natural join while avoiding the danger of equating attributes erroneously, SQL provides a form of the natural join construct that allows you to specify exactly which columns should be equated. This feature is illustrated by the following query:

```
select name, title
from (student natural join takes) join course using (course_id);
```

The operation **join ... using** requires a list of attribute names to be specified. Both relations being joined must have attributes with the specified names. Consider the operation $r_1 \text{ join } r_2 \text{ using}(A_1, A_2)$. The operation is similar to $r_1 \text{ natural join } r_2$, except that a pair of tuples t_1 from r_1 and t_2 from r_2 match if $t_1.A_1 = t_2.A_1$ and $t_1.A_2 = t_2.A_2$; even if r_1 and r_2 both have an attribute named A_3 , it is *not* required that $t_1.A_3 = t_2.A_3$.

Thus, in the preceding SQL query, the **join** construct permits *student.dept_name* and *course.dept_name* to differ, and the SQL query gives the correct answer.

4.1.2 Join Conditions

In Section 4.1.1, we saw how to express natural joins, and we saw the **join ... using** clause, which is a form of natural join that requires values to match only on specified attributes. SQL supports another form of join, in which an arbitrary join condition can be specified.

The **on** condition allows a general predicate over the relations being joined. This predicate is written like a **where** clause predicate except for the use of the keyword **on** rather than **where**. Like the **using** condition, the **on** condition appears at the end of the join expression.

Consider the following query, which has a join expression containing the **on** condition:

```
select *
from student join takes on student.ID = takes.ID;
```

The **on** condition above specifies that a tuple from *student* matches a tuple from *takes* if their *ID* values are equal. The join expression in this case is almost the same as the join expression *student natural join takes*, since the natural join operation also requires that for a *student* tuple and a *takes* tuple to match. The one difference is that the result has the *ID* attribute listed twice, in the join result, once for *student* and once for *takes*, even though their *ID* values must be the same.

In fact, the preceding query is equivalent to the following query:

```
select *
from student, takes
where student.ID = takes.ID;
```

As we have seen earlier, the relation name is used to disambiguate the attribute name *ID*, and thus the two occurrences can be referred to as *student.ID* and *takes.ID*, respectively. A version of this query that displays the *ID* value only once is as follows:

```
select student.ID as ID, name, dept_name, tot_cred,
        course_id, sec_id, semester, year, grade
from student join takes on student.ID = takes.ID;
```

The result of this query is exactly the same as the result of the natural join of *student* and *takes*, which we showed in Figure 4.3.

The **on** condition can express any SQL predicate, and thus join expressions using the **on** condition can express a richer class of join conditions than **natural join**. However,

as illustrated by our preceding example, a query using a join expression with an **on** condition can be replaced by an equivalent expression without the **on** condition, with the predicate in the **on** clause moved to the **where** clause. Thus, it may appear that the **on** condition is a redundant feature of SQL.

However, there are two good reasons for introducing the **on** condition. First, we shall see shortly that for a kind of join called an outer join, **on** conditions do behave in a manner different from **where** conditions. Second, an SQL query is often more readable by humans if the join condition is specified in the **on** clause and the rest of the conditions appear in the **where** clause.

4.1.3 Outer Joins

Suppose we wish to display a list of all students, displaying their *ID*, and *name*, *dept_name*, and *tot_cred*, along with the courses that they have taken. The following SQL query may appear to retrieve the required information:

```
select *
from student natural join takes;
```

Unfortunately, the above query does not work quite as intended. Suppose that there is some student who takes no courses. Then the tuple in the *student* relation for that particular student would not satisfy the condition of a natural join with any tuple in the *takes* relation, and that student's data would not appear in the result. We would thus not see any information about students who have not taken any courses. For example, in the *student* and *takes* relations of Figure 4.1 and Figure 4.2, note that student Snow, with ID 70557, has not taken any courses. Snow appears in *student*, but Snow's ID number does not appear in the *ID* column of *takes*. Thus, Snow does not appear in the result of the natural join.

More generally, some tuples in either or both of the relations being joined may be “lost” in this way. The **outer-join** operation works in a manner similar to the join operations we have already studied, but it preserves those tuples that would be lost in a join by creating tuples in the result containing null values.

For example, to ensure that the student named Snow from our earlier example appears in the result, a tuple could be added to the join result with all attributes from the *student* relation set to the corresponding values for the student Snow, and all the remaining attributes which come from the *takes* relation, namely, *course_id*, *sec_id*, *semester*, and *year*, set to *null*. Thus, the tuple for the student Snow is preserved in the result of the outer join.

There are three forms of outer join:

- The **left outer join** preserves tuples only in the relation named before (to the left of) the **left outer join** operation.

- The **right outer join** preserves tuples only in the relation named after (to the right of) the **right outer join** operation.
- The **full outer join** preserves tuples in both relations.

In contrast, the join operations we studied earlier that do not preserve nonmatched tuples are called **inner-join** operations, to distinguish them from the outer-join operations.

We now explain exactly how each form of outer join operates. We can compute the left outer-join operation as follows: First, compute the result of the inner join as before. Then, for every tuple t in the left-hand-side relation that does not match any tuple in the right-hand-side relation in the inner join, add a tuple r to the result of the join constructed as follows:

- The attributes of tuple r that are derived from the left-hand-side relation are filled in with the values from tuple t .
- The remaining attributes of r are filled with null values.

Figure 4.4 shows the result of:

```
select *
from student natural left outer join takes;
```

That result includes student Snow (*ID* 70557), unlike the result of an inner join, but the tuple for Snow includes nulls for the attributes that appear only in the schema of the *takes* relation.³

As another example of the use of the outer-join operation, we can write the query “Find all students who have not taken a course” as:

```
select ID
from student natural left outer join takes
where course_id is null;
```

The **right outer join** is symmetric to the **left outer join**. Tuples from the right-hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join. Thus, if we rewrite the preceding query using a right outer join and swapping the order in which we list the relations as follows:

```
select *
from takes natural right outer join student;
```

we get the same result except for the order in which the attributes appear in the result (see Figure 4.5).

³We show null values in tables using *null*, but most systems display null values as a blank field.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
70557	Snow	Physics	0	null	null	null	null	null
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	null

Figure 4.4 Result of *student* natural left outer join *takes*.

The **full outer join** is a combination of the left and right outer-join types. After the operation computes the result of the inner join, it extends with nulls those tuples from the left-hand-side relation that did not match with any from the right-hand-side relation and adds them to the result. Similarly, it extends with nulls those tuples from the right-hand-side relation that did not match with any tuples from the left-hand-side relation and adds them to the result. Said differently, full outer join is the union of a left outer join and the corresponding right outer join.⁴

As an example of the use of full outer join, consider the following query: “Display a list of all students in the Comp. Sci. department, along with the course sections, if any, that they have taken in Spring 2017; all course sections from Spring 2017 must

⁴In those systems, notably MySQL, that implement only left and right outer join, this is exactly how one has to write a full outer join.

<i>ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>
00128	CS-101	1	Fall	2017	A	Zhang	Comp. Sci.	102
00128	CS-347	1	Fall	2017	A-	Zhang	Comp. Sci.	102
12345	CS-101	1	Fall	2017	C	Shankar	Comp. Sci.	32
12345	CS-190	2	Spring	2017	A	Shankar	Comp. Sci.	32
12345	CS-315	1	Spring	2018	A	Shankar	Comp. Sci.	32
12345	CS-347	1	Fall	2017	A	Shankar	Comp. Sci.	32
19991	HIS-351	1	Spring	2018	B	Brandt	History	80
23121	FIN-201	1	Spring	2018	C+	Chavez	Finance	110
44553	PHY-101	1	Fall	2017	B-	Peltier	Physics	56
45678	CS-101	1	Fall	2017	F	Levy	Physics	46
45678	CS-101	1	Spring	2018	B+	Levy	Physics	46
45678	CS-319	1	Spring	2018	B	Levy	Physics	46
54321	CS-101	1	Fall	2017	A-	Williams	Comp. Sci.	54
54321	CS-190	2	Spring	2017	B+	Williams	Comp. Sci.	54
55739	MU-199	1	Spring	2018	A-	Sanchez	Music	38
70557	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	Snow	Physics	0
76543	CS-101	1	Fall	2017	A	Brown	Comp. Sci.	58
76543	CS-319	2	Spring	2018	A	Brown	Comp. Sci.	58
76653	EE-181	1	Spring	2017	C	Aoi	Elec. Eng.	60
98765	CS-101	1	Fall	2017	C-	Bourikas	Elec. Eng.	98
98765	CS-315	1	Spring	2018	B	Bourikas	Elec. Eng.	98
98988	BIO-101	1	Summer	2017	A	Tanaka	Biology	120
98988	BIO-301	1	Summer	2018	<i>null</i>	Tanaka	Biology	120

Figure 4.5 The result of *takes* natural right outer join *student*.

be displayed, even if no student from the Comp. Sci. department has taken the course section.” This query can be written as:

```
select *
from (select *
       from student
      where dept_name = 'Comp. Sci.')
natural full outer join
(select *
   from takes
  where semester = 'Spring' and year = 2017);
```

The result appears in Figure 4.6.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>tot_cred</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>	<i>grade</i>
00128	Zhang	Comp. Sci.	102	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
76543	Brown	Comp. Sci.	58	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>	<i>null</i>
76653	<i>null</i>	<i>null</i>	<i>null</i>	ECE-181	1	Spring	2017	C

Figure 4.6 Result of full outer join example (see text).

The **on** clause can be used with outer joins. The following query is identical to the first query we saw using “*student natural left outer join takes*,” except that the attribute *ID* appears twice in the result.

```
select *
from student left outer join takes on student.ID = takes.ID;
```

As we noted earlier, **on** and **where** behave differently for outer join. The reason for this is that outer join adds null-padded tuples only for those tuples that do not contribute to the result of the corresponding “inner” join. The **on** condition is part of the outer join specification, but a **where** clause is not. In our example, the case of the *student* tuple for student “Snow” with ID 70557, illustrates this distinction. Suppose we modify the preceding query by moving the **on** clause predicate to the **where** clause and instead using an **on** condition of *true*.⁵

```
select *
from student left outer join takes on true
where student.ID = takes.ID;
```

The earlier query, using the left outer join with the **on** condition, includes a tuple (70557, Snow, Physics, 0, *null*, *null*, *null*, *null*, *null*) because there is no tuple in *takes* with *ID* = 70557. In the latter query, however, every tuple satisfies the join condition *true*, so no null-padded tuples are generated by the outer join. The outer join actually generates the Cartesian product of the two relations. Since there is no tuple in *takes* with *ID* = 70557, every time a tuple appears in the outer join with *name* = “Snow”, the values for *student.ID* and *takes.ID* must be different, and such tuples would be eliminated by the **where** clause predicate. Thus, student Snow never appears in the result of the latter query.

⁵Some systems do not allow the use of the Boolean constant *true*. To test this on those systems, use a tautology (i.e., a predicate that always evaluates to true), like “1=1”.

Note 4.1 SQL AND MULTISER RELATIONAL ALGEBRA - PART 4

The relational algebra supports the left outer-join operation, denoted by \bowtie_0 , the right outer-join operation, denoted by \bowtie_0 , and the full outer-join operation, denoted by \bowtie_0 . It also supports the natural join operation, denoted by \bowtie , as well as the natural join versions of the left, right and full outer-join operations, denoted by \bowtie , \bowtie_L , and \bowtie_C . The definitions of all these operations are identical to the definitions of the corresponding operations in SQL, which we have seen in Section 4.1.

4.1.4 Join Types and Conditions

To distinguish normal joins from outer joins, normal joins are called **inner joins** in SQL. A join clause can thus specify **inner join** instead of **outer join** to specify that a normal join is to be used. The keyword **inner** is, however, optional. The default join type, when the **join** clause is used without the **outer** prefix, is the **inner join**. Thus,

```
select *
from student join takes using (ID);
```

is equivalent to:

```
select *
from student inner join takes using (ID);
```

Similarly, **natural join** is equivalent to **natural inner join**.

Figure 4.7 shows a full list of the various types of join that we have discussed. As can be seen from the figure, any form of join (inner, left outer, right outer, or full outer) can be combined with any join condition (natural, using, or on).

<i>Join types</i>	<i>Join conditions</i>
inner join left outer join right outer join full outer join	natural on < predicate > using (A ₁ , A ₂ , ..., A _n)

Figure 4.7 Join types and join conditions.

4.2 Views

It is not always desirable for all users to see the entire set of relations in the database. In Section 4.7, we shall see how to use the SQL authorization mechanism to restrict access to relations, but security considerations may require that only certain data in a relation be hidden from a user. Consider a clerk who needs to know an instructor's ID, name, and department name, but does not have authorization to see the instructor's salary amount. This person should see a relation described in SQL by:

```
select ID, name, dept_name
from instructor;
```

Aside from security concerns, we may wish to create a personalized collection of “virtual” relations that is better matched to a certain user’s intuition of the structure of the enterprise. In our university example, we may want to have a list of all course sections offered by the Physics department in the Fall 2017 semester, with the building and room number of each section. The relation that we would create for obtaining such a list is:

```
select course.course_id, sec_id, building, room_number
from course, section
where course.course_id = section.course_id
      and course.dept_name = 'Physics'
      and section.semester = 'Fall'
      and section.year = 2017;
```

It is possible to compute and store the results of these queries and then make the stored relations available to users. However, if we did so, and the underlying data in the relations *instructor*, *course*, or *section* changed, the stored query results would then no longer match the result of reexecuting the query on the relations. In general, it is a bad idea to compute and store query results such as those in the above examples (although there are some exceptions that we study later).

Instead, SQL allows a “virtual relation” to be defined by a query, and the relation conceptually contains the result of the query. The virtual relation is not precomputed and stored but instead is computed by executing the query whenever the virtual relation is used. We saw a feature for this in Section 3.8.6, where we described the **with** clause. The **with** clause allows us to assign a name to a subquery for use as often as desired, but in one particular query only. Here, we present a way to extend this concept beyond a single query by defining a **view**. It is possible to support a large number of views on top of any given set of actual relations.

4.2.1 View Definition

We define a view in SQL by using the **create view** command. To define a view, we must give the view a name and must state the query that computes the view. The form of the **create view** command is:

```
create view v as <query expression>;
```

where **<query expression>** is any legal query expression. The view name is represented by *v*.

Consider again the clerk who needs to access all data in the *instructor* relation, except *salary*. The clerk should not be authorized to access the *instructor* relation (we see in Section 4.7, how authorizations can be specified). Instead, a view relation *faculty* can be made available to the clerk, with the view defined as follows:

```
create view faculty as
  select ID, name, dept_name
  from instructor;
```

As explained earlier, the view relation conceptually contains the tuples in the query result, but it is not precomputed and stored. Instead, the database system stores the query expression associated with the view relation. Whenever the view relation is accessed, its tuples are created by computing the query result. Thus, the view relation is created whenever needed, on demand.

To create a view that lists all course sections offered by the Physics department in the Fall 2017 semester with the building and room number of each section, we write:

```
create view physics_fall_2017 as
  select course.course_id, sec_id, building, room_number
  from course, section
  where course.course_id = section.course_id
        and course.dept_name = 'Physics'
        and section.semester = 'Fall'
        and section.year = 2017;
```

Later, when we study the SQL authorization mechanism in Section 4.7, we shall see that users can be given access to views in place of, or in addition to, access to relations.

Views differ from the **with** statement in that views, once created, remain available until explicitly dropped. The named subquery defined by **with** is local to the query in which it is defined.

4.2.2 Using Views in SQL Queries

Once we have defined a view, we can use the view name to refer to the virtual relation that the view generates. Using the view *physics_fall_2017*, we can find all Physics courses offered in the Fall 2017 semester in the Watson building by writing:

```
select course_id
from physics_fall_2017
where building = 'Watson';
```

View names may appear in a query any place where a relation name may appear,

The attribute names of a view can be specified explicitly as follows:

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum(salary)
        from instructor
        group by dept_name;
```

The preceding view gives for each department the sum of the salaries of all the instructors at that department. Since the expression **sum**(salary) does not have a name, the attribute name is specified explicitly in the view definition.

Intuitively, at any given time, the set of tuples in the view relation is the result of evaluation of the query expression that defines the view. Thus, if a view relation is computed and stored, it may become out of date if the relations used to define it are modified. To avoid this, views are usually implemented as follows: When we define a view, the database system stores the definition of the view itself, rather than the result of evaluation of the query expression that defines the view. Wherever a view relation appears in a query, it is replaced by the stored query expression. Thus, whenever we evaluate the query, the view relation is recomputed.

One view may be used in the expression defining another view. For example, we can define a view *physics_fall_2017_watson* that lists the course ID and room number of all Physics courses offered in the Fall 2017 semester in the Watson building as follows:

```
create view physics_fall_2017_watson as
    select course_id, room_number
        from physics_fall_2017
        where building = 'Watson';
```

where *physics_fall_2017_watson* is itself a view relation. This is equivalent to:

```
create view physics_fall_2017_watson as
    select course_id, room_number
        from (select course.course_id, building, room_number
            from course, section
            where course.course_id = section.course_id
                and course.dept_name = 'Physics'
                and section.semester = 'Fall'
                and section.year = 2017)
            where building = 'Watson';
```

4.2.3 Materialized Views

Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views**.

For example, consider the view *departments_total_salary*. If that view is materialized, its results would be stored in the database, allowing queries that use the view to potentially run much faster by using the precomputed view result, instead of recomputing it.

However, if an *instructor* tuple is added to or deleted from the *instructor* relation, the result of the query defining the view would change, and as a result the materialized view's contents must be updated. Similarly, if an instructor's salary is updated, the tuple in *departments_total_salary* corresponding to that instructor's department must be updated.

The process of keeping the materialized view up-to-date is called **materialized view maintenance** (or often, just **view maintenance**) and is covered in Section 16.5. View maintenance can be done immediately when any of the relations on which the view is defined is updated. Some database systems, however, perform view maintenance lazily, when the view is accessed. Some systems update materialized views only periodically; in this case, the contents of the materialized view may be stale, that is, not up-to-date, when it is used, and it should not be used if the application needs up-to-date data. And some database systems permit the database administrator to control which of the preceding methods is used for each materialized view.

Applications that use a view frequently may benefit if the view is materialized. Applications that demand fast response to certain queries that compute aggregates over large relations can also benefit greatly by creating materialized views corresponding to the queries. In this case, the aggregated result is likely to be much smaller than the large relations on which the view is defined; as a result the materialized view can be used to answer the query very quickly, avoiding reading the large underlying relations. The benefits to queries from the materialization of a view must be weighed against the storage costs and the added overhead for updates.

SQL does not define a standard way of specifying that a view is materialized, but many database systems provide their own SQL extensions for this task. Some database systems always keep materialized views up-to-date when the underlying relations change, while others permit them to become out of date and periodically recompute them.

4.2.4 Update of a View

Although views are a useful tool for queries, they present serious problems if we express updates, insertions, or deletions with them. The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.

Suppose the view *faculty*, which we saw earlier, is made available to a clerk. Since we allow a view name to appear wherever a relation name is allowed, the clerk can write:

```
insert into faculty
    values ('30765', 'Green', 'Music');
```

This insertion must be represented by an insertion into the relation *instructor*, since *instructor* is the actual relation from which the database system constructs the view *faculty*. However, to insert a tuple into *instructor*, we must have some value for *salary*. There are two reasonable approaches to dealing with this insertion:

- Reject the insertion, and return an error message to the user.
- Insert a tuple ('30765', 'Green', 'Music', *null*) into the *instructor* relation.

Another problem with modification of the database through views occurs with a view such as:

```
create view instructor_info as
    select ID, name, building
        from instructor, department
            where instructor.dept_name = department.dept_name;
```

This view lists the *ID*, *name*, and building-name of each instructor in the university. Consider the following insertion through this view:

```
insert into instructor_info
    values ('69987', 'White', 'Taylor');
```

Suppose there is no instructor with ID 69987, and no department in the Taylor building. Then the only possible method of inserting tuples into the *instructor* and *department* relations is to insert ('69987', 'White', *null*, *null*) into *instructor* and (*null*, 'Taylor', *null*) into *department*. Then we obtain the relations shown in Figure 4.8. However, this update does not have the desired effect, since the view relation *instructor_info* still does *not* include the tuple ('69987', 'White', 'Taylor'). Thus, there is no way to update the relations *instructor* and *department* by using nulls to get the desired update on *instructor_info*.

Because of problems such as these, modifications are generally not permitted on view relations, except in limited cases. Different database systems specify different conditions under which they permit updates on view relations; see the database system manuals for details.

In general, an SQL view is said to be **updatable** (i.e., inserts, updates, or deletes can be applied on the view) if the following conditions are all satisfied by the query defining the view:

- The **from** clause has only one database relation.

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
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
69987	White	<i>null</i>	<i>null</i>

instructor

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Biology	Watson	90000
Comp. Sci.	Taylor	100000
Electrical Eng.	Taylor	85000
Finance	Painter	120000
History	Painter	50000
Music	Packard	80000
Physics	Watson	70000
<i>null</i>	Taylor	<i>null</i>

*department***Figure 4.8** Relations *instructor* and *department* after insertion of tuples.

- The **select** clause contains only attribute names of the relation and does not have any expressions, aggregates, or **distinct** specification.
- Any attribute not listed in the **select** clause can be set to *null*; that is, it does not have a **not null** constraint and is not part of a primary key.
- The query does not have a **group by** or **having** clause.

Under these constraints, the **update**, **insert**, and **delete** operations would be allowed on the following view:

```
create view history_instructors as
select *
from instructor
where dept_name = 'History';
```

Even with the conditions on updatability, the following problem still remains. Suppose that a user tries to insert the tuple ('25566', 'Brown', 'Biology', 100000) into the *history_instructors* view. This tuple can be inserted into the *instructor* relation, but it would not appear in the *history_instructors* view since it does not satisfy the selection imposed by the view.

By default, SQL would allow the above update to proceed. However, views can be defined with a **with check option** clause at the end of the view definition; then, if a tuple inserted into the view does not satisfy the view's **where** clause condition, the insertion is rejected by the database system. Updates are similarly rejected if the new value does not satisfy the **where** clause conditions.

SQL:1999 has a more complex set of rules about when inserts, updates, and deletes can be executed on a view that allows updates through a larger class of views; however, the rules are too complex to be discussed here.

An alternative, and often preferable, approach to modifying the database through a view is to use the trigger mechanism discussed in Section 5.3. The **instead of** feature in declaring triggers allows one to replace the default insert, update, and delete operations on a view with actions designed especially for each particular case.

4.3 Transactions

A **transaction** consists of a sequence of query and/or update statements. The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed. One of the following SQL statements must end the transaction:

- **Commit work** commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database. After the transaction is committed, a new transaction is automatically started.
- **Rollback work** causes the current transaction to be rolled back; that is, it undoes all the updates performed by the SQL statements in the transaction. Thus, the database state is restored to what it was before the first statement of the transaction was executed.

The keyword **work** is optional in both the statements.

Transaction rollback is useful if some error condition is detected during execution of a transaction. Commit is similar, in a sense, to saving changes to a document that is being edited, while rollback is similar to quitting the edit session without saving

changes. Once a transaction has executed **commit work**, its effects can no longer be undone by **rollback work**. The database system guarantees that in the event of some failure, such as an error in one of the SQL statements, a power outage, or a system crash, a transaction's effects will be rolled back if it has not yet executed **commit work**. In the case of power outage or other system crash, the rollback occurs when the system restarts.

For instance, consider a banking application where we need to transfer money from one bank account to another in the same bank. To do so, we need to update two account balances, subtracting the amount transferred from one, and adding it to the other. If the system crashes after subtracting the amount from the first account but before adding it to the second account, the bank balances will be inconsistent. A similar problem occurs if the second account is credited before subtracting the amount from the first account and the system crashes just after crediting the amount.

As another example, consider our running example of a university application. We assume that the attribute *tot_cred* of each tuple in the *student* relation is kept up-to-date by modifying it whenever the student successfully completes a course. To do so, whenever the *takes* relation is updated to record successful completion of a course by a student (by assigning an appropriate grade), the corresponding *student* tuple must also be updated. If the application performing these two updates crashes after one update is performed, but before the second one is performed, the data in the database will be inconsistent.

By either committing the actions of a transaction after all its steps are completed, or rolling back all its actions in case the transaction could not complete all its actions successfully, the database provides an abstraction of a transaction as being **atomic**, that is, indivisible. Either all the effects of the transaction are reflected in the database or none are (after rollback).

Applying the notion of transactions to the above applications, the update statements should be executed as a single transaction. An error while a transaction executes one of its statements would result in undoing the effects of the earlier statements of the transaction so that the database is not left in a partially updated state.

If a program terminates without executing either of these commands, the updates are either committed or rolled back. The standard does not specify which of the two happens, and the choice is implementation dependent.

In many SQL implementations, including MySQL and PostgreSQL, by default each SQL statement is taken to be a transaction on its own, and it gets committed as soon as it is executed. Such *automatic commit* of individual SQL statements must be turned off if a transaction consisting of multiple SQL statements needs to be executed. How to turn off automatic commit depends on the specific SQL implementation, although many databases support the command `set autocommit off`.⁶

⁶There is a standard way of turning autocommit on or off when using application program interfaces such as JDBC or ODBC, which we study in Section 5.1.1 and Section 5.1.3, respectively.

A better alternative, which is part of the SQL:1999 standard is to allow multiple SQL statements to be enclosed between the keywords **begin atomic** ... **end**. All the statements between the keywords then form a single transaction, which is committed by default if execution reaches the **end** statement. Only some databases, such as SQL Server, support the above syntax. However, several other databases, such as MySQL and PostgreSQL, support a **begin** statement which starts a transaction containing all subsequent SQL statements, but do not support the **end** statement; instead, the transaction must be ended by either a **commit work** or a **rollback work** command.

If you use a database such as Oracle, where the automatic commit is not the default for DML statements, be sure to issue a **commit** command after adding or modifying data, or else when you disconnect, all your database modifications will be rolled back!⁷ You should be aware that although Oracle has automatic commit turned off by default, that default may be overridden by local configuration settings.

We study further properties of transactions in Chapter 17; issues in implementing transactions are addressed in Chapter 18 and Chapter 19.

4.4 Integrity Constraints

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database. This is in contrast to *security constraints*, which guard against access to the database by unauthorized users.

Examples of integrity constraints are:

- An instructor name cannot be *null*.
- No two instructors can have the same instructor ID.
- Every department name in the *course* relation must have a matching department name in the *department* relation.
- The budget of a department must be greater than \$0.00.

In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, most database systems allow one to specify only those integrity constraints that can be tested with minimal overhead.

We have already seen some forms of integrity constraints in Section 3.2.2. We study some more forms of integrity constraints in this section. In Chapter 7, we study another form of integrity constraint, called **functional dependencies**, that is used primarily in the process of schema design.

⁷Oracle does automatically commit DDL statements.

Integrity constraints are usually identified as part of the database schema design process and declared as part of the **create table** command used to create relations. However, integrity constraints can also be added to an existing relation by using the command **alter table table-name add constraint**, where *constraint* can be any constraint on the relation. When such a command is executed, the system first ensures that the relation satisfies the specified constraint. If it does, the constraint is added to the relation; if not, the command is rejected.

4.4.1 Constraints on a Single Relation

We described in Section 3.2 how to define tables using the **create table** command. The **create table** command may also include integrity-constraint statements. In addition to the primary-key constraint, there are a number of other ones that can be included in the **create table** command. The allowed integrity constraints include

- **not null**
- **unique**
- **check(<predicate>)**

We cover each of these types of constraints in the following sections.

4.4.2 Not Null Constraint

As we discussed in Chapter 3, the null value is a member of all domains, and as a result it is a legal value for every attribute in SQL by default. For certain attributes, however, null values may be inappropriate. Consider a tuple in the *student* relation where *name* is *null*. Such a tuple gives student information for an unknown student; thus, it does not contain useful information. Similarly, we would not want the department budget to be *null*. In cases such as this, we wish to forbid null values, and we can do so by restricting the domain of the attributes *name* and *budget* to exclude null values, by declaring it as follows:

```
name varchar(20) not null
budget numeric(12,2) not null
```

The **not null** constraint prohibits the insertion of a null value for the attribute, and is an example of a **domain constraint**. Any database modification that would cause a null to be inserted in an attribute declared to be **not null** generates an error diagnostic.

There are many situations where we want to avoid null values. In particular, SQL prohibits null values in the primary key of a relation schema. Thus, in our university example, in the *department* relation, if the attribute *dept_name* is declared as the primary key for *department*, it cannot take a null value. As a result it would not need to be declared explicitly to be **not null**.

4.4.3 Unique Constraint

SQL also supports an integrity constraint:

$$\text{unique } (A_{j_1}, A_{j_2}, \dots, A_{j_m})$$

The **unique** specification says that attributes $A_{j_1}, A_{j_2}, \dots, A_{j_m}$ form a superkey; that is, no two tuples in the relation can be equal on all the listed attributes. However, attributes declared as unique are permitted to be *null* unless they have explicitly been declared to be **not null**. Recall that a null value does not equal any other value. (The treatment of nulls here is the same as that of the **unique** construct defined in Section 3.8.4.)

4.4.4 The Check Clause

When applied to a relation declaration, the clause **check**(P) specifies a predicate P that must be satisfied by every tuple in a relation.

A common use of the **check** clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system. For instance, a clause **check** ($\text{budget} > 0$) in the **create table** command for relation *department* would ensure that the value of **budget** is nonnegative.

As another example, consider the following:

```
create table section
  (course_id      varchar (8),
   sec_id        varchar (8),
   semester       varchar (6),
   year          numeric (4,0),
   building      varchar (15),
   room_number   varchar (7),
   time_slot_id  varchar (4),
   primary key (course_id, sec_id, semester, year),
   check (semester in ('Fall', 'Winter', 'Spring', 'Summer')));
```

Here, we use the **check** clause to simulate an enumerated type by specifying that *semester* must be one of 'Fall', 'Winter', 'Spring', or 'Summer'. Thus, the **check** clause permits attribute domains to be restricted in powerful ways that most programming-language type systems do not permit.

Null values present an interesting special case in the evaluation of a **check** clause. A **check** clause is satisfied if it is not false, so clauses that evaluate to **unknown** are not violations. If null values are not desired, a separate **not null** constraint (see Section 4.4.2) must be specified.

A **check** clause may appear on its own, as shown above, or as part of the declaration of an attribute. In Figure 4.9, we show the **check** constraint for the *semester* attribute

```

create table classroom
  (building      varchar (15),
   room_number varchar (7),
   capacity     numeric (4,0),
   primary key (building, room_number));

create table department
  (dept_name    varchar (20),
   building      varchar (15),
   budget        numeric (12,2) check (budget > 0),
   primary key (dept_name));

create table course
  (course_id    varchar (8),
   title         varchar (50),
   dept_name    varchar (20),
   credits       numeric (2,0) check (credits > 0),
   primary key (course_id),
   foreign key (dept_name) references department);

create table instructor
  (ID           varchar (5),
   name          varchar (20) not null,
   dept_name    varchar (20),
   salary        numeric (8,2) check (salary > 29000),
   primary key (ID),
   foreign key (dept_name) references department);

create table section
  (course_id    varchar (8),
   sec_id        varchar (8),
   semester      varchar (6) check (semester in
                                         ('Fall', 'Winter', 'Spring', 'Summer')),
   year          numeric (4,0) check (year > 1759 and year < 2100),
   building      varchar (15),
   room_number varchar (7),
   time_slot_id varchar (4),
   primary key (course_id, sec_id, semester, year),
   foreign key (course_id) references course,
   foreign key (building, room_number) references classroom);

```

Figure 4.9 SQL data definition for part of the university database.

as part of the declaration of *semester*. The placement of a **check** clause is a matter of coding style. Typically, constraints on the value of a single attribute are listed with that attribute, while more complex **check** clauses are listed separately at the end of a **create table** statement.

The predicate in the **check** clause can, according to the SQL standard, be an arbitrary predicate that can include a subquery. However, currently none of the widely used database products allows the predicate to contain a subquery.

4.4.5 Referential Integrity

Often, we wish to ensure that a value that appears in one relation (the *referencing* relation) for a given set of attributes also appears for a certain set of attributes in another relation (the *referenced* relation). As we saw earlier, in Section 2.3, such conditions are called *referential integrity constraints*, and *foreign keys* are a form of a referential integrity constraint where the referenced attributes form a primary key of the referenced relation.

Foreign keys can be specified as part of the SQL **create table** statement by using the **foreign key** clause, as we saw in Section 3.2.2. We illustrate foreign-key declarations by using the SQL DDL definition of part of our university database, shown in Figure 4.9. The definition of the *course* table has a declaration

foreign key (dept_name) references department.

This foreign-key declaration specifies that for each course tuple, the department name specified in the tuple must exist in the *department* relation. Without this constraint, it is possible for a course to specify a nonexistent department name.

By default, in SQL a foreign key references the primary-key attributes of the referenced table. SQL also supports a version of the **references** clause where a list of attributes of the referenced relation can be specified explicitly.⁸ For example, the foreign key declaration for the *course* relation can be specified as:

foreign key (dept_name) references department(dept_name)

The specified list of attributes must, however, be declared as a superkey of the referenced relation, using either a **primary key** constraint or a **unique** constraint. A more general form of a referential-integrity constraint, where the referenced columns need not be a candidate key, cannot be directly specified in SQL. The SQL standard specifies other constructs that can be used to implement such constraints, which are described in Section 4.4.8; however, these alternative constructs are not supported by any of the widely used database systems.

Note that the foreign key must reference a compatible set of attributes, that is, the number of attributes must be the same and the data types of corresponding attributes must be compatible.

⁸Some systems, notably MySQL, do not support the default and require that the attributes of the referenced relations be specified.

We can use the following as part of a table definition to declare that an attribute forms a foreign key:

```
dept_name varchar(20) references department
```

When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation (i.e., the transaction performing the update action is rolled back). However, a **foreign key** clause can specify that if a delete or update action on the referenced relation violates the constraint, then, instead of rejecting the action, the system must take steps to change the tuple in the referencing relation to restore the constraint. Consider this definition of an integrity constraint on the relation *course*:

```
create table course
(
    ...
    foreign key (dept_name) references department
        on delete cascade
        on update cascade,
    ...
);
```

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *department* results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete “**cascades**” to the *course* relation, deleting the tuple that refers to the department that was deleted. Similarly, the system does not reject an update to a field referenced by the constraint if it violates the constraint; instead, the system updates the field *dept_name* in the referencing tuples in *course* to the new value as well. SQL also allows the **foreign key** clause to specify actions other than **cascade**, if the constraint is violated: The referencing field (here, *dept_name*) can be set to *null* (by using **set null** in place of **cascade**), or to the default value for the domain (by using **set default**).

If there is a chain of foreign-key dependencies across multiple relations, a deletion or update at one end of the chain can propagate across the entire chain. An interesting case where the **foreign key** constraint on a relation references the same relation appears in Exercise 4.9. If a cascading update or delete causes a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction. As a result, all the changes caused by the transaction and its cascading actions are undone.

Null values complicate the semantics of referential-integrity constraints in SQL. Attributes of foreign keys are allowed to be *null*, provided that they have not otherwise been declared to be **not null**. If all the columns of a foreign key are nonnull in a given tuple, the usual definition of foreign-key constraints is used for that tuple. If any of the foreign-key columns is *null*, the tuple is defined automatically to satisfy the constraint. This definition may not always be the right choice, so SQL also provides constructs that allow you to change the behavior with null values; we do not discuss the constructs here.

4.4.6 Assigning Names to Constraints

It is possible for us to assign a name to integrity constraints. Such names are useful if we want to drop a constraint that was defined previously.

To name a constraint, we precede the constraint with the keyword **constraint** and the name we wish to assign it. So, for example, if we wish to assign the name *minsalary* to the **check** constraint on the *salary* attribute of *instructor* (see Figure 4.9), we would modify the declaration for *salary* to:

```
salary numeric(8,2), constraint minsalary check (salary > 29000),
```

Later, if we decide we no longer want this constraint, we can write:

```
alter table instructor drop constraint minsalary;
```

Lacking a name, we would need first to use system-specific features to identify the system-assigned name for the constraint. Not all systems support this, but, for example, in Oracle, the system table *user_constraints* contains this information.

4.4.7 Integrity Constraint Violation During a Transaction

Transactions may consist of several steps, and integrity constraints may be violated temporarily after one step, but a later step may remove the violation. For instance, suppose we have a relation *person* with primary key *name*, and an attribute *spouse*, and suppose that *spouse* is a foreign key on *person*. That is, the constraint says that the *spouse* attribute must contain a name that is present in the *person* table. Suppose we wish to note the fact that John and Mary are married to each other by inserting two tuples, one for John and one for Mary, in the preceding relation, with the spouse attributes set to Mary and John, respectively. The insertion of the first tuple would violate the foreign-key constraint, regardless of which of the two tuples is inserted first. After the second tuple is inserted, the foreign-key constraint would hold again.

To handle such situations, the SQL standard allows a clause **initially deferred** to be added to a constraint specification; the constraint would then be checked at the end of a transaction and not at intermediate steps. A constraint can alternatively be specified as **deferrable**, which means it is checked immediately by default but can be deferred when desired. For constraints declared as deferrable, executing a statement **set constraints constraint-list deferred** as part of a transaction causes the checking of the specified constraints to be deferred to the end of that transaction. Constraints that are to appear in a constraint list must have names assigned. The default behavior is to check constraints immediately, and many database implementations do not support deferred constraint checking.

We can work around the problem in the preceding example in another way, if the *spouse* attribute can be set to *null*: We set the spouse attributes to *null* when inserting the

tuples for John and Mary, and we update them later. However, this technique requires more programming effort, and it does not work if the attributes cannot be set to *null*.

4.4.8 Complex Check Conditions and Assertions

There are additional constructs in the SQL standard for specifying integrity constraints that are not currently supported by most systems. We discuss some of these in this section.

As defined by the SQL standard, the predicate in the **check** clause can be an arbitrary predicate that can include a subquery. If a database implementation supports subqueries in the **check** clause, we could specify the following referential-integrity constraint on the relation *section*:

```
check (time_slot_id in (select time_slot_id from time_slot))
```

The **check** condition verifies that the *time_slot_id* in each tuple in the *section* relation is actually the identifier of a time slot in the *time_slot* relation. Thus, the condition has to be checked not only when a tuple is inserted or modified in *section*, but also when the relation *time_slot* changes (in this case, when a tuple is deleted or modified in relation *time_slot*).

Another natural constraint on our university schema would be to require that every section has at least one instructor teaching the section. In an attempt to enforce this, we may try to declare that the attributes (*course_id*, *sec_id*, *semester*, *year*) of the *section* relation form a foreign key referencing the corresponding attributes of the *teaches* relation. Unfortunately, these attributes do not form a candidate key of the relation *teaches*. A check constraint similar to that for the *time_slot* attribute can be used to enforce this constraint, if check constraints with subqueries were supported by a database system.

Complex **check** conditions can be useful when we want to ensure the integrity of data, but they may be costly to test. In our example, the predicate in the **check** clause would not only have to be evaluated when a modification is made to the *section* relation, but it may have to be checked if a modification is made to the *time_slot* relation because that relation is referenced in the subquery.

An **assertion** is a predicate expressing a condition that we wish the database always to satisfy. Consider the following constraints, which can be expressed using assertions.

- For each tuple in the *student* relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
- An instructor cannot teach in two different classrooms in a semester in the same time slot.⁹

⁹We assume that lectures are not displayed remotely in a second classroom! An alternative constraint that specifies that “an instructor cannot teach two courses in a given semester in the same time slot” may not hold since courses are sometimes cross-listed; that is, the same course is given two identifiers and titles.

```

create assertion credits_earned_constraint check
(not exists (select ID
        from student
        where tot_cred <> (select coalesce(sum(credits), 0)
              from takes natural join course
              where student.ID= takes.ID
              and grade is not null and grade<> 'F' ))
```

Figure 4.10 An assertion example.

An assertion in SQL takes the form:

```
create assertion <assertion-name> check <predicate>;
```

In Figure 4.10, we show how the first example of constraints can be written in SQL. Since SQL does not provide a “for all X , $P(X)$ ” construct (where P is a predicate), we are forced to implement the constraint by an equivalent construct, “not exists X such that not $P(X)$ ”, that can be expressed in SQL.

We leave the specification of the second constraint as an exercise. Although these two constraints can be expressed using **check** predicates, using an assertion may be more natural, especially for the second constraint.

When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated. This testing may introduce a significant amount of overhead if complex assertions have been made. Hence, assertions should be used with great care. The high overhead of testing and maintaining assertions has led some system developers to omit support for general assertions, or to provide specialized forms of assertion that are easier to test.

Currently, none of the widely used database systems supports either subqueries in the **check** clause predicate or the **create assertion** construct. However, equivalent functionality can be implemented using triggers, which are described in Section 5.3, if they are supported by the database system. Section 5.3 also describes how the referential integrity constraint on *time_slot_id* can be implemented using triggers.

4.5 SQL Data Types and Schemas

In Chapter 3, we covered a number of built-in data types supported in SQL, such as integer types, real types, and character types. There are additional built-in data types supported by SQL, which we describe below. We also describe how to create basic user-defined types in SQL.

4.5.1 Date and Time Types in SQL

In addition to the basic data types we introduced in Section 3.2, the SQL standard supports several data types relating to dates and times:

- **date**: A calendar date containing a (four-digit) year, month, and day of the month.
- **time**: The time of day, in hours, minutes, and seconds. A variant, **time**(*p*), can be used to specify the number of fractional digits for seconds (the default being 0). It is also possible to store time-zone information along with the time by specifying **time with timezone**.
- **timestamp**: A combination of **date** and **time**. A variant, **timestamp**(*p*), can be used to specify the number of fractional digits for seconds (the default here being 6). Time-zone information is also stored if **with timezone** is specified.

Date and time values can be specified like this:

```
date '2018-04-25'  
time '09:30:00'  
timestamp '2018-04-25 10:29:01.45'
```

Dates must be specified in the format year followed by month followed by day, as shown.¹⁰ The seconds field of **time** or **timestamp** can have a fractional part, as in the timestamp above.

To extract individual fields of a **date** or **time** value *d*, we can use **extract** (*field from d*), where *field* can be one of **year**, **month**, **day**, **hour**, **minute**, or **second**. Time-zone information can be extracted using **timezone_hour** and **timezone_minute**.

SQL defines several functions to get the current date and time. For example, **current_date** returns the current date, **current_time** returns the current time (with time zone), and **localtime** returns the current local time (without time zone). Timestamps (date plus time) are returned by **current_timestamp** (with time zone) and **localtimestamp** (local date and time without time zone).

Some systems, including MySQL offer the **datetime** data type that represents a time that is not adjustable for time zone. In practice, specification of time has numerous special cases, including the use of standard time versus “daylight” or “summer” time. Systems vary in the range of times representable.

SQL allows comparison operations on all the types listed here, and it allows both arithmetic and comparison operations on the various numeric types. SQL also provides a data type called **interval**, and it allows computations based on dates and times and on intervals. For example, if *x* and *y* are of type **date**, then *x – y* is an interval whose value is the number of days from date *x* to date *y*. Similarly, adding or subtracting an interval from a date or time gives back a date or time, respectively.

¹⁰Many database systems offer greater flexibility in default conversions of strings to dates and timestamps.

4.5.2 Type Conversion and Formatting Functions

Although systems perform some data type **conversions** automatically, others need to be requested explicitly. We can use an expression of the form **cast** (*e as t*) to convert an expression *e* to the type *t*. Data-type conversions may be needed to perform certain operations or to enforce certain sort orders. For example, consider the *ID* attribute of *instructor*, which we have specified as being a string (**varchar(5)**). If we were to order output by this attribute, the ID 11111 comes before the ID 9, because the first character, '1', comes before '9'. However, if we were to write:

```
select cast(ID as numeric(5)) as inst_id
from instructor
order by inst_id
```

the result would be the sorted order we desire.

A different type of conversion may be required for data to be displayed as the result of a query. For example, we may wish numbers to be shown with a specific number of digits, or data to be displayed in a particular format (such as month-day-year or day-month-year). These changes in display format are not conversion of data type but rather conversion of format. Database systems offer a variety of formatting functions, and details vary among the leading systems. MySQL offers a **format** function. Oracle and PostgreSQL offer a set of functions, **to_char**, **to_number**, and **to_date**. SQL Server offers a **convert** function.

Another issue in displaying results is the handling of null values. In this text, we use **null** for clarity of reading, but the default in most systems is just to leave the field blank. We can choose how null values are output in a query result using the **coalesce** function. It takes an arbitrary number of arguments, all of which must be of the same type, and returns the first non-null argument. For example, if we wished to display instructor IDs and salaries but to show null salaries as 0, we would write:

```
select ID, coalesce(salary, 0) as salary
from instructor
```

A limitation of **coalesce** is the requirement that all the arguments must be of the same type. If we had wanted null salaries to appear as 'N/A' to indicate "not available", we would not be able to use **coalesce**. System-specific functions, such as Oracle's **decode**, do allow such conversions. The general form of **decode** is:

```
decode (value, match-1, replacement-1, match-2, replacement-2, ...,
match-N, replacement-N, default-replacement);
```

It compares *value* against the *match* values and if a match is found, it replaces the attribute value with the corresponding replacement value. If no match succeeds, then the attribute value is replaced with the default replacement value. There are no require-

ments that datatypes match. Conveniently, the value *null* may appear as a *match* value and, unlike the usual case, *null* is treated as being equal to *null*. Thus, we could replace null salaries with 'N/A' as follows:

```
select ID, decode (salary, null, 'N/A', salary) as salary
from instructor
```

4.5.3 Default Values

SQL allows a **default** value to be specified for an attribute as illustrated by the following **create table** statement:

```
create table student
  (ID          varchar (5),
   name        varchar (20) not null,
   dept_name   varchar (20),
   tot_cred    numeric (3,0) default 0,
   primary key (ID));
```

The default value of the *tot_cred* attribute is declared to be 0. As a result, when a tuple is inserted into the *student* relation, if no value is provided for the *tot_cred* attribute, its value is set to 0. The following **insert** statement illustrates how an insertion can omit the value for the *tot_cred* attribute.

```
insert into student(ID, name, dept_name)
values ('12789', 'Newman', 'Comp. Sci.');
```

4.5.4 Large-Object Types

Many database applications need to store attributes whose domain consists of large data items such as a photo, a high-resolution medical image, or a video. SQL, therefore, provides **large-object data types** for character data (**clob**) and binary data (**blob**). The letters “lob” in these data types stand for “Large OBject.” For example, we may declare attributes

```
book_review clob(10KB)
image blob(10MB)
movie blob(2GB)
```

For result tuples containing large objects (multiple megabytes to gigabytes), it is inefficient or impractical to retrieve an entire large object into memory. Instead, an application would usually use an SQL query to retrieve a “locator” for a large object and then use the locator to manipulate the object from the host language in which the application itself is written. For instance, the JDBC application program interface (described in Section 5.1.1) permits a locator to be fetched instead of the entire large

Note 4.2 TEMPORAL VALIDITY

In some situations, there is a need to include historical data, as, for example, if we wish to store not only the current salary of each instructor but also entire salary histories. It is easy enough to do this by adding two attributes to the *instructor* relation schema indicating the starting date for a given salary value and another indicating the end date. Then, an instructor may have several salary values, each corresponding to a specific pair of start and end dates. Those start and end dates are called the *valid time* values for the corresponding salary value.

Observe that there may now be more than one tuple in the *instructor* relation with the same value of ID. Issues in specifying primary key and foreign key constraints in the context of such temporal data are discussed in Section 7.10.

For a database system to support such temporal constructs, a first step is to provide syntax to specify that certain attributes define a valid time interval. We use Oracle 12's syntax as an example. The SQL DDL for *instructor* is augmented using a **period** declaration as follows, to indicate that *start_date* and *end_date* attributes specify a valid-time interval.

```
create table instructor
  (
    ...
    start_date      date,
    end_date       date,
    period for valid_time (start_date, end_date),
    ...
  );
```

Oracle 12c also provides several DML extensions to ease querying with temporal data. The **as of period for** construct can then be used in query to fetch only those tuples whose valid time period includes a specific time. To find instructors and their salaries as of some time in the past, say January 20, 2014, we write:

```
select name, salary, start_date, end_date
  from instructor as of period for valid_time '20-JAN-2014';
```

If we wish to find tuples whose period of validity includes all or part of a period of time, say, January 20, 2014 to January 30, 2014, we write:

```
select name, salary, start_date, end_date
  from instructor versions period for valid_time between '20-JAN-2014' and '30-JAN-2014';
```

Oracle 12c also implements a feature that allows stored database procedures (covered in Chapter 5) to be run as of a specified time period.

The above constructs ease the specification of the queries, although the queries can be written without using the constructs.

object; the locator can then be used to fetch the large object in small pieces, rather than all at once, much like reading data from an operating system file using a `read` function call.

4.5.5 User-Defined Types

SQL supports two forms of **user-defined data types**. The first form, which we cover here, is called **distinct types**. The other form, called **structured data types**, allows the creation of complex data types with nested record structures, arrays, and multisets. We do not cover structured data types in this chapter, but we describe them in Section 8.2.

It is possible for several attributes to have the same data type. For example, the *name* attributes for student name and instructor name might have the same domain: the set of all person names. However, the domains of *budget* and *dept_name* certainly ought to be distinct. It is perhaps less clear whether *name* and *dept_name* should have the same domain. At the implementation level, both instructor names and department names are character strings. However, we would normally not consider the query “Find all instructors who have the same name as a department” to be a meaningful query. Thus, if we view the database at the conceptual, rather than the physical, level, *name* and *dept_name* should have distinct domains.

More importantly, at a practical level, assigning an instructor’s name to a department name is probably a programming error; similarly, comparing a monetary value expressed in dollars directly with a monetary value expressed in pounds is also almost surely a programming error. A good type system should be able to detect such assignments or comparisons. To support such checks, SQL provides the notion of **distinct types**.

The `create type` clause can be used to define new types. For example, the statements:

```
create type Dollars as numeric(12,2) final;
create type Pounds as numeric(12,2) final;
```

define the user-defined types *Dollars* and *Pounds* to be decimal numbers with a total of 12 digits, two of which are placed after the decimal point.¹¹ The newly created types can then be used, for example, as types of attributes of relations. For example, we can declare the *department* table as:

```
create table department
  (dept_name    varchar (20),
   building      varchar (15),
   budget        Dollars);
```

An attempt to assign a value of type *Dollars* to a variable of type *Pounds* results in a compile-time error, although both are of the same numeric type. Such an assignment is likely to be due to a programmer error, where the programmer forgot about the

¹¹The keyword `final` isn’t really meaningful in this context but is required by the SQL:1999 standard for reasons we won’t get into here; some implementations allow the `final` keyword to be omitted.

differences in currency. Declaring different types for different currencies helps catch such errors.

As a result of strong type checking, the expression (*department.budget*+20) would not be accepted since the attribute and the integer constant 20 have different types. As we saw in Section 4.5.2, values of one type can be converted to another domain, as illustrated below:

```
cast (department.budget to numeric(12,2))
```

We could do addition on the numeric type, but to save the result back to an attribute of type *Dollars* we would have to use another cast expression to convert the type back to *Dollars*.

SQL provides **drop type** and **alter type** clauses to drop or modify types that have been created earlier.

Even before user-defined types were added to SQL (in SQL:1999), SQL had a similar but subtly different notion of **domain** (introduced in SQL-92), which can add integrity constraints to an underlying type. For example, we could define a domain *DDollars* as follows.

```
create domain DDollars as numeric(12,2) not null;
```

The domain *DDollars* can be used as an attribute type, just as we used the type *Dollars*. However, there are two significant differences between types and domains:

1. Domains can have constraints, such as **not null**, specified on them, and can have default values defined for variables of the domain type, whereas user-defined types cannot have constraints or default values specified on them. User-defined types are designed to be used not just for specifying attribute types, but also in procedural extensions to SQL where it may not be possible to enforce constraints.
2. Domains are not strongly typed. As a result, values of one domain type can be assigned to values of another domain type as long as the underlying types are compatible.

When applied to a domain, the **check** clause permits the schema designer to specify a predicate that must be satisfied by any attribute declared to be from this domain. For instance, a **check** clause can ensure that an instructor's salary domain allows only values greater than a specified value:

```
create domain YearlySalary numeric(8,2)
constraint salary_value_test check(value >= 29000.00);
```

The domain *YearlySalary* has a constraint that ensures that the *YearlySalary* is greater than or equal to \$29,000.00. The clause **constraint** *salary_value_test* is optional and is

Note 4.3 SUPPORT FOR TYPES AND DOMAINS

Although the **create type** and **create domain** constructs described in this section are part of the SQL standard, the forms of these constructs described here are not fully supported by most database implementations. PostgreSQL supports the **create domain** construct, but its **create type** construct has a different syntax and interpretation.

IBM DB2 supports a version of the **create type** that uses the syntax **create distinct type**, but it does not support **create domain**. Microsoft SQL Server implements a version of **create type** construct that supports domain constraints, similar to the SQL **create domain** construct.

Oracle does not support either construct as described here. Oracle, IBM DB2, PostgreSQL, and SQL Server all support object-oriented type systems using different forms of the **create type** construct.

However, SQL also defines a more complex object-oriented type system, which we study in Section 8.2. Types may have structure within them, like, for example, a *Name* type consisting of *firstname* and *lastname*. Subtyping is allowed as well; for example, a *Person* type may have subtypes *Student*, *Instructor*, etc. Inheritance rules are similar to those in object-oriented programming languages. It is possible to use references to tuples that behave much like references to objects in object-oriented programming languages. SQL allows array and multiset datatypes along with ways to manipulate those types.

We do not cover the details of these features here. Database systems differ in how they implement them, if they are implemented at all.

used to give the name *salary_value_test* to the constraint. The name is used by the system to indicate the constraint that an update violated.

As another example, a domain can be restricted to contain only a specified set of values by using the **in** clause:

```
create domain degree_level varchar(10)
constraint degree_level_test
check (value in ('Bachelors', 'Masters', 'Doctorate'));
```

4.5.6 Generating Unique Key Values

In our university example, we have seen primary-key attributes with different data types. Some, like *dept_name*, hold actual real-world information. Others, like *ID*, hold values created by the enterprise solely for identification purposes. Those latter types of primary-key domains generate the practical problem of new-value creation. Suppose

the university hires a new instructor. What ID should be assigned? How do we determine that the new ID is unique? Although it is possible to write an SQL statement to do this, such a statement would need to check all preexisting IDs, which would harm system performance. Alternatively, one could set up a special table holding the largest ID value issued so far. Then, when a new ID is needed, that value can be incremented to the next one in sequence and stored as the new largest value.

Database systems offer automatic management of unique key-value generation. The syntax differs among the most popular systems and, sometimes, between versions of systems. The syntax we show here is close to that of Oracle and DB2. Suppose that instead of declaring instructor IDs in the *instructor* relation as “*ID* *varchar(5)*”, we instead choose to let the system select a unique instructor ID value. Since this feature works only for numeric key-value data types, we change the type of *ID* to **number**, and write:

***ID* **number(5)** generated always as identity**

When the **always** option is used, any **insert** statement must avoid specifying a value for the automatically generated key. To do this, use the syntax for **insert** in which the attribute order is specified (see Section 3.9.2). For our example of *instructor*, we need specify only the values for *name*, *dept_name*, and *salary*, as shown in the following example:

```
insert into instructor (name, dept_name, salary)
values ('Newprof', 'Comp. Sci.', 100000);
```

The generated ID value can be found via a normal **select** query. If we replace **always** with **by default**, we have the option of specifying our own choice of *ID* or relying on the system to generate one.

In PostgreSQL, we can define the type of *ID* as **serial**, which tells PostgreSQL to automatically generate identifiers; in MySQL we use **auto_increment** in place of **generated always as identity**, while in SQL Server we can use just **identity**.

Additional options can be specified, with the **identity** specification, depending on the database, including setting minimum and maximum values, choosing the starting value, choosing the increment from one value to the next, and so on.

Further, many databases support a **create sequence** construct, which creates a sequence counter object separate from any relation, and allow SQL queries to get the next value from the sequence. Each call to get the next value increments the sequence counter. See the system manuals of the database to find the exact syntax for creating sequences, and for retrieving the next value. Using sequences, we can generate identifiers that are unique across multiple relations, for example, across *student.ID*, and *instructor.ID*.

4.5.7 Create Table Extensions

Applications often require the creation of tables that have the same schema as an existing table. SQL provides a **create table like** extension to support this task:¹²

```
create table temp_instructor like instructor;
```

The above statement creates a new table *temp_instructor* that has the same schema as *instructor*.

When writing a complex query, it is often useful to store the result of a query as a new table; the table is usually temporary. Two statements are required, one to create the table (with appropriate columns) and the second to insert the query result into the table. SQL:2003 provides a simpler technique to create a table containing the results of a query. For example, the following statement creates a table *t1* containing the results of a query.

```
create table t1 as
  (select *
   from instructor
   where dept_name = 'Music')
 with data;
```

By default, the names and data types of the columns are inferred from the query result. Names can be explicitly given to the columns by listing the column names after the relation name.

As defined by the SQL:2003 standard, if the **with data** clause is omitted, the table is created but not populated with data. However, many implementations populate the table with data by default even if the **with data** clause is omitted. Note that several implementations support the functionality of **create table ... like** and **create table ... as** using different syntax; see the respective system manuals for further details.

The above **create table ... as** statement, closely resembles the **create view** statement and both are defined by using queries. The main difference is that the contents of the table are set when the table is created, whereas the contents of a view always reflect the current query result.

4.5.8 Schemas, Catalogs, and Environments

To understand the motivation for schemas and catalogs, consider how files are named in a file system. Early file systems were flat; that is, all files were stored in a single directory. Current file systems have a directory (or, synonymously, folder) structure, with files stored within subdirectories. To name a file uniquely, we must specify the full path name of the file, for example, `/users/avi/db-book/chapter3.tex`.

¹²This syntax is not supported in all systems.

Like early file systems, early database systems also had a single name space for all relations. Users had to coordinate to make sure they did not try to use the same name for different relations. Contemporary database systems provide a three-level hierarchy for naming relations. The top level of the hierarchy consists of **catalogs**, each of which can contain **schemas**. SQL objects such as relations and views are contained within a **schema**. (Some database implementations use the term *database* in place of the term *catalog*.)

In order to perform any actions on a database, a user (or a program) must first *connect* to the database. The user must provide the user name and usually, a password for verifying the identity of the user. Each user has a default catalog and schema, and the combination is unique to the user. When a user connects to a database system, the default catalog and schema are set up for the connection; this corresponds to the current directory being set to the user's home directory when the user logs into an operating system.

To identify a relation uniquely, a three-part name may be used, for example,

catalog5.univ_schema.course

We may omit the catalog component, in which case the catalog part of the name is considered to be the default catalog for the connection. Thus, if *catalog5* is the default catalog, we can use *univ_schema.course* to identify the same relation uniquely.

If a user wishes to access a relation that exists in a different schema than the default schema for that user, the name of the schema must be specified. However, if a relation is in the default schema for a particular user, then even the schema name may be omitted. Thus, we can use just *course* if the default catalog is *catalog5* and the default schema is *univ_schema*.

With multiple catalogs and schemas available, different applications and different users can work independently without worrying about name clashes. Moreover, multiple versions of an application—one a production version, other test versions—can run on the same database system.

The default catalog and schema are part of an **SQL environment** that is set up for each connection. The environment additionally contains the user identifier (also referred to as the *authorization identifier*). All the usual SQL statements, including the DDL and DML statements, operate in the context of a schema.

We can create and drop schemas by means of **create schema** and **drop schema** statements. In most database systems, schemas are also created automatically when user accounts are created, with the schema name set to the user account name. The schema is created in either a default catalog or a catalog specified when creating the user account. The newly created schema becomes the default schema for the user account.

Creation and dropping of catalogs is implementation dependent and not part of the SQL standard.

4.6 Index Definition in SQL

Many queries reference only a small proportion of the records in a file. For example, a query like “Find all instructors in the Physics department” or “Find the *salary* value of the instructor with *ID* 22201” references only a fraction of the instructor records. It is inefficient for the system to read every record and to check *ID* field for the *ID* “32556,” or the *building* field for the value “Physics”.

An **index** on an attribute of a relation is a data structure that allows the database system to find those tuples in the relation that have a specified value for that attribute efficiently, without scanning through all the tuples of the relation. For example, if we create an index on attribute *dept_name* of relation *instructor*, the database system can find the record with any specified *dept_name* value, such as “Physics”, or “Music”, directly, without reading all the tuples of the *instructor* relation. An index can also be created on a list of attributes, for example, on attributes *name* and *dept_name* of *instructor*.

Indices are not required for correctness, since they are redundant data structures. Indices form part of the physical schema of the database, as opposed to its logical schema.

However, indices are important for efficient processing of transactions, including both update transactions and queries. Indices are also important for efficient enforcement of integrity constraints such as primary-key and foreign-key constraints. In principle, a database system can decide automatically what indices to create. However, because of the space cost of indices, as well as the effect of indices on update processing, it is not easy to automatically make the right choices about what indices to maintain.

Therefore, most SQL implementations provide the programmer with control over the creation and removal of indices via data-definition-language commands. We illustrate the syntax of these commands next. Although the syntax that we show is widely used and supported by many database systems, it is not part of the SQL standard. The SQL standard does not support control of the physical database schema; it restricts itself to the logical database schema.

We create an index with the **create index** command, which takes the form:

```
create index <index-name> on <relation-name> (<attribute-list>);
```

The *attribute-list* is the list of attributes of the relations that form the search key for the index.

To define an index named *dept_index* on the *instructor* relation with *dept_name* as the search key, we write:

```
create index dept_index on instructor (dept_name);
```

When a user submits an SQL query that can benefit from using an index, the SQL query processor automatically uses the index. For example, given an SQL query that

selects the *instructor* tuple with *dept_name* “Music”, the SQL query processor would use the index *dept_index* defined above to find the required tuple without reading the whole relation.

If we wish to declare that the search key is a candidate key, we add the attribute **unique** to the index definition. Thus, the command:

```
create unique index dept_index on instructor (dept_name);
```

declares *dept_name* to be a candidate key for *instructor* (which is probably not what we actually would want for our university database). If, at the time we enter the **create unique index** command, *dept_name* is not a candidate key, the system will display an error message, and the attempt to create the index will fail. If the index-creation attempt succeeds, any subsequent attempt to insert a tuple that violates the key declaration will fail. Note that the **unique** feature is redundant if the database system supports the **unique** declaration of the SQL standard.

The index name we specified for an index is required to drop an index. The **drop index** command takes the form:

```
drop index <index-name>;
```

Many database systems also provide a way to specify the type of index to be used, such as B⁺-tree or hash indices, which we study in Chapter 14. Some database systems also permit one of the indices on a relation to be declared to be clustered; the system then stores the relation sorted by the search key of the clustered index. We study in Chapter 14 how indices are actually implemented, as well as what indices are automatically created by databases, and how to decide on what additional indices to create.

4.7 Authorization

We may assign a user several forms of authorizations on parts of the database. Authorizations on data include:

- Authorization to read data.
- Authorization to insert new data.
- Authorization to update data.
- Authorization to delete data.

Each of these types of authorizations is called a **privilege**. We may authorize the user all, none, or a combination of these types of privileges on specified parts of a database, such as a relation or a view.

When a user submits a query or an update, the SQL implementation first checks if the query or update is authorized, based on the authorizations that the user has been granted. If the query or update is not authorized, it is rejected.

In addition to authorizations on data, users may also be granted authorizations on the database schema, allowing them, for example, to create, modify, or drop relations. A user who has some form of authorization may be allowed to pass on (grant) this authorization to other users, or to withdraw (revoke) an authorization that was granted earlier. In this section, we see how each of these authorizations can be specified in SQL.

The ultimate form of authority is that given to the database administrator. The database administrator may authorize new users, restructure the database, and so on. This form of authorization is analogous to that of a **superuser**, administrator, or operator for an operating system.

4.7.1 Granting and Revoking of Privileges

The SQL standard includes the **privileges** `select`, `insert`, `update`, and `delete`. The privilege **all privileges** can be used as a short form for all the allowable privileges. A user who creates a new relation is given all privileges on that relation automatically.

The SQL data-definition language includes commands to grant and revoke privileges. The **grant** statement is used to confer authorization. The basic form of this statement is:

```
grant <privilege list>
on <relation name or view name>
to <user/role list>;
```

The *privilege list* allows the granting of several privileges in one command. The notion of roles is covered in Section 4.7.2.

The **select** authorization on a relation is required to read tuples in the relation. The following **grant** statement grants database users Amit and Satoshi **select** authorization on the *department* relation:

```
grant select on department to Amit, Satoshi;
```

This allows those users to run queries on the *department* relation.

The **update** authorization on a relation allows a user to update any tuple in the relation. The **update** authorization may be given either on all attributes of the relation or on only some. If **update** authorization is included in a **grant** statement, the list of attributes on which update authorization is to be granted optionally appears in parentheses immediately after the **update** keyword. If the list of attributes is omitted, the update privilege will be granted on all attributes of the relation.

This **grant** statement gives users Amit and Satoshi update authorization on the *budget* attribute of the *department* relation:

```
grant update (budget) on department to Amit, Satoshi;
```

The **insert** authorization on a relation allows a user to insert tuples into the relation. The **insert** privilege may also specify a list of attributes; any inserts to the relation must specify only these attributes, and the system either gives each of the remaining attributes default values (if a default is defined for the attribute) or sets them to *null*.

The **delete** authorization on a relation allows a user to delete tuples from a relation.

The user name **public** refers to all current and future users of the system. Thus, privileges granted to **public** are implicitly granted to all current and future users.

By default, a user/role that is granted a privilege is not authorized to grant that privilege to another user/role. SQL allows a privilege grant to specify that the recipient may further grant the privilege to another user. We describe this feature in more detail in Section 4.7.5.

It is worth noting that the SQL authorization mechanism grants privileges on an entire relation, or on specified attributes of a relation. However, it does not permit authorizations on specific tuples of a relation.

To revoke an authorization, we use the **revoke** statement. It takes a form almost identical to that of **grant**:

```
revoke <privilege list>
on <relation name or view name>
from <user/role list>;
```

Thus, to revoke the privileges that we granted previously, we write

```
revoke select on department from Amit, Satoshi;
revoke update (budget) on department from Amit, Satoshi;
```

Revocation of privileges is more complex if the user from whom the privilege is revoked has granted the privilege to another user. We return to this issue in Section 4.7.5.

4.7.2 Roles

Consider the real-world roles of various people in a university. Each instructor must have the same types of authorizations on the same set of relations. Whenever a new instructor is appointed, she will have to be given all these authorizations individually.

A better approach would be to specify the authorizations that every instructor is to be given, and to identify separately which database users are instructors. The system can use these two pieces of information to determine the authorizations of each instructor. When a new instructor is hired, a user identifier must be allocated to him, and he must be identified as an instructor. Individual permissions given to instructors need not be specified again.

The notion of **roles** captures this concept. A set of roles is created in the database. Authorizations can be granted to roles, in exactly the same fashion as they are granted to individual users. Each database user is granted a set of roles (which may be empty) that she is authorized to perform.

In our university database, examples of roles could include *instructor*, *teaching_assistant*, *student*, *dean*, and *department_chair*.

A less preferable alternative would be to create an *instructor* userid and permit each instructor to connect to the database using the *instructor* userid. The problem with this approach is that it would not be possible to identify exactly which instructor carried out a database update, and this could create security risks. Furthermore, if an instructor leaves the university or is moved to a non instructional role, then a new *instructor* password must be created and distributed in a secure manner to all instructors. The use of roles has the benefit of requiring users to connect to the database with their own userid.

Any authorization that can be granted to a user can be granted to a role. Roles are granted to users just as authorizations are.

Roles can be created in SQL as follows:

```
create role instructor;
```

Roles can then be granted privileges just as the users can, as illustrated in this statement:

```
grant select on takes
to instructor;
```

Roles can be granted to users, as well as to other roles, as these statements show:

```
create role dean;
grant instructor to dean;
grant dean to Satoshi;
```

Thus, the privileges of a user or a role consist of:

- All privileges directly granted to the user/role.
- All privileges granted to roles that have been granted to the user/role.

Note that there can be a chain of roles; for example, the role *teaching_assistant* may be granted to all *instructors*. In turn, the role *instructor* is granted to all *deans*. Thus, the *dean* role inherits all privileges granted to the roles *instructor* and to *teaching_assistant* in addition to privileges granted directly to *dean*.

When a user logs in to the database system, the actions executed by the user during that session have all the privileges granted directly to the user, as well as all privileges

granted to roles that are granted (directly or indirectly via other roles) to that user. Thus, if a user Amit has been granted the role *dean*, user Amit holds all privileges granted directly to Amit, as well as privileges granted to *dean*, plus privileges granted to *instructor* and *teaching_assistant* if, as above, those roles were granted (directly or indirectly) to the role *dean*.

It is worth noting that the concept of role-based authorization is not specific to SQL, and role-based authorization is used for access control in a wide variety of shared applications.

4.7.3 Authorization on Views

In our university example, consider a staff member who needs to know the salaries of all faculty in a particular department, say the Geology department. This staff member is not authorized to see information regarding faculty in other departments. Thus, the staff member must be denied direct access to the *instructor* relation. But if he is to have access to the information for the Geology department, he might be granted access to a view that we shall call *geo_instructor*, consisting of only those *instructor* tuples pertaining to the Geology department. This view can be defined in SQL as follows:

```
create view geo_instructor as
  (select *
   from instructor
   where dept_name = 'Geology');
```

Suppose that the staff member issues the following SQL query:

```
select *
from geo_instructor;
```

The staff member is authorized to see the result of this query. However, when the query processor translates it into a query on the actual relations in the database, it replaces uses of a view by the definition of the view, producing a query on *instructor*. Thus, the system must check authorization on the clerk's query before it replaces views by their definitions.

A user who creates a view does not necessarily receive all privileges on that view. She receives only those privileges that provide no additional authorization beyond those that she already had. For example, a user who creates a view cannot be given **update** authorization on a view without having **update** authorization on the relations used to define the view. If a user creates a view on which no authorization can be granted, the system will deny the view creation request. In our *geo_instructor* view example, the creator of the view must have **select** authorization on the *instructor* relation.

As we will see in Section 5.2, SQL supports the creation of functions and procedures, which may, in turn, contain queries and updates. The **execute** privilege can be granted on a function or procedure, enabling a user to execute the function or proce-

dure. By default, just like views, functions and procedures have all the privileges that the creator of the function or procedure had. In effect, the function or procedure runs as if it were invoked by the user who created the function.

Although this behavior is appropriate in many situations, it is not always appropriate. Starting with SQL:2003, if the function definition has an extra clause **sql security invoker**, then it is executed under the privileges of the user who invokes the function, rather than the privileges of the **definer** of the function. This allows the creation of libraries of functions that can run under the same authorization as the invoker.

4.7.4 Authorizations on Schema

The SQL standard specifies a primitive authorization mechanism for the database schema: Only the owner of the schema can carry out any modification to the schema, such as creating or deleting relations, adding or dropping attributes of relations, and adding or dropping indices.

However, SQL includes a **references** privilege that permits a user to declare foreign keys when creating relations. The SQL **references** privilege is granted on specific attributes in a manner like that for the **update** privilege. The following **grant** statement allows user Mariano to create relations that reference the key *dept_name* of the *department* relation as a foreign key:

```
grant references (dept_name) on department to Mariano;
```

Initially, it may appear that there is no reason ever to prevent users from creating foreign keys referencing another relation. However, recall that foreign-key constraints restrict deletion and update operations on the referenced relation. Suppose Mariano creates a foreign key in a relation *r* referencing the *dept_name* attribute of the *department* relation and then inserts a tuple into *r* pertaining to the Geology department. It is no longer possible to delete the Geology department from the *department* relation without also modifying relation *r*. Thus, the definition of a foreign key by Mariano restricts future activity by other users; therefore, there is a need for the **references** privilege.

Continuing to use the example of the *department* relation, the **references** privilege on *department* is also required to create a **check** constraint on a relation *r* if the constraint has a subquery referencing *department*. This is reasonable for the same reason as the one we gave for foreign-key constraints; a check constraint that references a relation limits potential updates to that relation.

4.7.5 Transfer of Privileges

A user who has been granted some form of authorization may be allowed to pass on this authorization to other users. By default, a user/role that is granted a privilege is not authorized to grant that privilege to another user/role. If we wish to grant a privilege and to allow the recipient to pass the privilege on to other users, we append the **with grant option** clause to the appropriate **grant** command. For example, if we wish to allow

Amit the **select** privilege on *department* and allow Amit to grant this privilege to others, we write:

```
grant select on department to Amit with grant option;
```

The creator of an object (relation/view/role) holds all privileges on the object, including the privilege to grant privileges to others.

Consider, as an example, the granting of update authorization on the *teaches* relation of the university database. Assume that, initially, the database administrator grants update authorization on *teaches* to users U_1 , U_2 , and U_3 , who may, in turn, pass on this authorization to other users. The passing of a specific authorization from one user to another can be represented by an **authorization graph**. The nodes of this graph are the users.

Consider the graph for update authorization on *teaches*. The graph includes an edge $U_i \rightarrow U_j$ if user U_i grants update authorization on *teaches* to U_j . The root of the graph is the database administrator. In the sample graph in Figure 4.11, observe that user U_5 is granted authorization by both U_1 and U_2 ; U_4 is granted authorization by only U_1 .

A user has an authorization *if and only if* there is a path from the root of the authorization graph (the node representing the database administrator) down to the node representing the user.

4.7.6 Revoking of Privileges

Suppose that the database administrator decides to revoke the authorization of user U_1 . Since U_4 has authorization from U_1 , that authorization should be revoked as well. However, U_5 was granted authorization by both U_1 and U_2 . Since the database administrator did not revoke update authorization on *teaches* from U_2 , U_5 retains update

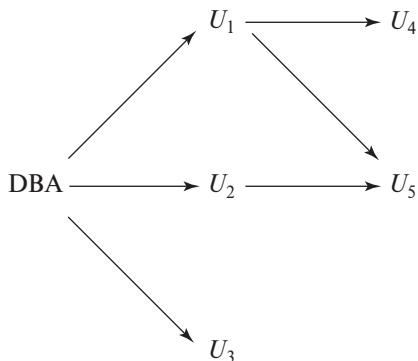


Figure 4.11 Authorization-grant graph (U_1, U_2, \dots, U_5 are users and DBA refers to the database administrator).

authorization on *teaches*. If U_2 eventually revokes authorization from U_5 , then U_5 loses the authorization.

A pair of devious users might attempt to defeat the rules for revocation of authorization by granting authorization to each other. For example, U_2 is initially granted an authorization by the database administrator, and U_2 further grants it to U_3 . Suppose U_3 now grants the privilege back to U_2 . If the database administrator revokes authorization from U_2 , it might appear that U_2 retains authorization through U_3 . However, note that once the administrator revokes authorization from U_2 , there is no path in the authorization graph from the root either to U_2 or to U_3 . Thus, SQL ensures that the authorization is revoked from both the users.

As we just saw, revocation of a privilege from a user/role may cause other users/roles also to lose that privilege. This behavior is called *cascading revocation*. In most database systems, cascading is the default behavior. However, the **revoke** statement may specify **restrict** in order to prevent cascading revocation:

```
revoke select on department from Amit, Satoshi restrict;
```

In this case, the system returns an error if there are any cascading revocations and does not carry out the revoke action.

The keyword **cascade** can be used instead of **restrict** to indicate that revocation should cascade; however, it can be omitted, as we have done in the preceding examples, since it is the default behavior.

The following **revoke** statement revokes only the grant option, rather than the actual **select** privilege:

```
revoke grant option for select on department from Amit;
```

Note that some database implementations do not support the above syntax; instead, the privilege itself can be revoked and then granted again without the grant option.

Cascading revocation is inappropriate in many situations. Suppose Satoshi has the role of *dean*, grants *instructor* to Amit, and later the role *dean* is revoked from Satoshi (perhaps because Satoshi leaves the university); Amit continues to be employed on the faculty and should retain the *instructor* role.

To deal with this situation, SQL permits a privilege to be granted by a role rather than by a user. SQL has a notion of the current role associated with a session. By default, the current role associated with a session is null (except in some special cases). The current role associated with a session can be set by executing **set role role_name**. The specified role must have been granted to the user, otherwise the **set role** statement fails.

To grant a privilege with the grantor set to the current role associated with a session, we can add the clause:

granted by current_role

to the grant statement, provided the current role is not null.

Suppose the granting of the role *instructor* (or other privileges) to Amit is done using the **granted by current_role** clause, with the current role set to *dean*, instead of the grantor being the user Satoshi. Then, revoking of roles/privileges (including the role *dean*) from Satoshi will not result in revoking of privileges that had the grantor set to the role *dean*, even if Satoshi was the user who executed the grant; thus, Amit would retain the *instructor* role even after Satoshi's privileges are revoked.

4.7.7 Row-Level Authorization

The types of authorization privileges we have studied apply at the level of relations or views. Some database systems provide mechanisms for fine-grained authorization at the level of specific tuples within a relation.

Suppose, for example, that we wish to allow a student to see her or his own data in the *takes* relation but not those data of other users. We can enforce such a restriction using row-level authorization, if the database supports it. We describe row-level authorization in Oracle below; PostgreSQL and SQL Server too support row-level authorization using a conceptually similar mechanism, but using a different syntax.

The Oracle **Virtual Private Database (VPD)** feature supports row-level authorization as follows. It allows a system administrator to associate a function with a relation; the function returns a predicate that gets added automatically to any query that uses the relation. The predicate can use the function **sys_context**, which returns the identifier of the user on whose behalf a query is being executed. For our example of students accessing their data in the *takes* relation, we would specify the following predicate to be associated with the *takes* relation:

$$ID = \text{sys_context} ('USERENV', 'SESSION_USER')$$

This predicate is added by the system to the **where** clause of every query that uses the *takes* relation. As a result, each student can see only those *takes* tuples whose ID value matches her ID.

VPD provides authorization at the level of specific tuples, or rows, of a relation, and is therefore said to be a **row-level authorization** mechanism. A potential pitfall with adding a predicate as described above is that it may change the meaning of a query significantly. For example, if a user wrote a query to find the average grade over all courses, she would end up getting the average of *her* grades, not all grades. Although the system would give the “right” answer for the rewritten query, that answer would not correspond to the query the user may have thought she was submitting.

4.8 Summary

- SQL supports several types of joins including natural join, inner and outer joins, and several types of join conditions.

- Natural join provides a simple way to write queries over multiple relations in which a **where** predicate would otherwise equate attributes with matching names from each relation. This convenience comes at the risk of query semantics changing if a new attribute is added to the schema.
 - The **join-using** construct provides a simple way to write queries over multiple relations in which equality is desired for some but not necessarily all attributes with matching names.
 - The **join-on** construct provides a way to include a join predicate in the **from** clause.
 - Outer join provides a means to retain tuples that, due to a join predicate (whether a natural join, a join-using, or a join-on), would otherwise not appear anywhere in the result relation. The retained tuples are padded with null values so as to conform to the result schema.
- View relations can be defined as relations containing the result of queries. Views are useful for hiding unneeded information and for gathering together information from more than one relation into a single view.
 - Transactions are sequences of queries and updates that together carry out a task. Transactions can be committed, or rolled back; when a transaction is rolled back, the effects of all updates performed by the transaction are undone.
 - Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency.
 - Referential-integrity constraints ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
 - Domain constraints specify the set of possible values that may be associated with an attribute. Such constraints may also prohibit the use of null values for particular attributes.
 - Assertions are declarative expressions that state predicates that we require always to be true.
 - The SQL data-definition language provides support for defining built-in domain types such as **date** and **time** as well as user-defined domain types.
 - Indices are important for efficient processing of queries, as well as for efficient enforcement of integrity constraints. Although not part of the SQL standard, SQL commands for creation of indices are supported by most database systems.
 - SQL authorization mechanisms allow one to differentiate among the users of the database on the type of access they are permitted on various data values in the database.

- Roles enable us to assign a set of privileges to a user according to the role that the user plays in the organization.

Review Terms

- Join types
 - Natural join
 - Inner join with **using** and **on**
 - Left, right and full outer join
 - Outer join with **using** and **on**
- View definition
 - Materialized views
 - View maintenance
 - View update
- Transactions
 - Commit work
 - Rollback work
 - Atomic transaction
- Constraints
 - Integrity constraints
 - Domain constraints
 - Unique constraint
 - Check clause
 - Referential integrity
 - ◊ Cascading deletes
 - ◊ Cascading updates
 - Assertions
- Data types
 - Date and time types
- Default values
- Large objects
 - ◊ clob
 - ◊ blob
- User-defined types
- distinct types
- Domains
- Type conversions
- Catalogs
- Schemas
- Indices
- Privileges
 - Types of privileges
 - ◊ **select**
 - ◊ **insert**
 - ◊ **update**
 - Granting of privileges
 - Revoking of privileges
 - Privilege to grant privileges
 - Grant option
- Roles
- Authorization on views
- Execute authorization
- Invoker privileges
- Row-level authorization
- Virtual private database (VPD)

Practice Exercises

- 4.1** Consider the following SQL query that seeks to find a list of titles of all courses taught in Spring 2017 along with the name of the instructor.

```
select name, title
from instructor natural join teaches natural join section natural join course
where semester = 'Spring' and year = 2017
```

What is wrong with this query?

- 4.2** Write the following queries in SQL:

- Display a list of all instructors, showing each instructor's ID and the number of sections taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use subqueries.
- Write the same query as in part a, but using a scalar subquery and not using outer join.
- Display the list of all course sections offered in Spring 2018, along with the ID and name of each instructor teaching the section. If a section has more than one instructor, that section should appear as many times in the result as it has instructors. If a section does not have any instructor, it should still appear in the result with the instructor name set to “—”.
- Display the list of all departments, with the total number of instructors in each department, without using subqueries. Make sure to show departments that have no instructors, and list those departments with an instructor count of zero.

- 4.3** Outer join expressions can be computed in SQL without using the SQL **outer join** operation. To illustrate this fact, show how to rewrite each of the following SQL queries without using the **outer join** expression.

- select * from student natural left outer join takes**
- select * from student natural full outer join takes**

- 4.4** Suppose we have three relations $r(A, B)$, $s(B, C)$, and $t(B, D)$, with all attributes declared as **not null**.

- Give instances of relations r , s , and t such that in the result of $(r \text{ natural left outer join } s) \text{ natural left outer join } t$ attribute C has a null value but attribute D has a non-null value.
- Are there instances of r , s , and t such that the result of $r \text{ natural left outer join } (s \text{ natural left outer join } t)$

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

Figure 4.12 Employee database.

has a null value for *C* but a non-null value for *D*? Explain why or why not.

- 4.5 **Testing SQL queries:** To test if a query specified in English has been correctly written in SQL, the SQL query is typically executed on multiple test databases, and a human checks if the SQL query result on each test database matches the intention of the specification in English.

- a. In Section 4.1.1 we saw an example of an erroneous SQL query which was intended to find which courses had been taught by each instructor; the query computed the natural join of *instructor*, *teaches*, and *course*, and as a result it unintentionally equated the *dept_name* attribute of *instructor* and *course*. Give an example of a dataset that would help catch this particular error.
- b. When creating test databases, it is important to create tuples in referenced relations that do not have any matching tuple in the referencing relation for each foreign key. Explain why, using an example query on the university database.
- c. When creating test databases, it is important to create tuples with null values for foreign-key attributes, provided the attribute is nullable (SQL allows foreign-key attributes to take on null values, as long as they are not part of the primary key and have not been declared as **not null**). Explain why, using an example query on the university database.

Hint: Use the queries from Exercise 4.2.

- 4.6 Show how to define the view *student_grades* (*ID*, *GPA*) giving the grade-point average of each student, based on the query in Exercise 3.2; recall that we used a relation *grade_points*(*grade*, *points*) to get the numeric points associated with a letter grade. Make sure your view definition correctly handles the case of *null* values for the *grade* attribute of the *takes* relation.
- 4.7 Consider the employee database of Figure 4.12. Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

- 4.8** As discussed in Section 4.4.8, we expect the constraint “an instructor cannot teach sections in two different classrooms in a semester in the same time slot” to hold.
- Write an SQL query that returns all (*instructor*, *section*) combinations that violate this constraint.
 - Write an SQL assertion to enforce this constraint (as discussed in Section 4.4.8, current generation database systems do not support such assertions, although they are part of the SQL standard).
- 4.9** SQL allows a foreign-key dependency to refer to the same relation, as in the following example:

```
create table manager
  (employee_ID      char(20),
   manager_ID       char(20),
   primary key employee_ID,
   foreign key (manager_ID) references manager(employee_ID)
                                on delete cascade )
```

Here, *employee_ID* is a key to the table *manager*, meaning that each employee has at most one manager. The foreign-key clause requires that every manager also be an employee. Explain exactly what happens when a tuple in the relation *manager* is deleted.

- 4.10** Given the relations *a*(*name*, *address*, *title*) and *b*(*name*, *address*, *salary*), show how to express *a* **natural full outer join** *b* using the **full outer-join** operation with an **on** condition rather than using the **natural join** syntax. This can be done using the **coalesce** operation. Make sure that the result relation does not contain two copies of the attributes *name* and *address* and that the solution is correct even if some tuples in *a* and *b* have null values for attributes *name* or *address*.
- 4.11** Operating systems usually offer only two types of authorization control for data files: read access and write access. Why do database systems offer so many kinds of authorization?
- 4.12** Suppose a user wants to grant **select** access on a relation to another user. Why should the user include (or not include) the clause **granted by current role** in the **grant** statement?
- 4.13** Consider a view *v* whose definition references only relation *r*.
- If a user is granted **select** authorization on *v*, does that user need to have **select** authorization on *r* as well? Why or why not?
 - If a user is granted **update** authorization on *v*, does that user need to have **update** authorization on *r* as well? Why or why not?

- Give an example of an **insert** operation on a view v to add a tuple t that is not visible in the result of **select * from v** . Explain your answer.

Exercises

- 4.14** Consider the query

```
select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2017
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

Explain why appending **natural join** $section$ in the **from** clause would not change the result.

- 4.15** Rewrite the query

```
select *
from section natural join classroom
```

without using a natural join but instead using an inner join with a **using** condition.

- 4.16** Write an SQL query using the university schema to find the ID of each student who has never taken a course at the university. Do this using no subqueries and no set operations (use an outer join).

- 4.17** Express the following query in SQL using no subqueries and no set operations.

```
select ID
from student
except
select s_id
from advisor
where i_ID is not null
```

- 4.18** For the database of Figure 4.12, write a query to find the ID of each employee with no manager. Note that an employee may simply have no manager listed or may have a *null* manager. Write your query using an outer join and then write it again using no outer join at all.

- 4.19** Under what circumstances would the query

```

select *
from student natural full outer join takes
      natural full outer join course

```

include tuples with null values for the *title* attribute?

- 4.20** Show how to define a view *tot_credits* (*year, num_credits*), giving the total number of credits taken in each year.
- 4.21** For the view of Exercise 4.18, explain why the database system would not allow a tuple to be inserted into the database through this view.
- 4.22** Show how to express the **coalesce** function using the **case** construct.
- 4.23** Explain why, when a manager, say Satoshi, grants an authorization, the grant should be done by the manager role, rather than by the user Satoshi.
- 4.24** Suppose user *A*, who has all authorization privileges on a relation *r*, grants **select** on relation *r* to **public** with grant option. Suppose user *B* then grants **select** on *r* to *A*. Does this cause a cycle in the authorization graph? Explain why.
- 4.25** Suppose a user creates a new relation *r1* with a foreign key referencing another relation *r2*. What authorization privilege does the user need on *r2*? Why should this not simply be allowed without any such authorization?
- 4.26** Explain the difference between integrity constraints and authorization constraints.

Further Reading

General SQL references were provided in Chapter 3. As noted earlier, many systems implement features in a non-standard manner, and, for that reason, a reference specific to the database system you are using is an essential guide. Most vendors also provide extensive support on the web.

The rules used by SQL to determine the updatability of a view, and how updates are reflected on the underlying database relations appeared in SQL:1999 and are summarized in [Melton and Simon (2001)].

The original SQL proposals for assertions date back to [Astrahan et al. (1976)], [Chamberlin et al. (1976)], and [Chamberlin et al. (1981)].

Bibliography

- [**Astrahan et al. (1976)**] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, “System R, A Relational Approach to Data Base

Management”, *ACM Transactions on Database Systems*, Volume 1, Number 2 (1976), pages 97–137.

[Chamberlin et al. (1976)] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, “SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control”, *IBM Journal of Research and Development*, Volume 20, Number 6 (1976), pages 560–575.

[Chamberlin et al. (1981)] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, “A History and Evaluation of System R”, *Communications of the ACM*, Volume 24, Number 10 (1981), pages 632–646.

[Melton and Simon (2001)] J. Melton and A. R. Simon, *SQL:1999, Understanding Relational Language Components*, Morgan Kaufmann (2001).

Credits

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

CHAPTER 4



Intermediate SQL

Practice Exercises

- 4.1 Consider the following SQL query that seeks to find a list of titles of all courses taught in Spring 2017 along with the name of the instructor.

```
select name, title  
from instructor natural join teaches natural join section natural join course  
where semester = 'Spring' and year = 2017
```

What is wrong with this query?

Answer:

Although the query is syntactically correct, it does not compute the expected answer because *dept_name* is an attribute of both *course* and *instructor*. As a result of the natural join, results are shown only when an instructor teaches a course in her or his own department.

- 4.2 Write the following queries in SQL:

- Display a list of all instructors, showing each instructor's ID and the number of sections taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use subqueries.
- Write the same query as in part a, but using a scalar subquery and not using outer join.
- Display the list of all course sections offered in Spring 2018, along with the ID and name of each instructor teaching the section. If a section has more than one instructor, that section should appear as many times in the result as it has instructors. If a section does not have any instructor, it should still appear in the result with the instructor name set to “—”.

- d. Display the list of all departments, with the total number of instructors in each department, without using subqueries. Make sure to show departments that have no instructors, and list those departments with an instructor count of zero.

Answer:

- a. Display a list of all instructors, showing each instructor's ID and the number of sections taught. Make sure to show the number of sections as 0 for instructors who have not taught any section. Your query should use an outer join, and should not use subqueries.

```
select ID, count(sec_id) as Number_of_sections
  from instructor natural left outer join teaches
       group by ID
```

The above query should not be written using `count(*)` since that would count null values also. It could be written using any attribute from *teaches* which does not occur in *instructor*, which would be correct although it may be confusing to the reader. (Attributes that occur in *instructor* would not be null even if the instructor has not taught any section.)

- b. Write the same query as above, but using a scalar subquery, and not using outerjoin.

```
select ID,
       (select count(*) as Number_of_sections
        from teaches T where T.id = I.id)
     from instructor I
```

- c. Display the list of all course sections offered in Spring 2018, along with the ID and name of each instructor teaching the section. If a section has more than one instructor, that section should appear as many times in the result as it has instructors. If a section does not have any instructor, it should still appear in the result with the instructor name set to “—”.

```
select course_id, sec_id, ID,
       decode(name, null, '—', name) as name
  from (section natural left outer join teaches)
       natural left outer join instructor
 where semester='Spring' and year= 2018
```

The query may also be written using the `coalesce` operator, by replacing `decode(..)` with `coalesce(name, '—')`. A more complex version of the query can be written using union of join result with another query that uses a subquery to find courses that do not match; refer to Exercise 4.3.

- d. Display the list of all departments, with the total number of instructors in each department, without using subqueries. Make sure to show departments that have no instructors, and list those departments with an instructor count of zero.

```
select dept_name, count(ID)
from department natural left outer join instructor
group by dept_name
```

- 4.3 Outer join expressions can be computed in SQL without using the SQL **outer join** operation. To illustrate this fact, show how to rewrite each of the following SQL queries without using the **outer join** expression.

- select * from student natural left outer join takes**
- select * from student natural full outer join takes**

Answer:

- a. **select * from student natural left outer join takes**
can be rewritten as:

```
select * from student natural join takes
union
select ID, name, dept_name, tot_cred, null, null, null, null, null
from student S1 where not exists
(select ID from takes T1 where T1.id = S1.id)
```

- b. **select * from student natural full outer join takes**
can be rewritten as:

```
(select * from student natural join takes)
union
(select ID, name, dept_name, tot_cred, null, null, null, null, null
from student S1
where not exists
(select ID from takes T1 where T1.id = S1.id))
union
(select ID, null, null, null, course_id, sec_id, semester, year, grade
from takes T1
where not exists
(select ID from student S1 where T1.id = S1.id))
```

- 4.4 Suppose we have three relations $r(A, B)$, $s(B, C)$, and $t(B, D)$, with all attributes declared as **not null**.

- Give instances of relations r , s , and t such that in the result of $(r \text{ natural left outer join } s) \text{ natural left outer join } t$ attribute C has a null value but attribute D has a non-null value.

- b. Are there instances of r , s , and t such that the result of
 $r \text{ natural left outer join } (s \text{ natural left outer join } t)$
has a null value for C but a non-null value for D ? Explain why or why not.

Answer:

- a. Consider $r = (a, b)$, $s = (b1, c1)$, $t = (b, d)$. The second expression would give $(a, b, null, d)$.
- b. Since $s \text{ natural left outer join } t$ is computed first, the absence of nulls in both s and t implies that each tuple of the result can have D null, but C can never be null.

4.5 Testing SQL queries: To test if a query specified in English has been correctly written in SQL, the SQL query is typically executed on multiple test databases, and a human checks if the SQL query result on each test database matches the intention of the specification in English.

- a. In Section 4.1.1 we saw an example of an erroneous SQL query which was intended to find which courses had been taught by each instructor; the query computed the natural join of *instructor*, *teaches*, and *course*, and as a result it unintentionally equated the *dept_name* attribute of *instructor* and *course*. Give an example of a dataset that would help catch this particular error.
- b. When creating test databases, it is important to create tuples in referenced relations that do not have any matching tuple in the referencing relation for each foreign key. Explain why, using an example query on the university database.
- c. When creating test databases, it is important to create tuples with null values for foreign-key attributes, provided the attribute is nullable (SQL allows foreign-key attributes to take on null values, as long as they are not part of the primary key and have not been declared as **not null**). Explain why, using an example query on the university database.

Hint: Use the queries from Exercise 4.2.

Answer:

- a. Consider the case where a professor in the Physics department teaches an Elec. Eng. course. Even though there is a valid corresponding entry in *teaches*, it is lost in the natural join of *instructor*, *teaches* and *course*, since the instructor's department name does not match the department name of the course. A dataset corresponding to the same is:

```

instructor = {('12345', 'Gauss', 'Physics', 10000)}
teaches = {('12345', 'EE321', 1, 'Spring', 2017)}
course = {'EE321', 'Magnetism', 'Elec. Eng.', 6)}

```

- b. The query in question 4.2(a) is a good example for this. Instructors who have not taught a single course should have number of sections as 0 in the query result. (Many other similar examples are possible.)
- c. Consider the query

```
select * from teaches natural join instructor;
```

In this query, we would lose some sections if *teaches.ID* is allowed to be *null* and such tuples exist. If, just because *teaches.ID* is a foreign key to *instructor*, we did not create such a tuple, the error in the above query would not be detected.

- 4.6** Show how to define the view *student_grades* (*ID, GPA*) giving the grade-point average of each student, based on the query in Exercise 3.2; recall that we used a relation *grade_points(grade, points)* to get the numeric points associated with a letter grade. Make sure your view definition correctly handles the case of *null* values for the *grade* attribute of the *takes* relation.

Answer:

We should not add credits for courses with a null grade; further, to correctly handle the case where a student has not completed any course, we should make sure we don't divide by zero, and should instead return a null value.

We break the query into a subquery that finds sum of credits and sum of credit-grade-points, taking null grades into account. The outer query divides the above to get the average, taking care of divide by zero.

```

create view student_grades(ID, GPA) as
  select ID, credit_points / decode(credit_sum, 0, null, credit_sum)
  from ((select ID, sum(decode(grade, null, 0, credits)) as credit_sum,
            sum(decode(grade, null, 0, credits*points)) as credit_points
            from(takes natural join course) natural left outer join grade_points
            group by ID)
  union
  select ID, null, null
  from student
  where ID not in (select ID from takes))

```

The view defined above takes care of *null* grades by considering the credit points to be 0 and not adding the corresponding credits in *credit_sum*.

```

employee (ID, person_name, street, city)
works (ID, company_name, salary)
company (company_name, city)
manages (ID, manager_id)

```

Figure 4.12 Employee database.

The query above ensures that a student who has not taken any course with non-null credits, and has *credit_sum* = 0 gets a GPA of *null*. This avoids the division by zero, which would otherwise have resulted.

In systems that do not support **decode**, an alternative is the **case** construct. Using **case**, the solution would be written as follows:

```

create view student_grades(ID, GPA) as
  select ID, credit_points / (case when credit_sum = 0 then null
                                else credit_sum end)
    from ((select ID, sum (case when grade is null then 0
                                    else credits end) as credit_sum,
                sum (case when grade is null then 0
                                    else credits*points end) as credit_points
              from(takes natural join course) natural left outer join grade_points
                group by ID)
union
  select ID, null, null
    from student
   where ID not in (select ID from takes))

```

An alternative way of writing the above query would be to use *student natural left outer join gpa*, in order to consider students who have not taken any course.

- 4.7** Consider the employee database of Figure 4.12. Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

Answer:

Please see ??.

Note that alternative data types are possible. Other choices for **not null** attributes may be acceptable.

- 4.8** As discussed in Section 4.4.8, we expect the constraint “an instructor cannot teach sections in two different classrooms in a semester in the same time slot” to hold.

```
create table employee
  (ID          numeric(6,0),
   person_name  char(20),
   street       char(30),
   city         char(30),
   primary key (ID))

create table works
  (ID          numeric(6,0),
   company_name char(15),
   salary       integer,
   primary key (ID),
   foreign key (ID) references employee,
   foreign key (company_name) references company)

create table company
  (company_name char(15),
   city         char(30),
   primary key (company_name))

create table manages
  (ID          numeric(6,0),
   manager_id   numeric(6,0),
   primary key (ID),
   foreign key (ID) references employee,
   foreign key (manager_id) references employee(ID))
```

Figure 4.101 Figure for Exercise 4.7.

- a. Write an SQL query that returns all (*instructor*, *section*) combinations that violate this constraint.
- b. Write an SQL assertion to enforce this constraint (as discussed in Section 4.4.8, current generation database systems do not support such assertions, although they are part of the SQL standard).

Answer:

a. Query:

```
select      ID, name, sec_id, semester, year, time_slot_id,
              count(distinct building, room_number)
from       instructor natural join teaches natural join section
group by   (ID, name, sec_id, semester, year, time_slot_id)
having    count(building, room_number) > 1
```

Note that the **distinct** keyword is required above. This is to allow two different sections to run concurrently in the same time slot and are taught by the same instructor without being reported as a constraint violation.

b. Query:

```
create assertion check not exists
( select ID, name, sec_id, semester, year, time_slot_id,
      count(distinct building, room_number)
from       instructor natural join teaches natural join section
group by   (ID, name, sec_id, semester, year, time_slot_id)
having    count(building, room_number) > 1)
```

- 4.9** SQL allows a foreign-key dependency to refer to the same relation, as in the following example:

```
create table manager
( employee_ID      char(20),
  manager_ID       char(20),
  primary key employee_ID,
  foreign key (manager_ID) references manager(employee_ID)
                           on delete cascade )
```

Here, *employee_ID* is a key to the table *manager*, meaning that each employee has at most one manager. The foreign-key clause requires that every manager also be an employee. Explain exactly what happens when a tuple in the relation *manager* is deleted.

Answer:

The tuples of all employees of the manager, at all levels, get deleted as well! This happens in a series of steps. The initial deletion will trigger deletion of all the tuples corresponding to direct employees of the manager. These deletions will in turn cause deletions of second-level employee tuples, and so on, till all direct and indirect employee tuples are deleted.

- 4.10** Given the relations *a*(*name, address, title*) and *b*(*name, address, salary*), show how to express *a* **natural full outer join** *b* using the **full outer-join** operation with an **on** condition rather than using the **natural join** syntax. This can be done using the **coalesce** operation. Make sure that the result relation does not contain two

copies of the attributes *name* and *address* and that the solution is correct even if some tuples in *a* and *b* have null values for attributes *name* or *address*.

Answer:

```
select coalesce(a.name, b.name) as name,
       coalesce(a.address, b.address) as address,
          a.title,
          b.salary
  from a full outer join b on a.name = b.name and
                           a.address = b.address
```

- 4.11** Operating systems usually offer only two types of authorization control for data files: read access and write access. Why do database systems offer so many kinds of authorization?

Answer: There are many reasons—we list a few here. One might wish to allow a user only to append new information without altering old information. One might wish to allow a user to access a relation but not change its schema. One might wish to limit access to aspects of the database that are not technically data access but instead impact resource utilization, such as creating an index.

- 4.12** Suppose a user wants to grant **select** access on a relation to another user. Why should the user include (or not include) the clause **granted by current role** in the **grant** statement?

Answer: Both cases give the same authorization at the time the statement is executed, but the long-term effects differ. If the grant is done based on the role, then the grant remains in effect even if the user who performed the grant leaves and that user's account is terminated. Whether that is a good or bad idea depends on the specific situation, but usually granting through a role is more consistent with a well-run enterprise.

- 4.13** Consider a view *v* whose definition references only relation *r*.

- If a user is granted **select** authorization on *v*, does that user need to have **select** authorization on *r* as well? Why or why not?
- If a user is granted **update** authorization on *v*, does that user need to have **update** authorization on *r* as well? Why or why not?
- Give an example of an **insert** operation on a view *v* to add a tuple *t* that is not visible in the result of **select * from v**. Explain your answer.

Answer:

- No. This allows a user to be granted access to only part of relation *r*.

- Yes. A valid update issued using view v must update r for the update to be stored in the database.
- Any tuple t compatible with the schema for v but not satisfying the **where** clause in the definition of view v is a valid example. One such example appears in Section 4.2.4.

CHAPTER 5



Advanced SQL

Chapter 3 and Chapter 4 provided detailed coverage of the basic structure of SQL. In this chapter, we first address the issue of how to access SQL from a general-purpose programming language, which is very important for building applications that use a database to manage data. We then cover some of the more advanced features of SQL, starting with how procedural code can be executed within the database either by extending the SQL language to support procedural actions or by allowing functions defined in procedural languages to be executed within the database. We describe triggers, which can be used to specify actions that are to be carried out automatically on certain events such as insertion, deletion, or update of tuples in a specified relation. Finally, we discuss recursive queries and advanced aggregation features supported by SQL.

5.1 Accessing SQL from a Programming Language

SQL provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general-purpose programming language. However, a database programmer must have access to a general-purpose programming language for at least two reasons:

1. Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Python that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.
2. Nondeclarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL. Applications usually have several components, and querying or updating data are only one component; other components are written in general-purpose programming languages. For an integrated application, there must be a means to combine SQL with a general-purpose programming language.

There are two approaches to accessing SQL from a general-purpose programming language:

1. **Dynamic SQL:** A general-purpose program can connect to and communicate with a database server using a collection of functions (for procedural languages) or methods (for object-oriented languages). Dynamic SQL allows the program to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time. The *dynamic SQL* component of SQL allows programs to construct and submit SQL queries at runtime.

In this chapter, we look at two standards for connecting to an SQL database and performing queries and updates. One, JDBC (Section 5.1.1), is an application program interface for the Java language. The other, ODBC (Section 5.1.3), is an application program interface originally developed for the C language, and subsequently extended to other languages such as C++, C#, Ruby, Go, PHP, and Visual Basic. We also illustrate how programs written in Python can connect to a database using the Python Database API (Section 5.1.2).

The ADO.NET API, designed for the Visual Basic .NET and C# languages, provides functions to access data, which at a high level are similar to the JDBC functions, although details differ. The ADO.NET API can also be used with some kinds of non-relational data sources. Details of ADO.NET may be found in the manuals available online and are not covered further in this chapter.

2. **Embedded SQL:** Like dynamic SQL, embedded SQL provides a means by which a program can interact with a database server. However, under embedded SQL, the SQL statements are identified at compile time using a preprocessor, which translates requests expressed in embedded SQL into function calls. At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities but may be specific to the database that is being used. Section 5.1.4 briefly covers embedded SQL.

A major challenge in mixing SQL with a general-purpose language is the mismatch in the ways these languages manipulate data. In SQL, the primary type of data are relations. SQL statements operate on relations and return relations as a result. Programming languages normally operate on a variable at a time, and those variables correspond roughly to the value of an attribute in a tuple in a relation. Thus, integrating these two types of languages into a single application requires providing a mechanism to return the result of a query in a manner that the program can handle.

Our examples in this section assume that we are accessing a database on a server that runs a database system. An alternative approach using an **embedded database** is discussed in Note 5.1 on page 198.

5.1.1 JDBC

The **JDBC** standard defines an **application program interface (API)** that Java programs can use to connect to database servers. (The word JDBC was originally an abbreviation for **Java Database Connectivity**, but the full form is no longer used.)

Figure 5.1 shows example Java code that uses the JDBC interface. The Java program must import `java.sql.*`, which contains the interface definitions for the functionality provided by JDBC.

5.1.1.1 Connecting to the Database

The first step in accessing a database from a Java program is to open a connection to the database. This step is required to select which database to use, such as an instance of Oracle running on your machine, or a PostgreSQL database running on another machine. Only after opening a connection can a Java program execute SQL statements.

```
public static void JDBCExample(String userid, String passwd)
{
    try {
        Connection conn = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:1521:univdb",
            userid, passwd);
        Statement stmt = conn.createStatement();
    } {
        try {
            stmt.executeUpdate(
                "insert into instructor values('77987','Kim','Physics',98000)");
        }
        catch (SQLException sqle) {
            System.out.println("Could not insert tuple. " + sqle);
        }
        ResultSet rset = stmt.executeQuery(
            "select dept_name, avg (salary) "+
            " from instructor "+
            " group by dept_name");
        while (rset.next()) {
            System.out.println(rset.getString("dept_name") + " " +
                rset.getFloat(2));
        }
    } {
        catch (Exception sqle)
        {
            System.out.println("Exception : " + sqle);
        }
    }
}
```

Figure 5.1 An example of JDBC code.

A connection is opened using the `getConnection()` method of the `DriverManager` class (within `java.sql`). This method takes three parameters.¹

1. The first parameter to the `getConnection()` call is a string that specifies the URL, or machine name, where the server runs (in our example, `db.yale.edu`), along with possibly some other information such as the protocol to be used to communicate with the database (in our example, `jdbc:oracle:thin:`; we shall shortly see why this is required), the port number the database system uses for communication (in our example, `2000`), and the specific database on the server to be used (in our example, `univdb`). Note that JDBC specifies only the API, not the communication protocol. A JDBC driver may support multiple protocols, and we must specify one supported by both the database and the driver. The protocol details are vendor specific.
2. The second parameter to `getConnection()` is a database user identifier, which is a string.
3. The third parameter is a password, which is also a string. (Note that the need to specify a password within the JDBC code presents a security risk if an unauthorized person accesses your Java code.)

In our example in the figure, we have created a `Connection` object whose handle is `conn`.

Each database product that supports JDBC (all the major database vendors do) provides a JDBC driver that must be dynamically loaded in order to access the database from Java. In fact, loading the driver must be done first, before connecting to the database. If the appropriate driver has been downloaded from the vendor's web site and is in the classpath, the `getConnection()` method will locate the needed driver.² The driver provides for the translation of product-independent JDBC calls into the product-specific calls needed by the specific database management system being used. The actual protocol used to exchange information with the database depends on the driver that is used, and it is not defined by the JDBC standard. Some drivers support more than one protocol, and a suitable protocol must be chosen depending on what protocol the particular database product supports. In our example, when opening a connection with the database, the string `jdbc:oracle:thin:` specifies a particular protocol supported by Oracle. The MySQL equivalent is `jdbc:mysql`:

5.1.1.2 Shipping SQL Statements to the Database System

Once a database connection is open, the program can use it to send SQL statements to the database system for execution. This is done via an instance of the class `Statement`.

¹There are multiple versions of the `getConnection()` method, which differ in the parameters that they accept. We present the most commonly used version.

²Prior to version 4, locating the driver was done manually by invoking `Class.forName` with one argument specifying a concrete class implementing the `java.sql.Driver` interface, in a line of code prior to the `getConnection` call.

A `Statement` object is not the SQL statement itself, but rather an object that allows the Java program to invoke methods that ship an SQL statement given as an argument for execution by the database system. Our example creates a `Statement` handle (`stmt`) on the connection `conn`.

To execute a statement, we invoke either the `executeQuery()` method or the `executeUpdate()` method, depending on whether the SQL statement is a query (and, thus, returns a result set) or nonquery statement such as `update`, `insert`, `delete`, or `create table`. In our example, `stmt.executeUpdate()` executes an update statement that inserts into the *instructor* relation. It returns an integer giving the number of tuples inserted, updated, or deleted. For DDL statements, the return value is zero.

5.1.1.3 Exceptions and Resource Management

Executing any SQL method might result in an exception being thrown. The `try { ... } catch { ... }` construct permits us to catch any exceptions (error conditions) that arise when JDBC calls are made and take appropriate action. In JDBC programming, it may be useful to distinguish between an `SQLException`, which is an SQL-specific exception, and the general case of an `Exception`, which could be any Java exception such as a null-pointer exception, or array-index-out-of-bounds exception. We show both in Figure 5.1. In practice, one would write more complete exception handlers than we do (for the sake of conciseness) in our example code.

Opening a connection, a statement, and other JDBC objects are all actions that consume system resources. Programmers must take care to ensure that programs close all such resources. Failure to do so may cause the database system's resource pools to become exhausted, rendering the system inaccessible or inoperative until a time-out period expires. One way to do this is to code explicit calls to close connections and statements. This approach fails if the code exits due to an exception and, in so doing, avoids the Java statement with the `close` invocation. For this reason, the preferred approach is to use the *try-with-resources* construct in Java. In the example of Figure 5.1, the opening of the connection and statement objects is done within parentheses rather than in the main body of the `try` in curly braces. Resources opened in the code within parentheses are closed automatically at the end of the `try` block. This protects us from leaving connections or statements unclosed. Since closing a statement implicitly closes objects opened for that statement (i.e., the `ResultSet` objects we shall discuss in the next section, this coding practice protects us from leaving resources unclosed.³ In the example of Figure 5.1, we could have closed the connection explicitly with the statement `conn.close()` and closed the statement explicitly with `stmt.close()`, though doing so was not necessary in our example.

5.1.1.4 Retrieving the Result of a Query

The example code of Figure 5.1 executes a query by using `stmt.executeQuery()`. It retrieves the set of tuples in the result into a `ResultSet` object `rset` and fetches them one

³This Java feature, called *try-with-resources*, was introduced in Java 7.

tuple at a time. The `next()` method on the result set tests whether or not there remains at least one unfetched tuple in the result set and if so, fetches it. The return value of the `next()` method is a Boolean indicating whether it fetched a tuple. Attributes from the fetched tuple are retrieved using various methods whose names begin with `get`. The method `getString()` can retrieve any of the basic SQL data types (converting the value to a Java String object), but more restrictive methods such as `getFloat()` can be used as well. The argument to the various `get` methods can either be an attribute name specified as a string, or an integer indicating the position of the desired attribute within the tuple. Figure 5.1 shows two ways of retrieving the values of attributes in a tuple: using the name of the attribute (`dept_name`) and using the position of the attribute (2, to denote the second attribute).

5.1.1.5 Prepared Statements

We can create a prepared statement in which some values are replaced by “?”, thereby specifying that actual values will be provided later. The database system compiles the query when it is prepared. Each time the query is executed (with new values to replace the “?”s), the database system can reuse the previously compiled form of the query and apply the new values as parameters. The code fragment in Figure 5.2 shows how prepared statements can be used.

The `prepareStatement()` method of the `Connection` class defines a query that may contain parameter values; some JDBC drivers may submit the query to the database for compilation as part of the method, but other drivers do not contact the database at this point. The method returns an object of class `PreparedStatement`. At this point, no SQL statement has been executed. The `executeQuery()` and `executeUpdate()` methods of `PreparedStatement` class do that. But before they can be invoked, we must use methods of class `PreparedStatement` that assign values for the “?” parameters. The `setString()` method and other similar methods such as `setInt()` for other basic SQL types allow us to specify the values for the parameters. The first argument specifies the “?” parameter for which we are assigning a value (the first parameter is 1, unlike most other Java constructs, which start with 0). The second argument specifies the value to be assigned.

In the example in Figure 5.2, we prepare an `insert` statement, set the “?” parameters, and then invoke `executeUpdate()`. The final two lines of our example show that parameter assignments remain unchanged until we specifically reassign them. Thus, the final statement, which invokes `executeUpdate()`, inserts the tuple (“88878”, “Perry”, “Finance”, 125000).

Prepared statements allow for more efficient execution in cases where the same query can be compiled once and then run multiple times with different parameter values. However, there is an even more significant advantage to prepared statements that makes them the preferred method of executing SQL queries whenever a user-entered value is used, even if the query is to be run only once. Suppose that we read in a user-entered value and then use Java string manipulation to construct the SQL statement.

```

PreparedStatement pStmt = conn.prepareStatement(
        "insert into instructor values(?, ?, ?, ?)");
pStmt.setString(1, "88877");
pStmt.setString(2, "Perry");
pStmt.setString(3, "Finance");
pStmt.setInt(4, 125000);
pStmt.executeUpdate();
pStmt.setString(1, "88878");
pStmt.executeUpdate();

```

Figure 5.2 Prepared statements in JDBC code.

If the user enters certain special characters, such as a single quote, the resulting SQL statement may be syntactically incorrect unless we take extraordinary care in checking the input. The `setString()` method does this for us automatically and inserts the needed escape characters to ensure syntactic correctness.

In our example, suppose that the values for the variables `ID`, `name`, `dept_name`, and `salary` have been entered by a user, and a corresponding row is to be inserted into the *instructor* relation. Suppose that, instead of using a prepared statement, a query is constructed by concatenating the strings using the following Java expression:

```

"insert into instructor values(' " + ID + " ', ' " + name + " ', " +
    " " + dept_name + " ', " + salary + ")"

```

and the query is executed directly using the `executeQuery()` method of a `Statement` object. Observe the use of single quotes in the string, which would surround the values of `ID`, `name` and `dept_name` in the generated SQL query.

Now, if the user typed a single quote in the `ID` or `name` fields, the query string would have a syntax error. It is quite possible that an instructor name may have a quotation mark in its name (for example, “O’Henry”).

While the above example might be considered an annoyance, the situation can be much worse. A technique called **SQL injection** can be used by malicious hackers to steal data or damage the database.

Suppose a Java program inputs a string `name` and constructs the query:

```

"select * from instructor where name = " " + name + " "

```

If the user, instead of entering a name, enters:

$X' \text{ or } Y' = Y$

then the resulting statement becomes:

```
"select * from instructor where name = "" + "X" or 'Y' = 'Y" + """
```

which is:

```
select * from instructor where name = 'X' or 'Y' = 'Y'
```

In the resulting query, the **where** clause is always true and the entire instructor relation is returned.

More clever malicious users could arrange to output even more data, including credentials such as passwords that allow the user to connect to the database and perform any actions they want. SQL injection attacks on **update** statements can be used to change the values that are being stored in updated columns. In fact there have been a number of attacks in the real world using SQL injections; attacks on multiple financial sites have resulted in theft of large amounts of money by using SQL injection attacks.

Use of a prepared statement would prevent this problem because the input string would have escape characters inserted, so the resulting query becomes:

```
"select * from instructor where name = 'X\' or \'Y\' = \'Y'
```

which is harmless and returns the empty relation.

Programmers must pass user-input strings to the database only through parameters of prepared statements; creating SQL queries by concatenating strings with user-input values is an extremely serious security risk and should never be done in any program.

Some database systems allow multiple SQL statements to be executed in a single JDBC **execute** method, with statements separated by a semicolon. This feature has been turned off by default on some JDBC drivers because it allows malicious hackers to insert whole SQL statements using SQL injection. For instance, in our earlier SQL injection example a malicious user could enter:

```
X'; drop table instructor; --
```

which will result in a query string with two statements separated by a semicolon being submitted to the database. Because these statements run with the privileges of the database userid used by the JDBC connection, devastating SQL statements such as **drop table**, or updates to any table of the user's choice, could be executed. However, some databases still allow execution of multiple statements as above; it is thus very important to correctly use prepared statements to avoid the risk of SQL injection.

5.1.1.6 Callable Statements

JDBC also provides a **CallableStatement** interface that allows invocation of SQL stored procedures and functions (described in Section 5.2). These play the same role for functions and procedures as **prepareStatement** does for queries.

```
CallableStatement cStmt1 = conn.prepareCall("{? = call some_function(?)}");
CallableStatement cStmt2 = conn.prepareCall("{call some_procedure(? ,?)}");
```

The data types of function return values and out parameters of procedures must be registered using the method `registerOutParameter()`, and can be retrieved using get methods similar to those for result sets. See a JDBC manual for more details.

5.1.1.7 Metadata Features

As we noted earlier, a Java application program does not include declarations for data stored in the database. Those declarations are part of the SQL DDL statements. Therefore, a Java program that uses JDBC must either have assumptions about the database schema hard-coded into the program or determine that information directly from the database system at runtime. The latter approach is usually preferable, since it makes the application program more robust to changes in the database schema.

Recall that when we submit a query using the `executeQuery()` method, the result of the query is contained in a `ResultSet` object. The interface `ResultSet` has a method, `getMetaData()`, that returns a `ResultSetMetaData` object that contains metadata about the result set. `ResultSetMetaData`, in turn, has methods to find metadata information, such as the number of columns in the result, the name of a specified column, or the type of a specified column. In this way, we can write code to execute a query even if we have no prior knowledge of the schema of the result.

The following Java code segment uses JDBC to print out the names and types of all columns of a result set. The variable `rs` in the code is assumed to refer to a `ResultSet` instance obtained by executing a query.

```
ResultSetMetaData rsmd = rs.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}
```

The `getColumnCount()` method returns the arity (number of attributes) of the result relation. That allows us to iterate through each attribute (note that we start at 1, as is conventional in JDBC). For each attribute, we retrieve its name and data type using the methods `getColumnName()` and `getColumnTypeName()`, respectively.

The `DatabaseMetaData` interface provides a way to find metadata about the database. The interface `Connection` has a method `getMetaData()` that returns a `DatabaseMetaData` object. The `DatabaseMetaData` interface in turn has a very large number of methods to get metadata about the database and the database system to which the application is connected.

For example, there are methods that return the product name and version number of the database system. Other methods allow the application to query the database system about its supported features.

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns(null, "univdb", "department", "%");
    // Arguments to getColumns: Catalog, Schema-pattern, Table-pattern,
    //           and Column-Pattern
    // Returns: One row for each column; row has a number of attributes
    //           such as COLUMN_NAME, TYPE_NAME
while( rs.next()) {
    System.out.println(rs.getString("COLUMN_NAME"),
                      rs.getString("TYPE_NAME"));
}
```

Figure 5.3 Finding column information in JDBC using DatabaseMetaData.

Still other methods return information about the database itself. The code in Figure 5.3 illustrates how to find information about columns (attributes) of relations in a database. The variable `conn` is assumed to be a handle for an already opened database connection. The method `getColumns()` takes four arguments: a catalog name (null signifies that the catalog name is to be ignored), a schema name pattern, a table name pattern, and a column name pattern. The schema name, table name, and column name patterns can be used to specify a name or a pattern. Patterns can use the SQL string matching special characters “%” and “_”; for instance, the pattern “%” matches all names. Only columns of tables of schemas satisfying the specified name or pattern are retrieved. Each row in the result set contains information about one column. The rows have a number of columns such as the name of the catalog, schema, table and column, the type of the column, and so on.

The `getTables()` method allows you to get a list of all tables in the database. The first three parameters to `getTables()` are the same as for `getColumns()`. The fourth parameter can be used to restrict the types of tables returned; if set to null, all tables, including system internal tables are returned, but the parameter can be set to restrict the tables returned to only user-created tables.

Examples of other methods provided by `DatabaseMetaData` that provide information about the database include those for primary keys (`getPrimaryKeys()`), foreign-key references (`getCrossReference()`), authorizations, database limits such as maximum number of connections, and so on.

The metadata interfaces can be used for a variety of tasks. For example, they can be used to write a database browser that allows a user to find the tables in a database, examine their schema, examine rows in a table, apply selections to see desired rows, and so on. The metadata information can be used to make code used for these tasks generic; for example, code to display the rows in a relation can be written in such a way that it would work on all possible relations regardless of their schema. Similarly, it is

possible to write code that takes a query string, executes the query, and prints out the results as a formatted table; the code can work regardless of the actual query submitted.

5.1.1.8 Other Features

JDBC provides a number of other features, such as **updatable result sets**. It can create an updatable result set from a query that performs a selection and/or a projection on a database relation. An update to a tuple in the result set then results in an update to the corresponding tuple of the database relation.

Recall from Section 4.3 that a transaction allows multiple actions to be treated as a single atomic unit which can be committed or rolled back. By default, each SQL statement is treated as a separate transaction that is committed automatically. The method `setAutoCommit()` in the JDBC `Connection` interface allows this behavior to be turned on or off. Thus, if `conn` is an open connection, `conn.setAutoCommit(false)` turns off automatic commit. Transactions must then be committed or rolled back explicitly using either `conn.commit()` or `conn.rollback()`. `conn.setAutoCommit(true)` turns on automatic commit.

JDBC provides interfaces to deal with large objects without requiring an entire large object to be created in memory. To fetch large objects, the `ResultSet` interface provides methods `getBlob()` and `getClob()` that are similar to the `getString()` method, but return objects of type `Blob` and `Clob`, respectively. These objects do not store the entire large object, but instead store “locators” for the large objects, that is, logical pointers to the actual large object in the database. Fetching data from these objects is very much like fetching data from a file or an input stream, and it can be performed using methods such as `getBytes()` and `getSubString()`.

Conversely, to store large objects in the database, the `PreparedStatement` class permits a database column whose type is **blob** to be linked to an input stream (such as a file that has been opened) using the method `setBlob(int parameterIndex, InputStream inputStream)`. When the prepared statement is executed, data are read from the input stream and written to the **blob** in the database. Similarly, a **clob** column can be set using the `setClob()` method, which takes as arguments a parameter index and a character stream.

JDBC includes a *row set* feature that allows result sets to be collected and shipped to other applications. Row sets can be scanned both backward and forward and can be modified.

5.1.2 Database Access from Python

Database access can be done from Python as illustrated by the method shown in Figure 5.4. The statement containing the `insert` query shows how to use the Python equivalent of JDBC prepared statements, with parameters identified in the SQL query by “%s”, and parameter values provided as a list. Updates are not committed to the database automatically; the `commit()` method needs to be called to commit an update.

```
import psycopg2

def PythonDatabaseExample(userid, passwd):
    try:
        conn = psycopg2.connect( host="db.yale.edu", port=5432,
                               dbname="univdb", user=userid, password=passwd)
        cur = conn.cursor()
        try:
            cur.execute("insert into instructor values(%s, %s, %s, %s)",
                       ("77987","Kim","Physics",98000))
            conn.commit();
        except Exception as sqle:
            print("Could not insert tuple. ", sqle)
            conn.rollback()
        cur.execute( "select dept_name, avg (salary) "
                     " from instructor group by dept_name")
        for dept in cur:
            print dept[0], dept[1]
    except Exception as sqle:
        print("Exception : ", sqle)
```

Figure 5.4 Database access from Python

The `try:, except ...:` block shows how to catch exceptions and to print information about the exception. The `for` loop illustrates how to loop over the result of a query execution, and to access individual attributes of a particular row.

The preceding program uses the `psycopg2` driver, which allows connection to PostgreSQL databases and is imported in the first line of the program. Drivers are usually database specific, with the `MySQLdb` driver to connect to MySQL, and `cx_Oracle` to connect to Oracle; but the `pyodbc` driver can connect to most databases that support ODBC. The Python Database API used in the program is implemented by drivers for many databases, but unlike with JDBC, there are minor differences in the API across different drivers, in particular in the parameters to the `connect()` function.

5.1.3 ODBC

The **Open Database Connectivity (ODBC)** standard defines an API that applications can use to open a connection with a database, send queries and updates, and get back results. Applications such as graphical user interfaces, statistics packages, and spreadsheets can make use of the same ODBC API to connect to any database server that supports ODBC.

Each database system supporting ODBC provides a library that must be linked with the client program. When the client program makes an ODBC API call, the code

```
void ODBCexample()
{
    RETCODE error;
    HENV env; /* environment */
    HDBC conn; /* database connection */

    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
               "avipasswd", SQL_NTS);
{
    char deptname[80];
    float salary;
    int lenOut1, lenOut2;
    HSTMT stmt;

    char * sqlquery = "select dept_name, sum (salary)
                       from instructor
                       group by dept_name";
    SQLAllocStmt(conn, &stmt);
    error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
    if (error == SQL_SUCCESS) {
        SQLBindCol(stmt, 1, SQL_C_CHAR, deptname , 80, &lenOut1);
        SQLBindCol(stmt, 2, SQL_C_FLOAT, &salary, 0 , &lenOut2);
        while (SQLFetch(stmt) == SQL_SUCCESS) {
            printf (" %s %g\n", deptname, salary);
        }
    }
    SQLFreeStmt(stmt, SQL_DROP);
}
SQLDisconnect(conn);
SQLFreeConnect(conn);
SQLFreeEnv(env);
}
```

Figure 5.5 ODBC code example.

in the library communicates with the server to carry out the requested action and fetch results.

Figure 5.5 shows an example of C code using the ODBC API. The first step in using ODBC to communicate with a server is to set up a connection with the server. To do so, the program first allocates an SQL environment, then a database connection handle. ODBC defines the types HENV, HDBC, and RETCODE. The program then opens the

database connection by using `SQLConnect`. This call takes several parameters, including the connection handle, the server to which to connect, the user identifier, and the password for the database. The constant `SQL_NTS` denotes that the previous argument is a null-terminated string.

Once the connection is set up, the program can send SQL commands to the database by using `SQLExecDirect`. C language variables can be bound to attributes of the query result, so that when a result tuple is fetched using `SQLFetch`, its attribute values are stored in corresponding C variables. The `SQLBindCol` function does this task; the second argument identifies the position of the attribute in the query result, and the third argument indicates the type conversion required from SQL to C. The next argument gives the address of the variable. For variable-length types like character arrays, the last two arguments give the maximum length of the variable and a location where the actual length is to be stored when a tuple is fetched. A negative value returned for the length field indicates that the value is `null`. For fixed-length types such as integer or float, the maximum length field is ignored, while a negative value returned for the length field indicates a null value.

The `SQLFetch` statement is in a `while` loop that is executed until `SQLFetch` returns a value other than `SQL_SUCCESS`. On each fetch, the program stores the values in C variables as specified by the calls on `SQLBindCol` and prints out these values.

At the end of the session, the program frees the statement handle, disconnects from the database, and frees up the connection and SQL environment handles. Good programming style requires that the result of every function call must be checked to make sure there are no errors; we have omitted most of these checks for brevity.

It is possible to create an SQL statement with parameters; for example, consider the statement `insert into department values(?, ?, ?)`. The question marks are placeholders for values which will be supplied later. The above statement can be “prepared,” that is, compiled at the database, and repeatedly executed by providing actual values for the placeholders—in this case, by providing a department name, building, and budget for the relation `department`.

ODBC defines functions for a variety of tasks, such as finding all the relations in the database and finding the names and types of columns of a query result or a relation in the database.

By default, each SQL statement is treated as a separate transaction that is committed automatically. The `SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)` turns off automatic commit on connection `conn`, and transactions must then be committed explicitly by `SQLTransact(conn, SQL_COMMIT)` or rolled back by `SQLTransact(conn, SQL_ROLLBACK)`.

The ODBC standard defines *conformance levels*, which specify subsets of the functionality defined by the standard. An ODBC implementation may provide only core level features, or it may provide more advanced (level 1 or level 2) features. Level 1 requires support for fetching information about the catalog, such as information about what relations are present and the types of their attributes. Level 2 requires further fea-

tures, such as the ability to send and retrieve arrays of parameter values and to retrieve more detailed catalog information.

The SQL standard defines a **call level interface (CLI)** that is similar to the ODBC interface.

5.1.4 Embedded SQL

The SQL standard defines embeddings of SQL in a variety of programming languages, such as C, C++, Cobol, Pascal, Java, PL/I, and Fortran. A language in which SQL queries are embedded is referred to as a *host* language, and the SQL structures permitted in the host language constitute *embedded SQL*.

Programs written in the host language can use the embedded SQL syntax to access and update data stored in a database. An embedded SQL program must be processed by a special preprocessor prior to compilation. The preprocessor replaces embedded SQL requests with host-language declarations and procedure calls that allow runtime execution of the database accesses. Then the resulting program is compiled by the host-language compiler. This is the main distinction between embedded SQL and JDBC or ODBC.

To identify embedded SQL requests to the preprocessor, we use the EXEC SQL statement; it has the form:

```
EXEC SQL <embedded SQL statement>;
```

Before executing any SQL statements, the program must first connect to the database. Variables of the host language can be used within embedded SQL statements, but they must be preceded by a colon (:) to distinguish them from SQL variables.

To iterate over the results of an embedded SQL query, we must declare a *cursor* variable, which can then be opened, and *fetch* commands issued in a host language loop to fetch consecutive rows of the query result. Attributes of a row can be fetched into host language variables. Database updates can also be performed using a cursor on a relation to iterate through the rows of the relation, optionally using a **where** clause to iterate through only selected rows. Embedded SQL commands can be used to update the current row where the cursor is pointing.

The exact syntax for embedded SQL requests depends on the language in which SQL is embedded. You may refer to the manuals of the specific language embedding that you use for further details.

In JDBC, SQL statements are interpreted at runtime (even if they are created using the prepared statement feature). When embedded SQL is used, there is a potential for catching some SQL-related errors (including data-type errors) at the time of preprocessing. SQL queries in embedded SQL programs are also easier to comprehend than in programs using dynamic SQL. However, there are also some disadvantages with embedded SQL. The preprocessor creates new host language code, which may complicate debugging of the program. The constructs used by the preprocessor to identify SQL

Note 5.1 EMBEDDED DATABASES

Both JDBC and ODBC assume that a server is running on the database system hosting the database. Some applications use a database that exists entirely within the application. Such applications maintain the database only for internal use and offer no accessibility to the database except through the application itself. In such cases, one may use an **embedded database** and use one of several packages that implement an SQL database accessible from within a programming language. Popular choices include Java DB, SQLite, HSQLDB, and 2. There is also an embedded version of MySQL.

Embedded database systems lack many of the features of full server-based database systems, but they offer advantages for applications that can benefit from the database abstractions but do not need to support very large databases or large-scale transaction processing.

Do not confuse embedded databases with embedded SQL; the latter is a means of connecting to a database running on a server.

statements may clash syntactically with host language syntax introduced in subsequent versions of the host language.

As a result, most current systems use dynamic SQL, rather than embedded SQL. One exception is the Microsoft Language Integrated Query (LINQ) facility, which extends the host language to include support for queries instead of using a preprocessor to translate embedded SQL queries into the host language.

5.2

Functions and Procedures

We have already seen several functions that are built into the SQL language. In this section, we show how developers can write their own functions and procedures, store them in the database, and then invoke them from SQL statements. Functions are particularly useful with specialized data types such as images and geometric objects. For instance, a line-segment data type used in a map database may have an associated function that checks whether two line segments overlap, and an image data type may have associated functions to compare two images for similarity.

Procedures and functions allow “business logic” to be stored in the database and executed from SQL statements. For example, universities usually have many rules about how many courses a student can take in a given semester, the minimum number of courses a full-time instructor must teach in a year, the maximum number of majors a student can be enrolled in, and so on. While such business logic can be encoded as programming-language procedures stored entirely outside the database, defining them as stored procedures in the database has several advantages. For example, it allows

```

create function dept_count(dept_name varchar(20))
returns integer
begin
declare d_count integer;
select count(*) into d_count
from instructor
where instructor.dept_name= dept_name
return d_count;
end

```

Figure 5.6 Function defined in SQL.

multiple applications to access the procedures, and it allows a single point of change in case the business rules change, without changing other parts of the application. Application code can then call the stored procedures instead of directly updating database relations.

SQL allows the definition of functions, procedures, and methods. These can be defined either by the procedural component of SQL or by an external programming language such as Java, C, or C++. We look at definitions in SQL first and then see how to use definitions in external languages in Section 5.2.3.

Although the syntax we present here is defined by the SQL standard, most databases implement nonstandard versions of this syntax. For example, the procedural languages supported by Oracle (PL/SQL), Microsoft SQL Server (TransactSQL), and PostgreSQL (PL/pgSQL) all differ from the standard syntax we present here. We illustrate some of the differences for the case of Oracle in Note 5.2 on page 204. See the respective system manuals for further details. Although parts of the syntax we present here may not be supported on such systems, the concepts we describe are applicable across implementations, although with a different syntax.

5.2.1 Declaring and Invoking SQL Functions and Procedures

Suppose that we want a function that, given the name of a department, returns the count of the number of instructors in that department. We can define the function as shown in Figure 5.6.⁴ This function can be used in a query that returns names and budgets of all departments with more than 12 instructors:

```

select dept_name, budget
from department
where dept_count(dept_name) > 12;

```

⁴If you are entering your own functions or procedures, you should write “**create or replace**” rather than **create** so that it is easy to modify your code (by replacing the function) during debugging.

```
create function instructor_of (dept_name varchar(20))
    returns table (
        ID varchar (5),
        name varchar (20),
        dept_name varchar (20),
        salary numeric (8,2))
return table
    (select ID, name, dept_name, salary
     from instructor
     where instructor.dept_name = instructor_of.dept_name);
```

Figure 5.7 Table function in SQL.

Performance problems have been observed on many database systems when invoking complex user-defined functions within a query, if the functions are invoked on a large number of tuples. Programmers should therefore take performance into consideration when deciding whether to use user-defined functions in a query.

The SQL standard supports functions that can return tables as results; such functions are called **table functions**. Consider the function defined in Figure 5.7. The function returns a table containing all the instructors of a particular department. Note that the function's parameter is referenced by prefixing it with the name of the function (*instructor_of.dept_name*).

The function can be used in a query as follows:

```
select *
  from table(instructor_of('Finance'));
```

This query returns all instructors of the 'Finance' department. In this simple case it is straightforward to write this query without using table-valued functions. In general, however, table-valued functions can be thought of as **parameterized views** that generalize the regular notion of views by allowing parameters.

SQL also supports procedures. The *dept_count* function could instead be written as a procedure:

```
create procedure dept_count_proc(in dept_name varchar(20),
                                         out d_count integer)
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name= dept_count_proc.dept_name
end
```

The keywords **in** and **out** indicate, respectively, parameters that are expected to have values assigned to them and parameters whose values are set in the procedure in order to return results.

Procedures can be invoked either from an SQL procedure or from embedded SQL by the **call** statement:

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

Procedures and functions can be invoked from dynamic SQL, as illustrated by the JDBC syntax in Section 5.1.1.5.

SQL permits more than one procedure of the same name, so long as the number of arguments of the procedures with the same name is different. The name, along with the number of arguments, is used to identify the procedure. SQL also permits more than one function with the same name, so long as the different functions with the same name either have different numbers of arguments, or for functions with the same number of arguments, they differ in the type of at least one argument.

5.2.2 Language Constructs for Procedures and Functions

SQL supports constructs that give it almost all the power of a general-purpose programming language. The part of the SQL standard that deals with these constructs is called the **Persistent Storage Module (PSM)**.

Variables are declared using a **declare** statement and can have any valid SQL data type. Assignments are performed using a **set** statement.

A compound statement is of the form **begin** ... **end**, and it may contain multiple SQL statements between the **begin** and the **end**. Local variables can be declared within a compound statement, as we have seen in Section 5.2.1. A compound statement of the form **begin atomic** ... **end** ensures that all the statements contained within it are executed as a single transaction.

The syntax for **while** statements and **repeat** statements is:

```
while boolean expression do
    sequence of statements;
end while

repeat
    sequence of statements;
until boolean expression
end repeat
```

There is also a **for** loop that permits iteration over all the results of a query:

```

declare n integer default 0;
for r as
    select budget from department
        where dept_name = 'Music'
do
    set n = n - r.budget
end for

```

The program fetches the query results one row at a time into the **for** loop variable (*r*, in the above example). The statement **leave** can be used to exit the loop, while **iterate** starts on the next tuple, from the beginning of the loop, skipping the remaining statements.

The conditional statements supported by SQL include **if-then-else** statements by using this syntax:

```

if boolean expression
    then statement or compound statement
elseif boolean expression
    then statement or compound statement
else statement or compound statement
end if

```

SQL also supports a case statement similar to the C/C++ language case statement (in addition to case expressions, which we saw in Chapter 3).

Figure 5.8 provides a larger example of the use of procedural constructs in SQL. The function *registerStudent* defined in the figure registers a student in a course section after verifying that the number of students in the section does not exceed the capacity of the room allocated to the section. The function returns an error code—a value greater than or equal to 0 signifies success, and a negative value signifies an error condition—and a message indicating the reason for the failure is returned as an **out** parameter.

The SQL procedural language also supports the signaling of **exception conditions** and declaring of **handlers** that can handle the exception, as in this code:

```

declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
    sequence of statements
end

```

The statements between the **begin** and the **end** can raise an exception by executing **signal** *out_of_classroom_seats*. The handler says that if the condition arises, the action to be taken is to exit the enclosing **begin end** statement. Alternative actions would be **continue**, which continues execution from the next statement following the one that raised the exception. In addition to explicitly defined conditions, there are also predefined conditions such as **sqlexception**, **sqlwarning**, and **not found**.

-- Registers a student after ensuring classroom capacity is not exceeded
-- Returns 0 on success, and -1 if capacity is exceeded.

```

create function registerStudent(
    in s_id varchar(5),
    in s_courseid varchar (8),
    in s_secid varchar (8),
    in s_semester varchar (6),
    in s_year numeric (4,0),
    out errorMsg varchar(100)

returns integer
begin
    declare currEnrol int;
    select count(*) into currEnrol
        from takes
        where course_id = s_courseid and sec_id = s_secid
            and semester = s_semester and year = s_year;
    declare limit int;
    select capacity into limit
        from classroom natural join section
        where course_id = s_courseid and sec_id = s_secid
            and semester = s_semester and year = s_year;
    if (currEnrol < limit)
        begin
            insert into takes values
                (s_id, s_courseid, s_secid, s_semester, s_year, null);
            return(0);
        end
        -- Otherwise, section capacity limit already reached
        set errorMsg = 'Enrollment limit reached for course ' || s_courseid
            || ' section ' || s_secid;
        return(-1);
end;
```

Figure 5.8 Procedure to register a student for a course section.

5.2.3 External Language Routines

Although the procedural extensions to SQL can be very useful, they are unfortunately not supported in a standard way across databases. Even the most basic features have different syntax or semantics in different database products. As a result, programmers have to learn a new language for each database product. An alternative that is gaining

Note 5.2 NONSTANDARD SYNTAX FOR PROCEDURES AND FUNCTIONS

Although the SQL standard defines the syntax for procedures and functions, most databases do not follow the standard strictly, and there is considerable variation in the syntax supported. One of the reasons for this situation is that these databases typically introduced support for procedures and functions before the syntax was standardized, and they continue to support their original syntax. It is not possible to list the syntax supported by each database here, but we illustrate a few of the differences in the case of Oracle's PL/SQL by showing below a version of the function from Figure 5.6 as it would be defined in PL/SQL.

```
create function dept_count (dname in instructor.dept_name%type) return integer
as
d_count integer;
begin
    select count(*) into d_count
    from instructor
    where instructor.dept_name = dname;
return d_count;
end;
```

While the two versions are similar in concept, there are a number of minor syntactic differences, some of which are evident when comparing the two versions of the function. Although not shown here, the syntax for control flow in PL/SQL also has several differences from the syntax presented here.

Observe that PL/SQL allows a type to be specified as the type of an attribute of a relation, by adding the suffix `%type`. On the other hand, PL/SQL does not directly support the ability to return a table, although there is an indirect way of implementing this functionality by creating a table type. The procedural languages supported by other databases also have a number of syntactic and semantic differences. See the respective language references for more information. The use of nonstandard syntax for stored procedures and functions is an impediment to porting an application to a different database.

support is to define procedures in an imperative programming language, but allow them to be invoked from SQL queries and trigger definitions.

SQL allows us to define functions in a programming language such as Java, C#, C, or C++. Functions defined in this fashion can be more efficient than functions defined in SQL, and computations that cannot be carried out in SQL can be executed by these functions.

External procedures and functions can be specified in this way (note that the exact syntax depends on the specific database system you use):

```
create procedure dept_count_proc( in dept_name varchar(20),
                                  out count integer)
language C
external name '/usr/avi/bin/dept_count_proc'

create function dept_count (dept_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'
```

In general, the external language procedures need to deal with null values in parameters (both **in** and **out**) and return values. They also need to communicate failure/success status and to deal with exceptions. This information can be communicated by extra parameters: an **sqlstate** value to indicate failure/success status, a parameter to store the return value of the function, and indicator variables for each parameter/function result to indicate if the value is null. Other mechanisms are possible to handle null values, for example, by passing pointers instead of values. The exact mechanisms depend on the database. However, if a function does not deal with these situations, an extra line **parameter style general** can be added to the declaration to indicate that the external procedures/functions take only the arguments shown and do not handle null values or exceptions.

Functions defined in a programming language and compiled outside the database system may be loaded and executed with the database-system code. However, doing so carries the risk that a bug in the program can corrupt the internal structures of the database and can bypass the access-control functionality of the database system. Database systems that are concerned more about efficient performance than about security may execute procedures in such a fashion. Database systems that are concerned about security may execute such code as part of a separate process, communicate the parameter values to it, and fetch results back via interprocess communication. However, the time overhead of interprocess communication is quite high; on typical CPU architectures, tens to hundreds of thousands of instructions can execute in the time taken for one interprocess communication.

If the code is written in a “safe” language such as Java or C#, there is another possibility: executing the code in a **sandbox** within the database query execution process itself. The sandbox allows the Java or C# code to access its own memory area, but it prevents the code from reading or updating the memory of the query execution process, or accessing files in the file system. (Creating a sandbox is not possible for a language such as C, which allows unrestricted access to memory through pointers.) Avoiding interprocess communication reduces function call overhead greatly.

Several database systems today support external language routines running in a sandbox within the query execution process. For example, Oracle and IBM DB2 allow Java functions to run as part of the database process. Microsoft SQL Server allows procedures compiled into the Common Language Runtime (CLR) to execute within the database process; such procedures could have been written, for example, in C# or Visual Basic. PostgreSQL allows functions defined in several languages, such as Perl, Python, and Tcl.

5.3 Triggers

A **trigger** is a statement that the system executes automatically as a side effect of a modification to the database. To define a trigger, we must:

- Specify when a trigger is to be executed. This is broken up into an *event* that causes the trigger to be checked and a *condition* that must be satisfied for trigger execution to proceed.
- Specify the *actions* to be taken when the trigger executes.

Once we enter a trigger into the database, the database system takes on the responsibility of executing it whenever the specified event occurs and the corresponding condition is satisfied.

5.3.1 Need for Triggers

Triggers can be used to implement certain integrity constraints that cannot be specified using the constraint mechanism of SQL. Triggers are also useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met. As an illustration, we could design a trigger that, whenever a tuple is inserted into the *takes* relation, updates the tuple in the *student* relation for the student taking the course by adding the number of credits for the course to the student's total credits. As another example, suppose a warehouse wishes to maintain a minimum inventory of each item; when the inventory level of an item falls below the minimum level, an order can be placed automatically. On an update of the inventory level of an item, the trigger compares the current inventory level with the minimum inventory level for the item, and if the level is at or below the minimum, a new order is created.

Note that triggers cannot usually perform updates outside the database, and hence, in the inventory replenishment example, we cannot use a trigger to place an order in the external world. Instead, we add an order to a relation holding reorders. We must create a separate permanently running system process that periodically scans that relation and places orders. Some database systems provide built-in support for sending email from SQL queries and triggers using this approach.

```

create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
    select time_slot_id
    from time_slot) /* time_slot_id not present in time_slot */
begin
    rollback
end;

create trigger timeslot_check2 after delete on timeslot
referencing old row as orow
for each row
when (orow.time_slot_id not in (
    select time_slot_id
    from time_slot) /* last tuple for time_slot_id deleted from time_slot */
and orow.time_slot_id in (
    select time_slot_id
    from section) /* and time_slot_id still referenced from section*/
begin
    rollback
end;

```

Figure 5.9 Using triggers to maintain referential integrity.

5.3.2 Triggers in SQL

We now consider how to implement triggers in SQL. The syntax we present here is defined by the SQL standard, but most databases implement nonstandard versions of this syntax. Although the syntax we present here may not be supported on such systems, the concepts we describe are applicable across implementations. We discuss nonstandard trigger implementations in Note 5.3 on page 212. In each system, trigger syntax is based upon that system's syntax for coding functions and procedures.

Figure 5.9 shows how triggers can be used to ensure referential integrity on the *time_slot_id* attribute of the *section* relation. The first trigger definition in the figure specifies that the trigger is initiated *after* any insert on the relation *section* and it ensures that the *time_slot_id* value being inserted is valid. SQL *bf insert* statement could insert multiple tuples of the relation, and the **for each row** clause in the trigger code would then explicitly iterate over each inserted row. The **referencing new row as** clause creates a variable *nrow* (called a **transition variable**) that stores the value of the row being inserted.

The **when** statement specifies a condition. The system executes the rest of the trigger body only for tuples that satisfy the condition. The **begin atomic ... end** clause can serve to collect multiple SQL statements into a single compound statement. In our example, though, there is only one statement, which rolls back the transaction that caused the trigger to get executed. Thus, any transaction that violates the referential integrity constraint gets rolled back, ensuring the data in the database satisfies the constraint.

It is not sufficient to check referential integrity on inserts alone; we also need to consider updates of *section*, as well as deletes and updates to the referenced table *time_slot*. The second trigger definition in Figure 5.9 considers the case of deletes to *time_slot*. This trigger checks that the *time_slot_id* of the tuple being deleted is either still present in *time_slot*, or that no tuple in *section* contains that particular *time_slot_id* value; otherwise, referential integrity would be violated.

To ensure referential integrity, we would also have to create triggers to handle updates to *section* and *time_slot*; we describe next how triggers can be executed on updates, but we leave the definition of these triggers as an exercise to the reader.

For updates, the trigger can specify attributes whose update causes the trigger to execute; updates to other attributes would not cause it to be executed. For example, to specify that a trigger executes after an update to the *grade* attribute of the *takes* relation, we write:

after update of takes on grade

The **referencing old row as** clause can be used to create a variable storing the old value of an updated or deleted row. The **referencing new row as** clause can be used with updates in addition to inserts.

Figure 5.10 shows how a trigger can be used to keep the *tot_cred* attribute value of *student* tuples up-to-date when the *grade* attribute is updated for a tuple in the *takes* relation. The trigger is executed only when the *grade* attribute is updated from a value that is either null or 'F' to a grade that indicates the course is successfully completed. The **update** statement is normal SQL syntax except for the use of the variable *nrow*.

A more realistic implementation of this example trigger would also handle grade corrections that change a successful completion grade to a failing grade and handle insertions into the *takes* relation where the *grade* indicates successful completion. We leave these as an exercise for the reader.

As another example of the use of a trigger, the action on **delete** of a *student* tuple could be to check if the student has any entries in the *takes* relation, and if so, to delete them.

Many database systems support a variety of other triggering events, such as when a user (application) logs on to the database (that is, opens a connection), the system shuts down, or changes are made to system settings.

Triggers can be activated **before** the event (**insert**, **delete**, or **update**) instead of **after** the event. Triggers that execute before an event can serve as extra constraints that can prevent invalid updates, inserts, or deletes. Instead of letting the invalid action proceed

```

create trigger credits_earned after update of takes on grade
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
    and (orow.grade = 'F' or orow.grade is null)
begin atomic
    update student
    set tot_cred = tot_cred +
        (select credits
         from course
         where course.course_id = nrow.course_id)
    where student.id = nrow.id;
end;

```

Figure 5.10 Using a trigger to maintain *credits_earned* values.

and cause an error, the trigger might take action to correct the problem so that the **update**, **insert**, or **delete** becomes valid. For example, if we attempt to insert an instructor into a department whose name does not appear in the *department* relation, the trigger could insert a tuple into the *department* relation for that department name before the insertion generates a foreign-key violation. As another example, suppose the value of an inserted grade is blank, presumably to indicate the absence of a grade. We can define a trigger that replaces the value with the **null** value. The **set** statement can be used to carry out such modifications. An example of such a trigger appears in Figure 5.11.

Instead of carrying out an action for each affected row, we can carry out a single action for the entire SQL statement that caused the insert, delete, or update. To do so, we use the **for each statement** clause instead of the **for each row** clause. The clauses

```

create trigger setnull before update of takes
referencing new row as nrow
for each row
when (nrow.grade = '')
begin atomic
    set nrow.grade = null;
end;

```

Figure 5.11 Example of using **set** to change an inserted value.

referencing old table as or referencing new table as can then be used to refer to temporary tables (called *transition tables*) containing all the affected rows. Transition tables cannot be used with **before** triggers, but they can be used with **after** triggers, regardless of whether they are statement triggers or row triggers. A single SQL statement can then be used to carry out multiple actions on the basis of the transition tables.

Triggers can be disabled or enabled; by default they are enabled when they are created, but they can be disabled by using **alter trigger trigger_name disable** (some databases use alternative syntax such as **disable trigger trigger_name**). A trigger that has been disabled can be enabled again. A trigger can instead be dropped, which removes it permanently, by using the command **drop trigger trigger_name**.

Returning to our inventory-replenishment example from Section 5.3.1, suppose we have the following relations:

- *inventory (item, level)*, which notes the current amount of the item in the warehouse.
- *minlevel (item, level)*, which notes the minimum amount of the item to be maintained.
- *reorder (item, amount)*, which notes the amount of the item to be ordered when its level falls below the minimum.
- *orders (item, amount)*, which notes the amount of the item to be ordered.

To place a reorder when inventory falls below a specified minimum, we can use the trigger shown in Figure 5.12. Note that we have been careful to place an order only when the amount falls from above the minimum level to below the minimum level. If we check only that the new value after an update is below the minimum level, we may place an order erroneously when the item has already been reordered.

SQL-based database systems use triggers widely, although before SQL:1999 they were not part of the SQL standard. Unfortunately, as a result, each database system implemented its own syntax for triggers, leading to incompatibilities. The SQL:1999 syntax for triggers that we use here is similar, but not identical, to the syntax in the IBM DB2 and Oracle database systems. See Note 5.3 on page 212.

5.3.3 When Not to Use Triggers

There are many good uses for triggers, such as those we have just seen in Section 5.3.2, but some uses are best handled by alternative techniques. For example, we could implement the **on delete cascade** feature of a foreign-key constraint by using a trigger instead of using the cascade feature. Not only would this be more work to implement, but also it would be much harder for a database user to understand the set of constraints implemented in the database.

```

create trigger reorder after update of level on inventory
referencing old row as orow, new row as nrow
for each row
when nrow.level <= (select level
        from minlevel
        where minlevel.item = orow.item)
and orow.level > (select level
        from minlevel
        where minlevel.item = orow.item)
begin atomic
    insert into orders
        (select item, amount
        from reorder
        where reorder.item = orow.item);
end;

```

Figure 5.12 Example of trigger for reordering an item.

As another example, triggers can be used to maintain materialized views. For instance, if we wished to support very fast access to the total number of students registered for each course section, we could do this by creating a relation

section_registration(course_id, sec_id, semester, year, total_students)

defined by the query

```

select course_id, sec_id, semester, year, count(ID) as total_students
from takes
group by course_id, sec_id, semester, year;

```

The value of *total_students* for each course must be maintained up-to-date by triggers on insert, delete, or update of the *takes* relation. Such maintenance may require insertion, update or deletion of tuples from *section_registration*, and triggers must be written accordingly.

However, many database systems now support materialized views, which are automatically maintained by the database system (see Section 4.2.3). As a result, there is no need to write trigger code for maintaining such materialized views.

Triggers have been used for maintaining copies, or replicas, of databases. A collection of triggers on insert, delete, or update can be created on each relation to record the changes in relations called **change** or **delta** relations. A separate process copies over the changes to the replica of the database. Modern database systems, however, provide

Note 5.3 NONSTANDARD TRIGGER SYNTAX

Although the trigger syntax we describe here is part of the SQL standard, and is supported by IBM DB2, most other database systems have nonstandard syntax for specifying triggers and may not implement all features in the SQL standard. We outline a few of the differences below; see the respective system manuals for further details.

For example, in the Oracle syntax, unlike the SQL standard syntax, the keyword **row** does not appear in the **referencing** statement. The keyword **atomic** does not appear after **begin**. The reference to *nrow* in the **select** statement nested in the **update** statement must begin with a colon (:) to inform the system that the variable *nrow* is defined externally from the SQL statement. Further, subqueries are not allowed in the **when** and **if** clauses. It is possible to work around this problem by moving complex predicates from the **when** clause into a separate query that saves the result into a local variable, and then reference that variable in an **if** clause, and the body of the trigger then moves into the corresponding **then** clause. Further, in Oracle, triggers are not allowed to execute a transaction rollback directly; however, they can instead use a function called `raise_application_error` to not only roll back the transaction but also return an error message to the user/application that performed the update.

As another example, in Microsoft SQL Server the keyword **on** is used instead of **after**. The **referencing** clause is omitted, and old and new rows are referenced by the tuple variables **deleted** and **inserted**. Further, the **for each row** clause is omitted, and **when** is replaced by **if**. The **before** specification is not supported, but an **instead of** specification is supported.

In PostgreSQL, triggers do not have a body, but instead invoke a procedure for each row, which can access variables **new** and **old** containing the old and new values of the row. Instead of performing a rollback, the trigger can raise an exception with an associated error message.

built-in facilities for database replication, making triggers unnecessary for replication in most cases. Replicated databases are discussed in detail in Chapter 23.

Another problem with triggers lies in unintended execution of the triggered action when data are loaded from a backup copy,⁵ or when database updates at a site are replicated on a backup site. In such cases, the triggered action has already been executed, and typically it should not be executed again. When loading data, triggers can be disabled explicitly. For backup replica systems that may have to take over from the primary system, triggers would have to be disabled initially and enabled when the backup

⁵We discuss database backup and recovery from failures in detail in Chapter 19.

<i>course_id</i>	<i>prereq_id</i>
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-190
CS-319	CS-101
CS-319	CS-315
CS-347	CS-319

Figure 5.13 An instance of the *prereq* relation.

site takes over processing from the primary system. As an alternative, some database systems allow triggers to be specified as **not for replication**, which ensures that they are not executed on the backup site during database replication. Other database systems provide a system variable that denotes that the database is a replica on which database actions are being replayed; the trigger body should check this variable and exit if it is true. Both solutions remove the need for explicit disabling and enabling of triggers.

Triggers should be written with great care, since a trigger error detected at runtime causes the failure of the action statement that set off the trigger. Furthermore, the action of one trigger can set off another trigger. In the worst case, this could even lead to an infinite chain of triggering. For example, suppose an insert trigger on a relation has an action that causes another (new) insert on the same relation. The insert action then triggers yet another insert action, and so on ad infinitum. Some database systems limit the length of such chains of triggers (for example, to 16 or 32) and consider longer chains of triggering an error. Other systems flag as an error any trigger that attempts to reference the relation whose modification caused the trigger to execute in the first place.

Triggers can serve a very useful purpose, but they are best avoided when alternatives exist. Many trigger applications can be substituted by appropriate use of stored procedures, which we discussed in Section 5.2.

5.4 Recursive Queries

Consider the instance of the relation *prereq* shown in Figure 5.13 containing information about the various courses offered at the university and the prerequisite for each course.⁶

Suppose now that we want to find out which courses are a prerequisite whether directly or indirectly, for a specific course—say, CS-347. That is, we wish to find a course

⁶This instance of *prereq* differs from that used earlier for reasons that will become apparent as we use it to explain recursive queries.

that is a direct prerequisite for CS-347, or is a prerequisite for a course that is a prerequisite for CS-347, and so on.

Thus, since CS-319 is a prerequisite for CS-347 and CS-315 and CS-101 are prerequisites for CS-319, CS-315 and CS-101 are also prerequisites (indirectly) for CS-347. Then, since CS-190 is a prerequisite for CS-315, CS-190 is another indirect prerequisite for CS-347. Continuing, we see that CS-101 is a prerequisite for CS-190, but note that CS-101 was already added to the list of prerequisites for CS-347. In a real university, rather than our example, we would not expect such a complex prerequisite structure, but this example serves to show some of the situations that might possibly arise.

The **transitive closure** of the relation *prereq* is a relation that contains all pairs (*cid*, *pre*) such that *pre* is a direct or indirect prerequisite of *cid*. There are numerous applications that require computation of similar transitive closures on **hierarchies**. For instance, organizations typically consist of several levels of organizational units. Machines consist of parts that in turn have subparts, and so on; for example, a bicycle may have subparts such as wheels and pedals, which in turn have subparts such as tires, rims, and spokes. Transitive closure can be used on such hierarchies to find, for example, all parts in a bicycle.

5.4.1 Transitive Closure Using Iteration

One way to write the preceding query is to use iteration: First find those courses that are a direct prerequisite of CS-347, then those courses that are a prerequisite of all the courses under the first set, and so on. This iterative process continues until we reach an iteration where no courses are added. Figure 5.14 shows a function *findAllPrereqs*(*cid*) to carry out this task; the function takes the *course_id* of the course as a parameter (*cid*), computes the set of all direct and indirect prerequisites of that course, and returns the set.

The procedure uses three temporary tables:

- *c_prereq*: stores the set of tuples to be returned.
- *new_c_prereq*: stores the courses found in the previous iteration.
- *temp*: used as temporary storage while sets of courses are manipulated.

Note that SQL allows the creation of temporary tables using the command **create temporary table**; such tables are available only within the transaction executing the query and are dropped when the transaction finishes. Moreover, if two instances of *findAllPrereqs* run concurrently, each gets its own copy of the temporary tables; if they shared a copy, their result could be incorrect.

The procedure inserts all direct prerequisites of course *cid* into *new_c_prereq* before the **repeat** loop. The **repeat** loop first adds all courses in *new_c_prereq* to *c_prereq*. Next, it computes prerequisites of all those courses in *new_c_prereq*, except those that have already been found to be prerequisites of *cid*, and stores them in the temporary table

```

create function findAllPrereqs(cid varchar(8))
  -- Finds all courses that are prerequisite (directly or indirectly) for cid
returns table (course_id varchar(8))
  -- The relation prereq(course_id, prereq_id) specifies which course is
  -- directly a prerequisite for another course.
begin
  create temporary table c_prereq (course_id varchar(8));
  -- table c_prereq stores the set of courses to be returned
  create temporary table new_c_prereq (course_id varchar(8));
  -- table new_c_prereq contains courses found in the previous iteration
  create temporary table temp (course_id varchar(8));
  -- table temp is used to store intermediate results
  insert into new_c_prereq
    select prereq_id
    from prereq
    where course_id = cid;
  repeat
    insert into c_prereq
      select course_id
      from new_c_prereq;

    insert into temp
      (select prereq.prereq_id
        from new_c_prereq, prereq
        where new_c_prereq.course_id = prereq.course_id
      )
    except (
      select course_id
      from c_prereq
    );
    delete from new_c_prereq;
    insert into new_c_prereq
      select *
      from temp;
    delete from temp;
    until not exists (select * from new_c_prereq)
    end repeat;
    return table c_prereq;
end

```

Figure 5.14 Finding all prerequisites of a course.

Iteration Number	Tuples in c1
0	
1	(CS-319)
2	(CS-319), (CS-315), (CS-101)
3	(CS-319), (CS-315), (CS-101), (CS-190)
4	(CS-319), (CS-315), (CS-101), (CS-190)
5	done

Figure 5.15 Prerequisites of CS-347 in iterations of function *findAllPrereqs*.

temp. Finally, it replaces the contents of *new_c_prereq* with the contents of *temp*. The **repeat** loop terminates when it finds no new (indirect) prerequisites.

Figure 5.15 shows the prerequisites that are found in each iteration when the procedure is called for CS-347. While *c_prereq* could have been updated in one SQL statement, we need first to construct *new_c_prereq* so we can tell when nothing is being added in the (final) iteration.

The use of the **except** clause in the function ensures that the function works even in the (abnormal) case where there is a cycle of prerequisites. For example, if *a* is a prerequisite for *b*, *b* is a prerequisite for *c*, and *c* is a prerequisite for *a*, there is a cycle.

While cycles may be unrealistic in course prerequisites, cycles are possible in other applications. For instance, suppose we have a relation *flights(to, from)* that says which cities can be reached from which other cities by a direct flight. We can write code similar to that in the *findAllPrereqs* function, to find all cities that are reachable by a sequence of one or more flights from a given city. All we have to do is to replace *prereq* with *flight* and replace attribute names correspondingly. In this situation, there can be cycles of reachability, but the function would work correctly since it would eliminate cities that have already been seen.

5.4.2 Recursion in SQL

It is rather inconvenient to specify transitive closure using iteration. There is an alternative approach, using recursive view definitions, that is easier to use.

We can use recursion to define the set of courses that are prerequisites of a particular course, say CS-347, as follows. The courses that are prerequisites (directly or indirectly) of CS-347 are:

- Courses that are prerequisites for CS-347.
- Courses that are prerequisites for those courses that are prerequisites (directly or indirectly) for CS-347.

Note that case 2 is recursive, since it defines the set of courses that are prerequisites of CS-347 in terms of the set of courses that are prerequisites of CS-347. Other examples

```

with recursive rec_prereq(course_id, prereq_id) as (
    select course_id, prereq_id
    from prereq
    union
    select rec_prereq.course_id, prereq.prereq_id
    from rec_prereq, prereq
    where rec_prereq.prereq_id = prereq.course_id
)
select *
from rec_prereq;

```

Figure 5.16 Recursive query in SQL.

of transitive closure, such as finding all subparts (direct or indirect) of a given part can also be defined in a similar manner, recursively.

The SQL standard supports a limited form of recursion, using the **with recursive** clause, where a view (or temporary view) is expressed in terms of itself. Recursive queries can be used, for example, to express transitive closure concisely. Recall that the **with** clause is used to define a temporary view whose definition is available only to the query in which it is defined. The additional keyword **recursive** specifies that the view is recursive.⁷

For example, we can find every pair (*cid,pre*) such that *pre* is directly or indirectly a prerequisite for course *cid*, using the recursive SQL view shown in Figure 5.16.

Any recursive view must be defined as the union⁸ of two subqueries: a **base query** that is nonrecursive and a **recursive query** that uses the recursive view. In the example in Figure 5.16, the base query is the select on *prereq* while the recursive query computes the join of *prereq* and *rec_prereq*.

The meaning of a recursive view is best understood as follows: First compute the base query and add all the resultant tuples to the recursively defined view relation *rec_prereq* (which is initially empty). Next compute the recursive query using the current contents of the view relation, and add all the resulting tuples back to the view relation. Keep repeating the above step until no new tuples are added to the view relation. The resultant view relation instance is called a **fixed point** of the recursive view definition. (The term “fixed” refers to the fact that there is no further change.) The view relation is thus defined to contain exactly the tuples in the fixed-point instance.

Applying this logic to our example, we first find all direct prerequisites of each course by executing the base query. The recursive query adds one more level of courses

⁷Some systems treat the **recursive** keyword as optional; others disallow it.

⁸Some systems, notably Oracle, require use of **union all**.

in each iteration, until the maximum depth of the course-prereq relationship is reached. At this point no new tuples are added to the view, and a fixed point is reached.

To find the prerequisites of a specific course, such as CS-347, we can modify the outer level query by adding a **where** clause “**where** *rec_prereq.course_id* = ‘CS-347’”. One way to evaluate the query with the selection is to compute the full contents of *rec_prereq* using the iterative technique, and then select from this result only those tuples whose *course_id* is CS-347. However, this would result in computing (course, prerequisite) pairs for all courses, all of which are irrelevant except for those for the course CS-347. In fact the database system is not required to use this iterative technique to compute the full result of the recursive query and then perform the selection. It may get the same result using other techniques that may be more efficient, such as that used in the function *findAllPrereqs* which we saw earlier. See the bibliographic notes for references to more information on this topic.

There are some restrictions on the recursive query in a recursive view; specifically, the query must be **monotonic**, that is, its result on a view relation instance V_1 must be a superset of its result on a view relation instance V_2 if V_1 is a superset of V_2 . Intuitively, if more tuples are added to the view relation, the recursive query must return at least the same set of tuples as before, and possibly return additional tuples.

In particular, recursive queries may not use any of the following constructs, since they would make the query nonmonotonic:

- Aggregation on the recursive view.
- **not exists** on a subquery that uses the recursive view.
- Set difference (**except**) whose right-hand side uses the recursive view.

For instance, if the recursive query was of the form $r - v$, where v is the recursive view, if we add a tuple to v , the result of the query can become smaller; the query is therefore not monotonic.

The meaning of recursive views can be defined by the iterative procedure as long as the recursive query is monotonic; if the recursive query is nonmonotonic, the meaning of the view is hard to define. SQL therefore requires the queries to be monotonic. Recursive queries are discussed in more detail in the context of the Datalog query language, in Section 27.4.6.

SQL also allows creation of recursively defined permanent views by using **create recursive view** in place of **with recursive**. Some implementations support recursive queries using a different syntax. This includes the Oracle **start with / connect by prior** syntax for what it calls hierarchical queries.⁹ See the respective system manuals for further details.

⁹Starting with Oracle 12.c, the standard syntax is accepted in addition to the legacy hierarchical syntax, with the **recursive** keyword omitted and with the requirement in our example that **union all** be used instead of **union**.

5.5 Advanced Aggregation Features

The aggregation support in SQL is quite powerful and handles most common tasks with ease. However, there are some tasks that are hard to implement efficiently with the basic aggregation features. In this section, we study features in SQL to handle some such tasks.

5.5.1 Ranking

Finding the position of a value within a set is a common operation. For instance, we may wish to assign students a rank in class based on their grade-point average (GPA), with the rank 1 going to the student with the highest GPA, the rank 2 to the student with the next highest GPA, and so on. A related type of query is to find the percentile in which a value in a (multi)set belongs, for example, the bottom third, middle third, or top third. While such queries can be expressed using the SQL constructs we have seen so far, they are difficult to express and inefficient to evaluate. Programmers may resort to writing the query partly in SQL and partly in a programming language. We study SQL support for direct expression of these types of queries here.

In our university example, the *takes* relation shows the grade each student earned in each course taken. To illustrate ranking, let us assume we have a view *student_grades* (*ID*, *GPA*) giving the grade-point average of each student.¹⁰

Ranking is done with an **order by** specification. The following query gives the rank of each student:

```
select ID, rank() over (order by (GPA) desc) as s_rank
      from student_grades;
```

Note that the order of tuples in the output is not defined, so they may not be sorted by rank. An extra **order by** clause is needed to get them in sorted order, as follows:

```
select ID, rank () over (order by (GPA) desc) as s_rank
      from student_grades
      order by s_rank;
```

A basic issue with ranking is how to deal with the case of multiple tuples that are the same on the ordering attribute(s). In our example, this means deciding what to do if there are two students with the same GPA. The **rank** function gives the same rank to all tuples that are equal on the **order by** attributes. For instance, if the highest GPA is shared by two students, both would get rank 1. The next rank given would be 3, not 2, so if three students get the next highest GPA, they would all get rank 3, and the next

¹⁰The SQL statement to create the view *student_grades* is somewhat complex since we must convert the letter grades in the *takes* relation to numbers and weight the grades for each course by the number of credits for that course. The definition of this view is the goal of Exercise 4.6.

student(s) would get rank 6, and so on. There is also a **dense_rank** function that does not create gaps in the ordering. In the preceding example, the tuples with the second highest value all get rank 2, and tuples with the third highest value get rank 3, and so on.

If there are null values among the values being ranked, they are treated as the highest values. That makes sense in some situations, although for our example, it would result in students with no courses being shown as having the highest GPAs. Thus, we see that care needs to be taken in writing ranking queries in cases where null values may appear. SQL permits the user to specify where they should occur by using **nulls first** or **nulls last**, for instance:

```
select ID, rank () over (order by GPA desc nulls last) as s_rank  
from student_grades;
```

It is possible to express the preceding query with the basic SQL aggregation functions, using the following query:

```
select ID, (1 + (select count(*)  
         from student_grades B  
         where B.GPA > A.GPA)) as s_rank  
from student_grades A  
order by s_rank;
```

It should be clear that the rank of a student is merely 1 plus the number of students with a higher *GPA*, which is exactly what the query specifies.¹¹ However, this computation of each student's rank takes time linear in the size of the relation, leading to an overall time quadratic in the size of the relation. On large relations, the above query could take a very long time to execute. In contrast, the system's implementation of the **rank** clause can sort the relation and compute the rank in much less time.

Ranking can be done within partitions of the data. For instance, suppose we wish to rank students by department rather than across the entire university. Assume that a view is defined like *student_grades* but including the department name: *dept_grades*(*ID*, *dept_name*, *GPA*). The following query then gives the rank of students within each section:

```
select ID, dept_name,  
     rank () over (partition by dept_name order by GPA desc) as dept_rank  
from dept_grades  
order by dept_name, dept_rank;
```

¹¹There is a slight technical difference if a student has not taken any courses and therefore has a *null* GPA. Due to how comparisons of null values work in SQL, a student with a null GPA does not contribute to other students' **count** values.

The outer **order by** clause orders the result tuples by department name, and within each department by the rank.

Multiple **rank** expressions can be used within a single **select** statement; thus, we can obtain the overall rank and the rank within the department by using two **rank** expressions in the same **select** clause. When ranking (possibly with partitioning) occurs along with a **group by** clause, the **group by** clause is applied first, and partitioning and ranking are done on the results of the **group by**. Thus, aggregate values can then be used for ranking.

It is often the case, especially for large results, that we may be interested only in the top-ranking tuples of the result rather than the entire list. For rank queries, this can be done by nesting the ranking query within a containing query whose **where** clause chooses only those tuples whose rank is lower than some specified value. For example, to find the top 5 ranking students based on GPA we could extend our earlier example by writing:

```
select *
  from (select ID, rank() over (order by (GPA) desc) as s_rank
        from student_grades)
       where s_rank <= 5;
```

This query does not necessarily give 5 students, since there could be ties. For example, if 2 students tie for fifth, the result would contain a total of 6 tuples. Note that the bottom n is simply the same as the top n with a reverse sorting order.

Several database systems provide nonstandard SQL syntax to specify directly that only the top n results are required. In our example, this would allow us to find the top 5 students without the need to use the **rank** function. However, those constructs result in exactly the number of tuples specified (5 in our example), and so ties for the final position are broken arbitrarily. The exact syntax for these “top n ” queries varies widely among systems; see Note 5.4 on page 222. Note that the top n constructs do not support partitioning; so we cannot get the top n within each partition without performing ranking.

Several other functions can be used in place of **rank**. For instance, **percent_rank** of a tuple gives the rank of the tuple as a fraction. If there are n tuples in the partition¹² and the rank of the tuple is r , then its percent rank is defined as $(r - 1)/(n - 1)$ (and as **null** if there is only one tuple in the partition). The function **cume_dist**, short for cumulative distribution, for a tuple is defined as p/n where p is the number of tuples in the partition with ordering values preceding or equal to the ordering value of the tuple and n is the number of tuples in the partition. The function **row_number** sorts the rows and gives each row a unique number corresponding to its position in the sort order; different rows with the same ordering value would get different row numbers, in a nondeterministic fashion.

¹²The entire set is treated as a single partition if no explicit partition is used.

Note 5.4 TOP-N QUERIES

Often, only the first few tuples of a query result are required. This may occur in a ranking query where only top-ranked results are of interest. Another case where this may occur is in a query with an **order by** from which only the top values are of interest. Restricting results to the top-ranked results can be done using the **rank** function as we saw earlier, but that syntax is rather cumbersome. Many databases support a simpler syntax for such restriction, but the syntax varies widely among the leading database systems. We provide a few examples here.

Some systems (including MySQL and PostgreSQL) allow a clause **limit *n*** to be added at the end of an SQL query to specify that only the first *n* tuples should be output. This clause can be used in conjunction with an **order by** clause to fetch the top *n* tuples, as illustrated by the following query, which retrieves the ID and GPA of the top 10 students in order of GPA:

```
select ID, GPA
from student_grades
order by GPA desc
limit 10;
```

In IBM DB2 and the most recent versions of Oracle, the equivalent of the **limit** clause is **fetch first 10 rows only**. Microsoft SQL Server places its version of this feature in the **select** clause rather than adding a separate **limit** clause. The **select** clause is written as: **select top 10 *ID, GPA***.

Oracle (both current and older versions) offers the concept of a *row number* to provide this feature. A special, hidden attribute *rownum* numbers tuples of a result relation in order of retrieval. This attribute can then be used in a **where** clause within a containing query. However, the use of this feature is a bit tricky, since the *rownum* is decided before rows are sorted by an **order by** clause. To use it properly, a nested query should be used as follows:

```
select *
from (select ID, GPA
        from student_grades
        order by GPA desc)
where rownum <= 10;
```

The nested query ensures that the predicate on *rownum* is applied only after the **order by** is applied.

Some database systems have features allowing tuple limits to be exceeded in case of ties. See your system's documentation for details.

Finally, for a given constant n , the ranking function $\text{ntile}(n)$ takes the tuples in each partition in the specified order and divides them into n buckets with equal numbers of tuples.¹³ For each tuple, $\text{ntile}(n)$ then gives the number of the bucket in which it is placed, with bucket numbers starting with 1. This function is particularly useful for constructing histograms based on percentiles. We can show the quartile into which each student falls based on GPA by the following query:

```
select ID, ntile(4) over (order by (GPA desc)) as quartile
from student_grades;
```

5.5.2 Windowing

Window queries compute an aggregate function over ranges of tuples. This is useful, for example, to compute an aggregate of a fixed range of time; the time range is called a *window*. Windows may overlap, in which case a tuple may contribute to more than one window. This is unlike the partitions we saw earlier, where a tuple could contribute to only one partition.

An example of the use of windowing is trend analysis. Consider our earlier sales example. Sales may fluctuate widely from day to day based on factors like weather (e.g., a snowstorm, flood, hurricane, or earthquake might reduce sales for a period of time). However, over a sufficiently long period of time, fluctuations might be less (continuing the example, sales may “make up” for weather-related downturns). Stock-market trend analysis is another example of the use of the windowing concept. Various “moving averages” are found on business and investment web sites.

It is relatively easy to write an SQL query using those features we have already studied to compute an aggregate over one window, for example, sales over a fixed 3-day period. However, if we want to do this for *every* 3-day period, the query becomes cumbersome.

SQL provides a windowing feature to support such queries. Suppose we are given a view *tot_credits* (*year*, *num_credits*) giving the total number of credits taken by students in each year.¹⁴ Note that this relation can contain at most one tuple for each year. Consider the following query:

```
select year, avg(num_credits)
over (order by year rows 3 preceding)
as avg_total_credits
from tot_credits;
```

¹³If the total number of tuples in a partition is not divisible by n , then the number of tuples in each bucket can differ by at most 1. Tuples with the same value for the ordering attribute may be assigned to different buckets, nondeterministically, in order to make the number of tuples in each bucket equal.

¹⁴We leave the definition of this view in terms of our university example as an exercise.

This query computes averages over the three *preceding* tuples in the specified sort order. Thus, for 2019, if tuples for years 2018 and 2017 are present in the relation *tot_credits*, since each year is represented by only one tuple, the result of the window definition is the average of the values for years 2017, 2018, and 2019. The averages each year would be computed in a similar manner. For the earliest year in the relation *tot_credits*, the average would be over only that year itself, while for the next year, the average would be over 2 years. Note that this example makes sense only because each year appears only once in *tot_weight*. Were this not the case, then there would be several possible orderings of tuples for the same year could be in any order. We shall see shortly a windowing query that uses a range of values instead of a specific number of tuples.

Suppose that instead of going back a fixed number of tuples, we want the window to consist of all prior years. That means the number of prior years considered is not fixed. To get the average total credits over all prior years, we write:

```
select year, avg(num_credits)
    over (order by year rows unbounded preceding)
        as avg_total_credits
from tot_credits;
```

It is possible to use the keyword **following** in place of **preceding**. If we did this in our example, the *year* value specifies the beginning of the window instead of the end. Similarly, we can specify a window beginning before the current tuple and ending after it:

```
select year, avg(num_credits)
    over (order by year rows between 3 preceding and 2 following)
        as avg_total_credits
from tot_credits;
```

In our example, all tuples pertain to the entire university. Suppose instead we have credit data for each department in a view *tot_credits_dept* (*dept_name*, *year*, *num_credits*) giving the total number of credits students took with the particular department in the specified year. (Again, we leave writing this view definition as an exercise.) We can write windowing queries that treat each department separately by partitioning by *dept_name*:

```
select dept_name, year, avg(num_credits)
    over (partition by dept_name
        order by year rows between 3 preceding and current row)
        as avg_total_credits
from tot_credits_dept;
```

<i>item_name</i>	<i>color</i>	<i>clothes_size</i>	<i>quantity</i>
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
pants	dark	small	14
pants	dark	medium	6
pants	dark	large	0
pants	pastel	small	1
pants	pastel	medium	0
pants	pastel	large	1
pants	white	small	3
pants	white	medium	0
pants	white	large	2
shirt	dark	small	2
shirt	dark	medium	6
shirt	dark	large	6
shirt	pastel	small	4
shirt	pastel	medium	1
shirt	pastel	large	2
shirt	white	small	17
shirt	white	medium	1
shirt	white	large	10
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3

Figure 5.17 An example of *sales* relation.

<i>item_name</i>	<i>clothes_size</i>	<i>dark</i>	<i>pastel</i>	<i>white</i>
dress	small	2	4	2
dress	medium	6	3	3
dress	large	12	3	0
pants	small	14	1	3
pants	medium	6	0	0
pants	large	0	1	2
shirt	small	2	4	17
shirt	medium	6	1	1
shirt	large	6	2	10
skirt	small	2	11	2
skirt	medium	5	9	5
skirt	large	1	15	3

Figure 5.18 Result of SQL pivot operation on the *sales* relation of Figure 5.17.

The use of the keyword **range** in place of **row** allows the windowing query to cover all tuples with a particular value rather than covering a specific number of tuples. Thus for example, **rows current row** refers to exactly one tuple, while **range current row** refers to all tuples whose value for the *sort* attribute is the same as that of the current tuple. The **range** keyword is not implemented fully in every system.¹⁵

5.5.3 Pivoting

Consider an application where a shop wants to find out what kinds of clothes are popular. Let us suppose that clothes are characterized by their *item_name*, *color*, and *size*, and that we have a relation *sales* with the schema.

sales (item_name, color, clothes_size, quantity)

Suppose that *item_name* can take on the values (skirt, dress, shirt, pants), *color* can take on the values (dark, pastel, white), *clothes_size* can take on values (small, medium, large), and *quantity* is an integer value representing the total number of items sold of a given (*item_name*, *color*, *clothes_size*) combination. An instance of the *sales* relation is shown in Figure 5.17.

Figure 5.18 shows an alternative way to view the data that is present in Figure 5.17; the values “dark”, “pastel”, and “white” of attribute *color* have become attribute names in Figure 5.18. The table in Figure 5.18 is an example of a **cross-tabulation** (or **cross-tab**, for short), also referred to as a **pivot-table**.

The values of the new attributes *dark*, *pastel* and *white* in our example are defined as follows. For a particular combination of *item_name*, *clothes_size* (e.g., (“dress”, “dark”))

¹⁵Some systems, such as PostgreSQL, allow **range** only with **unbounded**.

if there is a single tuple with *color* value “dark”, the *quantity* value of that attribute appears as the value for the attribute *dark*. If there are multiple such tuples, the values are aggregated using the **sum** aggregate in our example; in general other aggregate functions could be used instead. Values for the other two attributes, *pastel* and *white*, are similarly defined.

In general, a cross-tab is a table derived from a relation (say, *R*), where values for some attribute of relation *R* (say, *A*) become attribute names in the result; the attribute *A* is the **pivot** attribute. Cross-tabs are widely used for data analysis, and are discussed in more detail in Section 11.3.

Several SQL implementations, such as Microsoft SQL Server, and Oracle, support a **pivot** clause that allows creation of cross-tabs. Given the *sales* relation from Figure 5.17, the query:

```
select *
from sales
pivot (
    sum(quantity)
    for color in ('dark', 'pastel', 'white')
)
```

returns the result shown in Figure 5.18.

Note that the **for** clause within the **pivot** clause specifies (i) a pivot attribute (*color*, in the above query), (ii) the values of that attribute that should appear as attribute names in the pivot result (dark, pastel and white, in the above query), and (iii) the aggregate function that should be used to compute the value of the new attributes (aggregate function **sum**, on the attribute *quantity*, in the above query).

The attribute *color* and *quantity* do not appear in the result, but all other attributes are retained. In case more than one tuple contributes values to a given cell, the aggregate operation within the **pivot** clause specifies how the values should be combined. In the above example, the *quantity* values are aggregated using the **sum** function.

A query using **pivot** can be written using basic SQL constructs, without using the pivot construct, but the construct simplifies the task of writing such queries.

5.5.4 Rollup and Cube

SQL supports generalizations of the **group by** construct using the **rollup** and **cube** operations, which allow multiple **group by** queries to be run in a single query, with the result returned as a single relation.

Consider again our retail shop example and the relation:

$$\text{sales} (\text{item_name}, \text{color}, \text{clothes_size}, \text{quantity})$$

We can find the number of items sold in each item name by writing a simple **group by** query:

```
select item_name, sum(quantity) as quantity
from sales
group by item_name;
```

Similarly, we can find the number of items sold in each color, and each size. We can further find a breakdown of sales by item-name and color by writing:

```
select item_name, color, sum(quantity) as quantity
from sales
group by item_name, color;
```

Similarly, a query with **group by item_name, color, clothes_size** would allow us to see the sales breakdown by (*item_name, color, clothes_size*) combinations.

Data analysts often need to view data aggregated in multiple ways as illustrated above. The SQL **rollup** and **cube** constructs provide a concise way to get multiple such aggregates using a single query, instead of writing multiple queries.

The **rollup** construct is illustrated using the following query:

```
select item_name, color, sum(quantity)
from sales
group by rollup(item_name, color);
```

The result of the query is shown in Figure 5.19. The above query is equivalent to the following query using the **union** operation.

```
(select item_name, color, sum(quantity) as quantity
from sales
group by item_name, color)
union
(select item_name, null as color, sum(quantity) as quantity
from sales
group by item_name)
union
(select null as item_name, null as color, sum(quantity) as quantity
from sales)
```

The construct **group by rollup(item_name, color)** generates 3 groupings:

```
{ (item_name, color), (item_name), () }
```

where () denotes an empty **group by** list. Observe that a grouping is present for each prefix of the attributes listed in the **rollup** clause, including the empty prefix. The query result contains the union of the results by these groupings. The different groupings generate different schemas; to bring the results of the different groupings to a common

<i>item_name</i>	<i>color</i>	<i>quantity</i>
skirt	dark	8
skirt	pastel	35
skirt	white	10
dress	dark	20
dress	pastel	10
dress	white	5
shirt	dark	14
shirt	pastel	7
shirt	white	28
pants	dark	20
pants	pastel	2
pants	white	5
skirt	<i>null</i>	53
dress	<i>null</i>	35
shirt	<i>null</i>	49
pants	<i>null</i>	27
<i>null</i>	<i>null</i>	164

Figure 5.19 Query result: `group by rollup (item_name, color)`.

schema, tuples in the result contain *null* as the value of those attributes not present in a particular grouping.¹⁶

The **cube** construct generates an even larger number of groupings, consisting of *all subsets* of the attributes listed in the **cube** construct. For example, the query:

```
select item_name, color, clothes_size, sum(quantity)
  from sales
 group by cube(item_name, color, clothes_size);
```

generates the following groupings:

```
{ (item_name, color, clothes_size), (item_name, color), (item_name, clothes_size),
  (color, clothes_size), (item_name), (color), (clothes_size), () }
```

To bring the results of the different groupings to a common schema, as with **rollup**, tuples in the result contain *null* as the value of those attributes not present in a particular grouping.

¹⁶The SQL **outer union** operation can be used to perform a union of relations that may not have a common schema. The resultant schema has the union of all the attributes across the inputs; each input tuple is mapped to an output tuple by adding all the attributes missing in that tuple, with the value set to null. Our union query can be written using outer union, and in that case we do not need to explicitly generate null-value attributes using *null* as attribute-name constructs, as we have done in the above query.

Multiple **rollups** and **cubes** can be used in a single **group by** clause. For instance, the following query:

```
select item_name, color, clothes_size, sum(quantity)
  from sales
 group by rollup(item_name), rollup(color, clothes_size);
```

generates the groupings:

```
{ (item_name, color, clothes_size), (item_name, color), (item_name),
  (color, clothes_size), (color), () }
```

To understand why, observe that **rollup(item_name)** generates a set of two groupings, $\{(item_name), ()\}$, while **rollup(color, clothes_size)** generates a set of three groupings, $\{(color, clothes_size), (color), ()\}$. The Cartesian product of the two sets gives us the six groupings shown.

Neither the **rollup** nor the **cube** clause gives complete control on the groupings that are generated. For instance, we cannot use them to specify that we want only groupings $\{(color, clothes_size), (clothes_size, item_name)\}$. Such restricted groupings can be generated by using the **grouping sets** construct, in which one can specify the specific list of groupings to be used. To obtain only groupings $\{(color, clothes_size), (clothes_size, item_name)\}$, we would write:

```
select item_name, color, clothes_size, sum(quantity)
  from sales
 group by grouping sets ((color, clothes_size), (clothes_size, item_name));
```

Analysts may want to distinguish those nulls generated by **rollup** and **cube** operations from “normal” nulls actually stored in the database or arising from an outer join. The **grouping()** function returns 1 if its argument is a null value generated by a **rollup** or **cube** and 0 otherwise (note that the **grouping** function is different from the **grouping sets** construct). If we wish to display the **rollup** query result shown in Figure 5.19, but using the value “all” in place of nulls generated by **rollup**, we can use the query:

```
select (case when grouping(item_name) = 1 then 'all'
            else item_name end) as item_name,
       (case when grouping(color) = 1 then 'all'
            else color end) as color,
       sum(quantity) as quantity
  from sales
 group by rollup(item_name, color);
```

One might consider using the following query using **coalesce**, but it would incorrectly convert null item names and colors to **all**:

```

select coalesce (item_name,'all') as item_name,
       coalesce (color,'all') as color,
       sum(quantity) as quantity
  from sales
 group by rollup(item_name, color);

```

5.6 Summary

- SQL queries can be invoked from host languages via embedded and dynamic SQL. The ODBC and JDBC standards define application program interfaces to access SQL databases from C and Java language programs.
- Functions and procedures can be defined using SQL procedural extensions that allow iteration and conditional (if-then-else) statements.
- Triggers define actions to be executed automatically when certain events occur and corresponding conditions are satisfied. Triggers have many uses, such as business rule implementation and audit logging. They may carry out actions outside the database system by means of external language routines.
- Some queries, such as transitive closure, can be expressed either by using iteration or by using recursive SQL queries. Recursion can be expressed using either recursive views or recursive **with** clause definitions.
- SQL supports several advanced aggregation features, including ranking and windowing queries, as well as pivot, and rollup/cube operations. These simplify the expression of some aggregates and allow more efficient evaluation.

Review Terms

- JDBC
- Prepared statements
- SQL injection
- Metadata
- Updatable result sets
- Open Database Connectivity (ODBC)
- Embedded SQL
- Embedded database
- Stored procedures and functions
- Table functions.
- Parameterized views
- Persistent Storage Module (PSM).
- Exception conditions
- Handlers
- External language routines
- Sandbox
- Trigger
- Transitive closure
- Hierarchies

- Create temporary table
- Base query
- Recursive query
- Fixed point
- Monotonic
- Windowing
- Ranking functions
- Cross-tabulation
- Cross-tab
- Pivot-table
- Pivot
- SQL **group by cube, group by rollup**

Practice Exercises

5.1 Consider the following relations for a company database:

- *emp (ename, dname, salary)*
- *mgr (ename, mname)*

and the Java code in Figure 5.20, which uses the JDBC API. Assume that the userid, password, machine name, etc. are all okay. Describe in concise English what the Java program does. (That is, produce an English sentence like “It finds the manager of the toy department,” not a line-by-line description of what each Java statement does.)

5.2 Write a Java method using JDBC metadata features that takes a `ResultSet` as an input parameter and prints out the result in tabular form, with appropriate names as column headings.

5.3 Suppose that we wish to find all courses that must be taken before some given course. That means finding not only the prerequisites of that course, but prerequisites of prerequisites, and so on. Write a complete Java program using JDBC that:

- Takes a *course_id* value from the keyboard.
- Finds prerequisites of that course using an SQL query submitted via JDBC.
- For each course returned, finds its prerequisites and continues this process iteratively until no new prerequisite courses are found.
- Prints out the result.

For this exercise, do not use a recursive SQL query, but rather use the iterative approach described previously. A well-developed solution will be robust to the error case where a university has accidentally created a cycle of prerequisites (that is, for example, course *A* is a prerequisite for course *B*, course *B* is a prerequisite for course *C*, and course *C* is a prerequisite for course *A*).

```
import java.sql.*;
public class Mystery {
    public static void main(String[] args) {
        try {
            Connection con=DriverManager.getConnection(
                "jdbc:oracle:thin:star/X@//edgar.cse.lehigh.edu:1521/XE");
            q = "select mname from mgr where ename = ?";
            PreparedStatement stmt=con.prepareStatement();
        }
        {
            String q;
            String empName = "dog";
            boolean more;
            ResultSet result;
            do {
                stmt.setString(1, empName);
                result = stmt.executeQuery(q);
                more = result.next();
                if (more) {
                    empName = result.getString("mname");
                    System.out.println (empName);
                }
            } while (more);
            s.close();
            con.close();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Figure 5.20 Java code for Exercise 5.1 (using Oracle JDBC).

- 5.4 Describe the circumstances in which you would choose to use embedded SQL rather than SQL alone or only a general-purpose programming language.
- 5.5 Show how to enforce the constraint “an instructor cannot teach two different sections in a semester in the same time slot.” using a trigger (remember that the constraint can be violated by changes to the *teaches* relation as well as to the *section* relation).

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

Figure 5.21 Banking database for Exercise 5.6.

- 5.6** Consider the bank database of Figure 5.21. Let us define a view *branch_cust* as follows:

```
create view branch_cust as
    select branch_name, customer_name
    from depositor, account
    where depositor.account_number = account.account_number
```

Suppose that the view is *materialized*; that is, the view is computed and stored. Write triggers to *Maintain* the view, that is, to keep it up-to-date on insertions to *depositor* or *account*. It is not necessary to handle deletions or updates. Note that, for simplicity, we have not required the elimination of duplicates.

- 5.7** Consider the bank database of Figure 5.21. Write an SQL trigger to carry out the following action: On **delete** of an account, for each customer-owner of the account, check if the owner has any remaining accounts, and if she does not, delete her from the *depositor* relation.
- 5.8** Given a relation *S(student, subject, marks)*, write a query to find the top 10 students by total marks, by using SQL ranking. Include all students tied for the final spot in the ranking, even if that results in more than 10 total students.
- 5.9** Given a relation *nyse(year, month, day, shares_traded, dollar_volume)* with trading data from the New York Stock Exchange, list each trading day in order of number of shares traded, and show each day's rank.
- 5.10** Using the relation from Exercise 5.9, write an SQL query to generate a report showing the number of shares traded, number of trades, and total dollar volume broken down by year, each month of each year, and each trading day.
- 5.11** Show how to express **group by cube(a, b, c, d)** using **rollup**; your answer should have only one **group by** clause.

Exercises

5.12 Write a Java program that allows university administrators to print the teaching record of an instructor.

- a. Start by having the user input the login *ID* and password; then open the proper connection.
- b. The user is asked next for a search substring and the system returns (*ID*, *name*) pairs of instructors whose names match the substring. Use the `like ('%substring%)` construct in SQL to do this. If the search comes back empty, allow continued searches until there is a nonempty result.
- c. Then the user is asked to enter an ID number, which is a number between 0 and 99999. Once a valid number is entered, check if an instructor with that ID exists. If there is no instructor with the given ID, print a reasonable message and quit.
- d. If the instructor has taught no courses, print a message saying that. Otherwise print the teaching record for the instructor, showing the department name, course identifier, course title, section number, semester, year, and total enrollment (and sort those by `dept_name, course_id, year, semester`).

Test carefully for bad input. Make sure your SQL queries won't throw an exception. At login, exceptions may occur since the user might type a bad password, but catch those exceptions and allow the user to try again.

5.13 Suppose you were asked to define a class `MetaDisplay` in Java, containing a method `static void printTable(String r)`; the method takes a relation name *r* as input, executes the query "`select * from r`", and prints the result out in tabular format, with the attribute names displayed in the header of the table.

- a. What do you need to know about relation *r* to be able to print the result in the specified tabular format?
- b. What JDBC methods(s) can get you the required information?
- c. Write the method `printTable(String r)` using the JDBC API.

5.14 Repeat Exercise 5.13 using ODBC, defining `void printTable(char *r)` as a function instead of a method.

5.15 Consider an employee database with two relations

employee (*employee_name*, *street*, *city*)
works (*employee_name*, *company_name*, *salary*)

where the primary keys are underlined. Write a function *avg_salary* that takes a company name as an argument and finds the average salary of employees at that company. Then, write an SQL statement, using that function, to find companies whose employees earn a higher salary, on average, than the average salary at “First Bank”.

- 5.16** Consider the relational schema

$$\begin{aligned} & \textit{part}(\underline{\textit{part_id}}, \textit{name}, \textit{cost}) \\ & \textit{subpart}(\underline{\textit{part_id}}, \underline{\textit{subpart_id}}, \textit{count}) \end{aligned}$$

where the primary-key attributes are underlined. A tuple $(p_1, p_2, 3)$ in the *subpart* relation denotes that the part with *part_id* p_2 is a direct subpart of the part with *part_id* p_1 , and p_1 has 3 copies of p_2 . Note that p_2 may itself have further subparts. Write a recursive SQL query that outputs the names of all subparts of the part with part-id ‘P-100’.

- 5.17** Consider the relational schema from Exercise 5.16. Write a JDBC function using nonrecursive SQL to find the total cost of part “P-100”, including the costs of all its subparts. Be sure to take into account the fact that a part may have multiple occurrences of a subpart. You may use recursion in Java if you wish.
- 5.18** Redo Exercise 5.12 using the language of your database system for coding stored procedures and functions. Note that you are likely to have to consult the online documentation for your system as a reference, since most systems use syntax differing from the SQL standard version followed in the text. Specifically, write a procedure that takes an instructor *ID* as an argument and produces printed output in the format specified in Exercise 5.12, or an appropriate message if the instructor does not exist or has taught no courses. (For a simpler version of this exercise, rather than providing printed output, assume a relation with the appropriate schema and insert your answer there without worrying about testing for erroneous argument values.)
- 5.19** Suppose there are two relations *r* and *s*, such that the foreign key *B* of *r* references the primary key *A* of *s*. Describe how the trigger mechanism can be used to implement the **on delete cascade** option when a tuple is deleted from *s*.
- 5.20** The execution of a trigger can cause another action to be triggered. Most database systems place a limit on how deep the nesting can be. Explain why they might place such a limit.
- 5.21** Modify the recursive query in Figure 5.16 to define a relation

$$\textit{prereq_depth}(\textit{course_id}, \textit{prereq_id}, \textit{depth})$$

<i>building</i>	<i>room_number</i>	<i>time_slot_id</i>	<i>course_id</i>	<i>sec_id</i>
Garfield	359	A	BIO-101	1
Garfield	359	B	BIO-101	2
Saucon	651	A	CS-101	2
Saucon	550	C	CS-319	1
Painter	705	D	MU-199	1
Painter	403	D	FIN-201	1

Figure 5.22 The relation *r* for Exercise 5.24.

where the attribute *depth* indicates how many levels of intermediate prerequisites there are between the course and the prerequisite. Direct prerequisites have a depth of 0. Note that a prerequisite course may have multiple depths and thus may appear more than once.

- 5.22 Given relation $s(a, b, c)$, write an SQL statement to generate a histogram showing the sum of c values versus a , dividing a into 20 equal-sized partitions (i.e., where each partition contains 5 percent of the tuples in s , sorted by a).
- 5.23 Consider the *nyse* relation of Exercise 5.9. For each month of each year, show the total monthly dollar volume and the average monthly dollar volume for that month and the two prior months. (*Hint:* First write a query to find the total dollar volume for each month of each year. Once that is right, put that in the from clause of the outer query that solves the full problem. That outer query will need windowing. The subquery does not.)
- 5.24 Consider the relation, *r*, shown in Figure 5.22. Give the result of the following query:

```
select building, room_number, time_slot_id, count(*)
from r
group by rollup (building, room_number, time_slot_id)
```

Tools

We provide sample JDBC code on our book web site db-book.com.

Most database vendors, including IBM, Microsoft, and Oracle, provide OLAP tools as part of their database systems, or as add-on applications. Tools may be integrated with a larger “business intelligence” product such as IBM Cognos. Many companies also provide analysis tools for specific applications, such as customer relationship management (e.g., Oracle Siebel CRM).

Further Reading

More details about JDBC may be found at docs.oracle.com/javase/tutorial/jdbc.

In order to write stored procedures, stored functions, and triggers that can be executed on a given system, you need to refer to the system documentation.

Although our discussion of recursive queries focused on SQL syntax, there are other approaches to recursion in relational databases. Datalog is a database language based on the Prolog programming language and is described in more detail in Section 27.4 (available online).

OLAP features in SQL, including rollup, and cubes were introduced in SQL:1999, and window functions with ranking and partitioning were added in SQL:2003. OLAP features, including window functions, are supported by most databases today. Although most follow the SQL standard syntax that we have presented, there are some differences; refer to the system manuals of the system that you are using for further details. Microsoft's Multidimensional Expressions (MDX) is an SQL-like query language designed for querying OLAP cubes.

Credits

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

CHAPTER 5



Advanced SQL

Practice Exercises

5.1 Consider the following relations for a company database:

- *emp (ename, dname, salary)*
- *mgr (ename, mname)*

and the Java code in Figure 5.20, which uses the JDBC API. Assume that the userid, password, machine name, etc. are all okay. Describe in concise English what the Java program does. (That is, produce an English sentence like “It finds the manager of the toy department,” not a line-by-line description of what each Java statement does.)

Answer:

It prints out the manager of “dog,” that manager’s manager, etc., until we reach a manager who has no manager (presumably, the CEO, who most certainly is a cat). Note: If you try to run this, use your own Oracle ID and password.

5.2 Write a Java method using JDBC metadata features that takes a `ResultSet` as an input parameter and prints out the result in tabular form, with appropriate names as column headings.

Answer:

Please see ??

5.3 Suppose that we wish to find all courses that must be taken before some given course. That means finding not only the prerequisites of that course, but prerequisites of prerequisites, and so on. Write a complete Java program using JDBC that:

- Takes a *course_id* value from the keyboard.
- Finds prerequisites of that course using an SQL query submitted via JDBC.

```

import java.sql.*;
public class Mystery {
    public static void main(String[] args) {
        try {
            Connection con=DriverManager.getConnection(
                "jdbc:oracle:thin:star/X@//edgar.cse.lehigh.edu:1521/XE");
            q = "select mname from mgr where ename = ?";
            PreparedStatement stmt=con.prepareStatement();
        }
        {
            String q;
            String empName = "dog";
            boolean more;
            ResultSet result;
            do {
                stmt.setString(1, empName);
                result = stmt.executeQuery(q);
                more = result.next();
                if (more) {
                    empName = result.getString("mname");
                    System.out.println(empName);
                }
            } while (more);
            s.close();
            con.close();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}

```

Figure 5.20 Java code for Exercise 5.1 (using Oracle JDBC).

- For each course returned, finds its prerequisites and continues this process iteratively until no new prerequisite courses are found.
- Prints out the result.

For this exercise, do not use a recursive SQL query, but rather use the iterative approach described previously. A well-developed solution will be robust to the error case where a university has accidentally created a cycle of prerequisites (that is, for example, course *A* is a prerequisite for course *B*, course *B* is a prerequisite for course *C*, and course *C* is a prerequisite for course *A*).

```

printTable(ResultSet result) throws SQLException {
    metadata = result.getMetaData();
    num_cols = metadata.getColumnCount();
    for(int i = 1; i <= num_cols; i++) {
        System.out.print(metadata.getColumnName(i) + '\t');
    }
    System.out.println();
    while(result.next()) {
        for(int i = 1; i <= num_cols; i++) {
            System.out.print(result.getString(i) + '\t'
        }
        System.out.println();
    }
}

```

Figure 5.101 Java method using JDBC for Exercise 5.2.**Answer:**

Please see ??

- 5.4** Describe the circumstances in which you would choose to use embedded SQL rather than SQL alone or only a general-purpose programming language.

Answer:

Writing queries in SQL is typically much easier than coding the same queries in a general-purpose programming language. However, not all kinds of queries can be written in SQL. Also, nondeclarative actions such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface cannot be done from within SQL. Under circumstances in which we want the best of both worlds, we can choose embedded SQL or dynamic SQL, rather than using SQL alone or using only a general-purpose programming language.

- 5.5** Show how to enforce the constraint “an instructor cannot teach two different sections in a semester in the same time slot.” using a trigger (remember that the constraint can be violated by changes to the *teaches* relation as well as to the *section* relation).

Answer:

Please see ??

- 5.6** Consider the bank database of Figure 5.21. Let us define a view *branch_cust* as follows:

```

import java.sql.*;
import java.util.Scanner;
import java.util.Arrays;
public class AllCoursePrereqs {
    public static void main(String[] args) {
        try {
            Connection con=DriverManager.getConnection
                ("jdbc:oracle:thin:@edgar0.cse.lehigh.edu:1521:cse241","star","pw");
            Statement s=con.createStatement();
        }{
            String q;
            String c;
            ResultSet result;
            int maxCourse = 0;
            q = "select count(*) as C from course";
            result = s.executeQuery(q);
            if (!result.next()) System.out.println ("Unexpected empty result.");
            else maxCourse = Integer.parseInt(result.getString("C"));
            int numCourse = 0, oldNumCourse = -1;
            String[] prereqs = new String [maxCourse];
            Scanner krb = new Scanner(System.in);
            System.out.print("Input a course id (number): ");
            String course = krb.nextLine();
            String courseString = "" + '\'\' + course + '\'\'';
            while (numCourse != oldNumCourse) {
                for (int i = oldNumCourse + 1; i < numCourse; i++) {
                    courseString += ", " + '\'\' + prereqs[i] + '\'\'';
                }
                oldNumCourse = numCourse;
                q = "select prereq_id from prereq where course_id in (" +
                    courseString + ")";
                result = s.executeQuery(q);
                while (result.next()) {
                    c = result.getString("prereq_id");
                    boolean found = false;
                    for (int i = 0; i < numCourse; i++)
                        found |= prereqs[i].equals(c);
                    if (!found) prereqs[numCourse++] = c;
                }
                courseString = "" + '\'\' + prereqs[oldNumCourse] + '\'\'';
            }
            Arrays.sort(prereqs,0,numCourse);
            System.out.print("The courses that must be taken prior to " +
                course + " are: ");
            for (int i = 0; i < numCourse; i++)
                System.out.print ((i==0?" ":"") + prereqs[i]);
            System.out.println();
        } catch(Exception e){e.printStackTrace();
    }
}

```

Figure 5.102 Complete Java program using JDBC for Exercise 5.3.

```

create trigger onesecc before insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id in (
    select time_slot_id
    from teaches natural join section
    where ID in (
        select ID
        from teaches natural join section
        where sec_id = nrow.sec_id and course_id = nrow.course_id and
            semester = nrow.semester and year = nrow.year
    )))
begin
    rollback
end;

```

```

create trigger oneteach before insert on teaches
referencing new row as nrow
for each row
when (exists (
    select time_slot_id
    from teaches natural join section
    where ID = nrow.ID
    intersect
        select time_slot_id
        from section
        where sec_id = nrow.sec_id and course_id = nrow.course_id and
            semester = nrow.semester and year = nrow.year
))
begin
    rollback
end;

```

Figure 5.103 Trigger code for Exercise 5.5.

```

create view branch_cust as
    select branch_name, customer_name
    from depositor, account
    where depositor.account_number = account.account_number

```

```

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

```

Figure 5.21 Banking database for Exercise 5.6.

Suppose that the view is *materialized*; that is, the view is computed and stored. Write triggers to *Maintain* the view, that is, to keep it up-to-date on insertions to *depositor* or *account*. It is not necessary to handle deletions or updates. Note that, for simplicity, we have not required the elimination of duplicates.

Answer:

Please see ??

- 5.7** Consider the bank database of Figure 5.21. Write an SQL trigger to carry out the following action: On **delete** of an account, for each customer-owner of the

```

create trigger insert_into_branch_cust_via_depositor
after insert on depositor
referencing new row as inserted
for each row
insert into branch_cust
    select branch_name, inserted.customer_name
    from account
    where inserted.account_number = account.account_number

create trigger insert_into_branch_cust_via_account
after insert on account
referencing new row as inserted
for each statement
insert into branch_cust
    select inserted.branch_name, customer_name
    from depositor
    where depositor.account_number = inserted.account_number

```

Figure 5.22 Trigger code for Exercise 5.6.

account, check if the owner has any remaining accounts, and if she does not, delete her from the *depositor* relation.

Answer:

```
create trigger check-delete-trigger after delete on account
referencing old row as orow
for each row
delete from depositor
where depositor.customer_name not in
( select customer_name from depositor
  where account_number <> orow.account_number )
end
```

- 5.8** Given a relation $S(\text{student}, \text{subject}, \text{marks})$, write a query to find the top 10 students by total marks, by using SQL ranking. Include all students tied for the final spot in the ranking, even if that results in more than 10 total students.

Answer:

```
select *
from (
  select student, total, rank() over (order by (total) desc) as t_rank
  from (
    select student, sum(marks) as total
    from S group by student
  )
)
)
where t_rank <= 10
```

- 5.9** Given a relation $nyse(\text{year}, \text{month}, \text{day}, \text{shares_traded}, \text{dollar_volume})$ with trading data from the New York Stock Exchange, list each trading day in order of number of shares traded, and show each day's rank.

Answer:

```
select year, month, day, shares_traded,
       rank() over (order by shares_traded desc ) as mostshares
  from nyse
```

- 5.10** Using the relation from Exercise 5.9, write an SQL query to generate a report showing the number of shares traded, number of trades, and total dollar volume broken down by year, each month of each year, and each trading day.

Answer:

```
select year, month, day, sum(shares_traded) as shares,
       sum(num_trades) as trades, sum(dollar_volume) as total_volume
  from nyse
 group by rollup (year, month, day)
```

- 5.11** Show how to express **group by cube(a, b, c, d)** using **rollup**; your answer should have only one **group by** clause.

Answer:

```
groupby rollup(a), rollup(b), rollup(c ), rollup(d)
```

CHAPTER 9



Application Development

Practically all use of databases occurs from within application programs. Correspondingly, almost all user interaction with databases is indirect, via application programs. In this chapter, we study tools and technologies that are used to build applications, focusing on interactive applications that use databases to store and retrieve data.

A key requirement for any user-centric application is a good user interface. The two most common types of user interfaces today for database-backed applications are the web and mobile app interfaces.

In the initial part of this chapter, we provide an introduction to application programs and user interfaces (Section 9.1), and to web technologies (Section 9.2). We then discuss development of web applications using the widely used Java Servlets technology at the back end (Section 9.3), and using other frameworks (Section 9.4). Client-side code implemented using JavaScript or mobile app technologies is crucial for building responsive user interfaces, and we discuss some of these technologies (Section 9.5). We then provide an overview of web application architectures (Section 9.6) and cover performance issues in building large web applications (Section 9.7). Finally, we discuss issues in application security that are key to making applications resilient to attacks (Section 9.8), and encryption and its use in applications (Section 9.9).

9.1

Application Programs and User Interfaces

Although many people interact with databases, very few people use a query language to interact with a database system directly. The most common way in which users interact with databases is through an **application program** that provides a user interface at the front end and interfaces with a database at the back end. Such applications take input from users, typically through a forms-based interface, and either enter data into a database or extract information from a database based on the user input, and they then generate output, which is displayed to the user.

As an example of an application, consider a university registration system. Like other such applications, the registration system first requires you to identify and authenticate yourself, typically by a user name and password. The application then uses your

identity to extract information, such as your name and the courses for which you have registered, from the database and displays the information. The application provides a number of interfaces that let you register for courses and query other information, such as course and instructor information. Organizations use such applications to automate a variety of tasks, such as sales, purchases, accounting and payroll, human-resources management, and inventory management, among many others.

Application programs may be used even when it is not apparent that they are being used. For example, a news site may provide a page that is transparently customized to individual users, even if the user does not explicitly fill any forms when interacting with the site. To do so, it actually runs an application program that generates a customized page for each user; customization can, for example, be based on the history of articles browsed by the user.

A typical application program includes a front-end component, which deals with the user interface, a backend component, which communicates with a database, and a middle layer, which contains “business logic,” that is, code that executes specific requests for information or updates, enforcing rules of business such as what actions should be carried out to execute a given task or who can carry out what task.

Applications such as airline reservations have been around since the 1960s. In the early days of computer applications, applications ran on large “mainframe” computers, and users interacted with the application through terminals, some of which even supported forms. The growth of personal computers resulted in the development of database applications with graphical user interfaces, or GUIs. These interfaces depended on code running on a personal computer that directly communicated with a shared database. Such an architecture was called a *client-server architecture*. There were two drawbacks to using such applications: first, user machines had direct access to databases, leading to security risks. Second, any change to the application or the database required all the copies of the application, located on individual computers, to be updated together.

Two approaches have evolved to avoid the above problems:

- Web browsers provide a *universal front end*, used by all kinds of information services. Browsers use a standardized syntax, the **HyperText Markup Language (HTML)** standard, which supports both formatted display of information and creation of forms-based interfaces. The HTML standard is independent of the operating system or browser, and pretty much every computer today has a web browser installed. Thus a web-based application can be accessed from any computer that is connected to the internet.

Unlike client-server architectures, there is no need to install any application-specific software on client machines in order to use web-based applications.

However, sophisticated user interfaces, supporting features well beyond what is possible using plain HTML, are now widely used, and are built with the scripting language JavaScript, which is supported by most web browsers. JavaScript programs, unlike programs written in C, can be run in a safe mode, guaranteeing

they cannot cause security problems. JavaScript programs are downloaded transparently to the browser and do not need any explicit software installation on the user's computer.

While the web browser provides the front end for user interaction, application programs constitute the back end. Typically, requests from a browser are sent to a web server, which in turn executes an application program to process the request. A variety of technologies are available for creating application programs that run at the back end, including Java servlets, Java Server Pages (JSP), Active Server Page (ASP), or scripting languages such as PHP and Python.

- Application programs are installed on individual devices, which are primarily mobile devices. They communicate with backend applications through an API and do not have direct access to the database. The back end application provides services, including user authentication, and ensures that users can only access services that they are authorized to access.

This approach is widely used in mobile applications. One of the motivations for building such applications was to customize the display for the small screen of mobile devices. A second was to allow application code, which can be relatively large, to be downloaded or updated when the device is connected to a high-speed network, instead of downloading such code when a web page is accessed, perhaps over a lower bandwidth or more expensive mobile network.

With the increasing use of JavaScript code as part of web front ends, the difference between the two approaches above has today significantly decreased. The back end often provides an API that can be invoked from either mobile app or JavaScript code to carry out any required task at the back end. In fact, the same back end is often used to build multiple front ends, which could include web front ends with JavaScript, and multiple mobile platforms (primarily Android and iOS, today).

9.2 Web Fundamentals

In this section, we review some of the fundamental technology behind the World Wide Web, for readers who are not familiar with the technology underlying the web.

9.2.1 Uniform Resource Locators

A **uniform resource locator (URL)** is a globally unique name for each document that can be accessed on the web. An example of a URL is:

`http://www.acm.org/sigmod`

The first part of the URL indicates how the document is to be accessed: “http” indicates that the document is to be accessed by the **HyperText Transfer Protocol (HTTP)**,

```
<html>
<body>
<table border>
<tr> <th>ID</th>      <th>Name</th>      <th>Department</th> </tr>
<tr> <td>00128</td> <td>Zhang</td> <td>Comp. Sci.</td> </tr>
<tr> <td>12345</td> <td>Shankar</td> <td>Comp. Sci.</td> </tr>
<tr> <td>19991</td> <td>Brandt</td> <td>History</td> </tr>
</table>
</body>
</html>
```

Figure 9.1 Tabular data in HTML format.

which is a protocol for transferring HTML documents; “https” would indicate that the secure version of the HTTP protocol must be used, and is the preferred mode today. The second part gives the name of a machine that has a web server. The rest of the URL is the path name of the file on the machine, or other unique identifier of a document within the machine.

A URL can contain the identifier of a program located on the web server machine, as well as arguments to be given to the program. An example of such a URL is

<https://www.google.com/search?q=silberschatz>

which says that the program search on the server `www.google.com` should be executed with the argument `q=silberschatz`. On receiving a request for such a URL, the web server executes the program, using the given arguments. The program returns an HTML document to the web server, which sends it back to the front end.

9.2.2 HyperText Markup Language

Figure 9.1 is an example of a table represented in the HTML format, while Figure 9.2 shows the displayed image generated by a browser from the HTML representation of the table. The HTML source shows a few of the HTML tags. Every HTML page should be enclosed in an `html` tag, while the body of the page is enclosed in a `body` tag. A table

ID	Name	Department
00128	Zhang	Comp. Sci.
12345	Shankar	Comp. Sci.
19991	Brandt	History

Figure 9.2 Display of HTML source from Figure 9.1.

```

<html>
<body>
<form action="PersonQuery" method=get>
Search for:
<select name="persontype">
    <option value="student" selected>Student </option>
    <option value="instructor"> Instructor </option>
</select> <br>
Name: <input type=text size=20 name="name">
<input type=submit value="submit">
</form>
</body>
</html>

```

Figure 9.3 An HTML form.

is specified by a **table** tag, which contains rows specified by a **tr** tag. The header row of the table has table cells specified by a **th** tag, while regular rows have table cells specified by a **td** tag. We do not go into more details about the tags here; see the bibliographical notes for references containing more detailed descriptions of HTML.

Figure 9.3 shows how to specify an HTML form that allows users to select the person type (student or instructor) from a menu and to input a number in a text box. Figure 9.4 shows how the above form is displayed in a web browser. Two methods of accepting input are illustrated in the form, but HTML also supports several other input methods. The **action** attribute of the **form** tag specifies that when the form is submitted (by clicking on the submit button), the form data should be sent to the URL **PersonQuery** (the URL is relative to that of the page). The web server is configured such that when this URL is accessed, a corresponding application program is invoked, with the user-provided values for the arguments **persontype** and **name** (specified in the **select** and **input** fields). The application program generates an HTML document, which is then sent back and displayed to the user; we shall see how to construct such programs later in this chapter.

HTTP defines two ways in which values entered by a user at the browser can be sent to the web server. The **get** method encodes the values as part of the URL. For example, if the Google search page used a form with an input parameter named **q** with the **get**

Figure 9.4 Display of HTML source from Figure 9.3.

method, and the user typed in the string “silberschatz” and submitted the form, the browser would request the following URL from the web server:

<https://www.google.com/search?q=silberschatz>

The post method would instead send a request for the URL <https://www.google.com>, and send the parameter values as part of the HTTP protocol exchange between the web server and the browser. The form in Figure 9.3 specifies that the form uses the get method.

Although HTML code can be created using a plain text editor, there are a number of editors that permit direct creation of HTML text by using a graphical interface. Such editors allow constructs such as forms, menus, and tables to be inserted into the HTML document from a menu of choices, instead of manually typing in the code to generate the constructs.

HTML supports *stylesheets*, which can alter the default definitions of how an HTML formatting construct is displayed, as well as other display attributes such as background color of the page. The *cascading stylesheet* (CSS) standard allows the same stylesheet to be used for multiple HTML documents, giving a distinctive but uniform look to all the pages on a web site. You can find more information on stylesheets online, for example at www.w3schools.com/css/.

The HTML5 standard, which was released in 2014, provides a wide variety of form input types, including the following:

- Date and time selection, using `<input type="date" name="abc">`, and `<input type="time" name="xyz">`. Browsers would typically display a graphical date or time picker for such an input field; the input value is saved in the form attributes `abc` and `xyz`. The optional attributes `min` and `max` can be used to specify minimum and maximum values that can be chosen.
- File selection, using `<input type="file", name="xyz">`, which allows a file to be chosen, and its name saved in the form attribute `xyz`.
- Input restrictions (constraints) on a variety of input types, including minimum, maximum, format matching a regular expression, and so on. For example, `<input type="number" name="start" min="0" max="55" step="5" value="0">` allows the user to choose one of 0, 5, 10, 15, and so on till 55, with a default value of 0.

9.2.3 Web Servers and Sessions

A **web server** is a program running on the server machine that accepts requests from a web browser and sends back results in the form of HTML documents. The browser and web server communicate via HTTP. Web servers provide powerful features, beyond the simple transfer of documents. The most important feature is the ability to execute

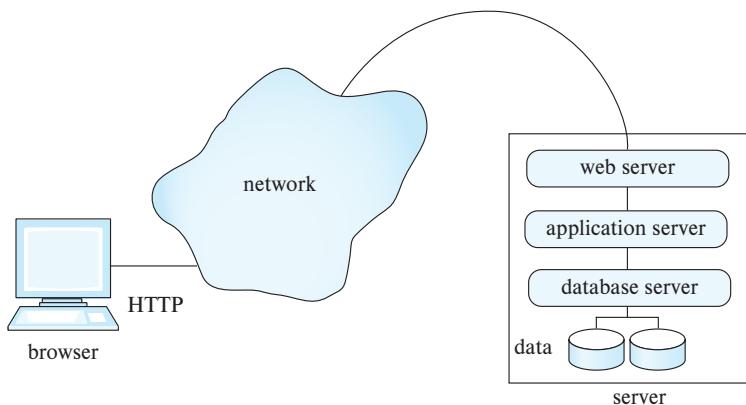


Figure 9.5 Three-layer web application architecture.

programs, with arguments supplied by the user, and to deliver the results back as an HTML document.

As a result, a web server can act as an intermediary to provide access to a variety of information services. A new service can be created by creating and installing an application program that provides the service. The **common gateway interface (CGI)** standard defines how the web server communicates with application programs. The application program typically communicates with a database server, through ODBC, JDBC, or other protocols, in order to get or store data.

Figure 9.5 shows a web application built using a three-layer architecture, with a web server, an application server, and a database server. Using multiple levels of servers increases system overhead; the CGI interface starts a new process to service each request, which results in even greater overhead.

Most web applications today therefore use a two-layer web application architecture, where the web and application servers are combined into a single server, as shown in Figure 9.6. We study systems based on the two-layer architecture in more detail in subsequent sections.

There is no continuous connection between the client and the web server; when a web server receives a request, a connection is temporarily created to send the request and receive the response from the web server. But the connection may then be closed, and the next request could come over a new connection. In contrast, when a user logs on to a computer, or connects to a database using ODBC or JDBC, a session is created, and session information is retained at the server and the client until the session is terminated—information such as the user-identifier of the user and session options that the user has set. One important reason that HTTP is **connectionless** is that most computers have limits on the number of simultaneous connections they can accommodate, and if a large number of sites on the web open connections to a single server, this limit would be exceeded, denying service to further users. With a connectionless protocol, the con-

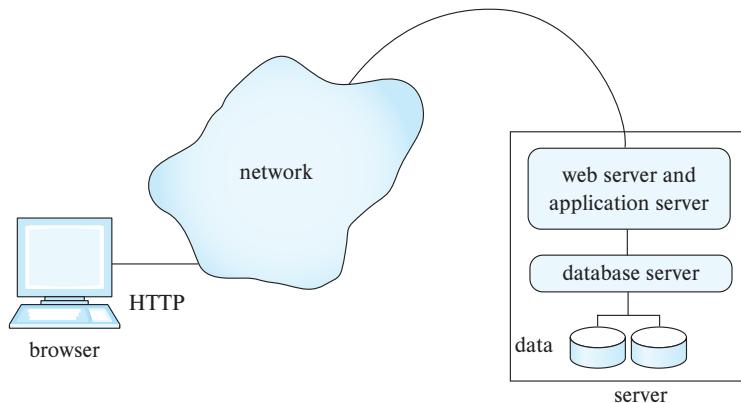


Figure 9.6 Two-layer web application architecture.

nection can be broken as soon as a request is satisfied, leaving connections available for other requests.¹

Most web applications, however, need session information to allow meaningful user interaction. For instance, applications typically restrict access to information, and therefore need to authenticate users. Authentication should be done once per session, and further interactions in the session should not require reauthentication.

To implement sessions in spite of connections getting closed, extra information has to be stored at the client and returned with each request in a session; the server uses this information to identify that a request is part of a user session. Extra information about the session also has to be maintained at the server.

This extra information is usually maintained in the form of a **cookie** at the client; a cookie is simply a small piece of text containing identifying information and with an associated name. For example, google.com may set a cookie with the name `prefs`, which encodes preferences set by the user such as the preferred language and the number of answers displayed per page. On each search request, google.com can retrieve the cookie named `prefs` from the user's browser, and display results according to the specified preferences. A domain (web site) is permitted to retrieve only cookies that it has set, not cookies set by other domains, and cookie names can be reused across domains.

For the purpose of tracking a user session, an application may generate a session identifier (usually a random number not currently in use as a session identifier), and send a cookie named (for instance) `sessionid` containing the session identifier. The session identifier is also stored locally at the server. When a request comes in, the application server requests the cookie named `sessionid` from the client. If the client

¹For performance reasons, connections may be kept open for a short while, to allow subsequent requests to reuse the connection. However, there is no guarantee that the connection will be kept open, and applications must be designed assuming the connection may be closed as soon as a request is serviced.

does not have the cookie stored, or returns a value that is not currently recorded as a valid session identifier at the server, the application concludes that the request is not part of a current session. If the cookie value matches a stored session identifier, the request is identified as part of an ongoing session.

If an application needs to identify users securely, it can set the cookie only after authenticating the user; for example a user may be authenticated only when a valid user name and password are submitted.²

For applications that do not require high security, such as publicly available news sites, cookies can be stored permanently at the browser and at the server; they identify the user on subsequent visits to the same site, without any identification information being typed in. For applications that require higher security, the server may invalidate (drop) the session after a time-out period, or when the user logs out. (Typically a user logs out by clicking on a logout button, which submits a logout form, whose action is to invalidate the current session.) Invalidating a session merely consists of dropping the session identifier from the list of active sessions at the application server.

9.3

Servlets

The Java **servlet** specification defines an application programming interface for communication between the web/application server and the application program. The `HttpServlet` class in Java implements the servlet API specification; servlet classes used to implement specific functions are defined as subclasses of this class.³ Often the word *servlet* is used to refer to a Java program (and class) that implements the servlet interface. Figure 9.7 shows a servlet example; we explain it in detail shortly.

The code for a servlet is loaded into the web/application server when the server is started, or when the server receives a remote HTTP request to execute a particular servlet. The task of a servlet is to process such a request, which may involve accessing a database to retrieve necessary information, and dynamically generating an HTML page to be returned to the client browser.

9.3.1 A Servlet Example

Servlets are commonly used to generate dynamic responses to HTTP requests. They can access inputs provided through HTML forms, apply “business logic” to decide what

²The user identifier could be stored at the client end, in a cookie named, for example, `userid`. Such cookies can be used for low-security applications, such as free web sites identifying their users. However, for applications that require a higher level of security, such a mechanism creates a security risk: The value of a cookie can be changed at the browser by a malicious user, who can then masquerade as a different user. Setting a cookie (named `sessionid`, for example) to a randomly generated session identifier (from a large space of numbers) makes it highly improbable that a user can masquerade as (i.e., pretend to be) another user. A sequentially generated session identifier, on the other hand, would be susceptible to masquerading.

³The servlet interface can also support non-HTTP requests, although our example uses only HTTP.

response to provide, and then generate HTML output to be sent back to the browser. Servlet code is executed on a web or application server.

Figure 9.7 shows an example of servlet code to implement the form in Figure 9.3. The servlet is called `PersonQueryServlet`, while the form specifies that “action=“PersonQuery”.” The web/application server must be told that this servlet is to be used to handle requests for `PersonQuery`, which is done by using the anno-

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

@WebServlet("PersonQuery")
public class PersonQueryServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        ... check if user is logged in ...
        out.println("<HEAD><TITLE> Query Result</TITLE></HEAD>");
        out.println("<BODY>");

        String personotype = request.getParameter("personotype");
        String name = request.getParameter("name");
        if(personotype.equals("student")) {
            ... code to find students with the specified name ...
            ... using JDBC to communicate with the database ..
            ... Assume ResultSet rs has been retrieved, and
            ... contains attributes ID, name, and department name
            String headers = new String[]{"ID", "Name", "Department Name"};
            Util::resultSetToHTML(rs, headers, out);
        }
        else {
            ... as above, but for instructors ...
        }
        out.println("</BODY>");
        out.close();
    }
}
```

Figure 9.7 Example of servlet code.

tation `@.WebServlet("PersonQuery")` shown in the code. The form specifies that the HTTP get mechanism is used for transmitting parameters. So the `doGet()` method of the servlet, as defined in the code, is invoked.

Each servlet request results in a new thread within which the call is executed, so multiple requests can be handled in parallel. Any values from the form menus and input fields on the web page, as well as cookies, pass through an object of the `HttpServletRequest` class that is created for the request, and the reply to the request passes through an object of the class `HttpServletResponse`.

The `doGet()` method in the example extracts values of the parameters `personType` and `name` by using `request.getParameter()`, and uses these values to run a query against a database. The code used to access the database and to get attribute values from the query result is not shown; refer to Section 5.1.1.5 for details of how to use JDBC to access a database. We assume that the result of the query in the form of a JDBC `ResultSet` is available in the variable `resultset`.

The servlet code returns the results of the query to the requester by outputting them to the `HttpServletResponse` object `response`. Outputting the results to `response` is implemented by first getting a `PrintWriter` object `out` from `response`, and then printing the query result in HTML format to `out`. In our example, the query result is printed by calling the function `Util::resultSetToHTML(resultset, header, out)`, which is shown in Figure 9.8. The function uses JDBC metadata function on the `ResultSet` `rs` to figure out how many columns need to be printed. An array of column headers is passed to this function to be printed out; the column names could have been obtained using JDBC metadata, but the database column names may not be appropriate for display to a user, so we provide meaningful column names to the function.

9.3.2 Servlet Sessions

Recall that the interaction between a browser and a web/application server is stateless. That is, each time the browser makes a request to the server, the browser needs to connect to the server, request some information, then disconnect from the server. Cookies can be used to recognize that a request is from the same browser session as an earlier request. However, cookies form a low-level mechanism, and programmers require a better abstraction to deal with sessions.

The servlet API provides a method of tracking a session and storing information pertaining to it. Invocation of the method `getSession(false)` of the class `HttpServletRequest` retrieves the `HttpSession` object corresponding to the browser that sent the request. An argument value of `true` would have specified that a new session object must be created if the request is a new request.

When the `getSession()` method is invoked, the server first asks the client to return a cookie with a specified name. If the client does not have a cookie of that name, or returns a value that does not match any ongoing session, then the request is not part of an ongoing session. In this case, `getSession()` would return a null value, and the servlet could direct the user to a login page.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Util {
    public static void resultSetToHTML(ResultSet rs,
                                      String headers[], PrintWriter out) {
        ResultSetMetaData rsmd = rs.getMetaData();
        int numCols = rsmd.getColumnCount();
        out.println("<table border=1>");
        out.println("<tr>");
        for (int i=0; i < numCols; i++)
            out.println("<th>" + headers[i] + "</th>");
        out.println("</tr>");
        while (rs.next()) {
            out.println("<tr>");
            for (int i=0; i < numCols; i++)
                out.println("<td>" + rs.getString(i) + "</td>");
            out.println("</tr>");
        }
    }
}
```

Figure 9.8 Utility function to output ResultSet as a table.

The login page could allow the user to provide a user name and password. The servlet corresponding to the login page could verify that the password matches the user; for example, by using the user name to retrieve the password from the database and checking if the password entered matches the stored password.⁴

If the user is properly authenticated, the login servlet would execute `getSession(true)`, which would return a new session object. To create a new session, the server would internally carry out the following tasks: set a cookie (called, for example, `sessionId`) with a session identifier as its associated value at the client browser, create a new session object, and associate the session identifier value with the session object.

⁴It is a bad idea to store unencrypted passwords in the database, since anyone with access to the database contents, such as a system administrator or a hacker, could steal the password. Instead, a hashing function is applied to the password, and the result is stored in the database; even if someone sees the hashing result stored in the database, it is very hard to infer what was the original password. The same hashing function is applied to the user-supplied password, and the result is compared with the stored hashing result. Further, to ensure that even if two users use the same password the hash values are different, the password system typically stores a different random string (called the *salt*) for each user, and it appends the random string to the password before computing the hash value. Thus, the password relation would have the schema `user_password(user, salt, passwordhash)`, where `passwordhash` is generated by `hash(append(password,salt))`. Encryption is described in more detail in Section 9.9.1.

The servlet code can also store and look up (attribute-name, value) pairs in the HttpSession object, to maintain state across multiple requests within a session. For example, after the user is authenticated and the session object has been created, the login servlet could store the user-id of the user as a session parameter by executing the method

```
session.setAttribute("userid", userid)
```

on the session object returned by getSession(); the Java variable userid is assumed to contain the user identifier.

If the request was part of an ongoing session, the browser would have returned the cookie value, and the corresponding session object would be returned by getSession(). The servlet could then retrieve session parameters such as user-id from the session object by executing the method

```
session.getAttribute("userid")
```

on the session object returned above. If the attribute userid is not set, the function would return a null value, which would indicate that the client user has not been authenticated.

Consider the line in the servlet code in Figure 9.7 that says "... check if user is logged in...". The following code implements this check; if the user is not logged in, it sends an error message, and after a gap of 5 seconds, redirects the user to the login page.

```
Session session = request.getSession(false);
if (session == null || session.getAttribute(userid) == null) {
    out.println("You are not logged in.");
    response.setHeader("Refresh", "5;url=login.html");
    return();
}
```

9.3.3 Servlet Life Cycle

The life cycle of a servlet is controlled by the web/application server in which the servlet has been deployed. When there is a client request for a specific servlet, the server first checks if an instance of the servlet exists or not. If not, the server loads the servlet class into the Java virtual machine (JVM) and creates an instance of the servlet class. In addition, the server calls the init() method to initialize the servlet instance. Notice that each servlet instance is initialized only once when it is loaded.

After making sure the servlet instance does exist, the server invokes the service method of the servlet, with a request object and a response object as parameters. By default, the server creates a new thread to execute the service method; thus, multiple

requests on a servlet can execute in parallel, without having to wait for earlier requests to complete execution. The `service` method calls `doGet` or `doPost` as appropriate.

When no longer required, a servlet can be shut down by calling the `destroy()` method. The server can be set up to shut down a servlet automatically if no requests have been made on a servlet within a time-out period; the time-out period is a server parameter that can be set as appropriate for the application.

9.3.4 Application Servers

Many application servers provide built-in support for servlets. One of the most popular is the Tomcat Server from the Apache Jakarta Project. Other application servers that support servlets include Glassfish, JBoss, BEA Weblogic Application Server, Oracle Application Server, and IBM's WebSphere Application Server.

The best way to develop servlet applications is by using an IDE such as Eclipse or NetBeans, which come with Tomcat or Glassfish servers built in.

Application servers usually provide a variety of useful services, in addition to basic servlet support. They allow applications to be deployed or stopped, and they provide functionality to monitor the status of the application server, including performance statistics. Many application servers also support the Java 2 Enterprise Edition (J2EE) platform, which provides support and APIs for a variety of tasks, such as for handling objects, and parallel processing across multiple application servers.

9.4 Alternative Server-Side Frameworks

There are several alternatives to Java Servlets for processing requests at the application server, including scripting languages and web application frameworks developed for languages such as Python.

9.4.1 Server-Side Scripting

Writing even a simple web application in a programming language such as Java or C is a time-consuming task that requires many lines of code and programmers who are familiar with the intricacies of the language. An alternative approach, that of **server-side scripting**, provides a much easier method for creating many applications. Scripting languages provide constructs that can be embedded within HTML documents.

In server-side scripting, before delivering a web page, the server executes the scripts embedded within the HTML contents of the page. Each piece of script, when executed, can generate text that is added to the page (or may even delete content from the page). The source code of the scripts is removed from the page, so the client may not even be aware that the page originally had any code in it. The executed script may also contain SQL code that is executed against a database. Many of these languages come with libraries and tools that together constitute a framework for web application development.

```
<html>
<head> <title> Hello </title> </head>
<body>
<% if (request.getParameter("name") == null)
{ out.println("Hello World"); }
else { out.println("Hello, " + request.getParameter("name")); }
%>
</body>
</html>
```

Figure 9.9 A JSP page with embedded Java code.

Some of the widely used scripting frameworks include Java Server Pages (JSP), ASP.NET from Microsoft, PHP, and Ruby on Rails. These frameworks allow code written in languages such as Java, C#, VBScript, and Ruby to be embedded into or invoked from HTML pages. For instance, JSP allows Java code to be embedded in HTML pages, while Microsoft's ASP.NET and ASP support embedded C# and VBScript.

9.4.1.1 Java Server Pages

Next we briefly describe **Java Server Pages (JSP)**, a scripting language that allows HTML programmers to mix static HTML with dynamically generated HTML. The motivation is that, for many dynamic web pages, most of their content is still static (i.e., the same content is present whenever the page is generated). The dynamic content of the web pages (which are generated, for example, on the basis of form parameters) is often a small part of the page. Creating such pages by writing servlet code results in a large amount of HTML being coded as Java strings. JSP instead allows Java code to be embedded in static HTML; the embedded Java code generates the dynamic part of the page. JSP scripts are actually translated into servlet code that is then compiled, but the application programmer is saved the trouble of writing much of the Java code to create the servlet.

Figure 9.9 shows the source text of a JSP page that includes embedded Java code. The Java code in the script is distinguished from the surrounding HTML code by being enclosed in `<% ... %>`. The code uses `request.getParameter()` to get the value of the attribute `name`.

When a JSP page is requested by a browser, the application server generates HTML output from the page, which is sent to the browser. The HTML part of the JSP page is output as is.⁵ Wherever Java code is embedded within `<% ... %>`, the code is replaced in the HTML output by the text it prints to the object `out`. In the JSP code in Figure 9.9,

⁵JSP allows a more complex embedding, where HTML code is within a Java if-else statement, and gets output conditionally depending on whether the if condition evaluates to true or not. We omit details here.

if no value was entered for the form parameter name, the script prints “Hello World”; if a value was entered, the script prints “Hello” followed by the name.

A more realistic example may perform more complex actions, such as looking up values from a database using JDBC.

JSP also supports the concept of a *tag library*, which allows the use of tags that look much like HTML tags but are interpreted at the server and are replaced by appropriately generated HTML code. JSP provides a standard set of tags that define variables and control flow (iterators, if-then-else), along with an expression language based on JavaScript (but interpreted at the server). The set of tags is extensible, and a number of tag libraries have been implemented. For example, there is a tag library that supports paginated display of large data sets and a library that simplifies display and parsing of dates and times.

9.4.1.2 PHP

PHP is a scripting language that is widely used for server-side scripting. PHP code can be intermixed with HTML in a manner similar to JSP. The characters “<?php” indicate the start of PHP code, while the characters “?>” indicate the end of PHP code. The following code performs the same actions as the JSP code in Figure 9.9.

```
<html>
<head> <title> Hello </title> </head>
<body>
<?php if (!isset($_REQUEST['name'])) {
    { echo 'Hello World'; }
    else { echo 'Hello, ' . $_REQUEST['name']; }
?
</body>
</html>
```

The array `$_REQUEST` contains the request parameters. Note that the array is indexed by the parameter name; in PHP arrays can be indexed by arbitrary strings, not just numbers. The function `isset` checks if the element of the array has been initialized. The `echo` function prints its argument to the output HTML. The operator “.” between two strings concatenates the strings.

A suitably configured web server would interpret any file whose name ends in “.php” to be a PHP file. If the file is requested, the web server processes it in a manner similar to how JSP files are processed and returns the generated HTML to the browser.

A number of libraries are available for the PHP language, including libraries for database access using ODBC (similar to JDBC in Java).

9.4.2 Web Application Frameworks

Web application development frameworks ease the task of constructing web applications by providing features such as these:

- A library of functions to support HTML and HTTP features, including sessions.
- A template scripting system.
- A controller that maps user interaction events such as form submits to appropriate functions that handle the event. The controller also manages authentication and sessions. Some frameworks also provide tools for managing authorizations.
- A (relatively) declarative way of specifying a form with validation constraints on user inputs, from which the system generates HTML and Javascript/Ajax code to implement the form.
- An object-oriented model with an object-relational mapping to store data in a relational database (described in Section 9.6.2).

Thus, these frameworks provide a variety of features that are required to build web applications in an integrated manner. By generating forms from declarative specifications and managing data access transparently, the frameworks minimize the amount of coding that a web application programmer has to carry out.

There are a large number of such frameworks, based on different languages. Some of the more widely used frameworks include the Django framework for the Python language, Ruby on Rails, which supports the Rails framework on the Ruby programming language, Apache Struts, Swing, Tapestry, and WebObjects, all based on Java/JSP. Many of these frameworks also make it easy to create simple **CRUD** web interfaces; that is, interfaces that support create, read, update and delete of objects/tuples by generating code from an object model or a database. Such tools are particularly useful to get simple applications running quickly, and the generated code can be edited to build more sophisticated web interfaces.

9.4.3 The Django Framework

The Django framework for Python is a widely used web application framework. We illustrate a few features of the framework through examples.

Views in Django are functions that are equivalent to servlets in Java. Django requires a mapping, typically specified in a file `urls.py`, which maps URLs to Django views. When the Django application server receives an HTTP request, it uses the URL mapping to decide which view function to invoke.

Figure 9.10 shows sample code implementing the person query task that we earlier implemented using Java servlets. The code shows a view called `person_query_view`. We assume that the `PersonQuery` URL is mapped to the view `person_query_view`, and is invoked from the HTML form shown earlier in Figure 9.3.

We also assume that the root of the application is mapped to a `login_view`. We have not shown the code for `login_view`, but we assume it displays a login form, and on submit it invokes the `authenticate` view. We have not shown the `authenticate` view,

either, but we assume that it checks the login name and password. If the password is validated, the `authenticate` view redirects to a `person_query_form`, which displays the HTML code that we saw earlier in Figure 9.3; if password validation fails, it redirects to the `login_view`.

Returning to Figure 9.10, the view `person_query_view()` first checks if the user is logged in by checking the session variable `username`. If the session variable is not set, the browser is redirected to the login screen. Otherwise, the requested user informa-

```
from django.http import HttpResponseRedirect
from django.db import connection

def result_set_to_html(headers, cursor):
    html = "<table border=1>"
    html += "<tr>"
    for header in headers:
        html += "<th>" + header + "</th>"
    html += "</tr>"
    for row in cursor.fetchall():
        html += "<tr>"
        for col in row:
            html += "<td>" + col + "</td>"
        html += "</tr>"
    html += "</table>"
    return html

def person_query_view(request):
    if "username" not in request.session:
        return login_view(request)
    persontype = request.GET.get("persontype")
    personname = request.GET.get("personname")
    if persontype == "student":
        query_tmpl = "select id, name, dept_name from student where name=%s"
    else:
        query_tmpl = "select id, name, dept_name from instructor where name=%s"
    with connection.cursor() as cursor:
        cursor.execute(query_tmpl, [personname])
        headers = ["ID", "Name", "Department Name"]
        return HttpResponseRedirect(result_set_to_html(headers, cursor))
```

Figure 9.10 The person query application in Django.

tion is fetched by connecting to the database; connection details for the database are specified in a Django configuration file `settings.py` and are omitted in our description. A cursor (similar to a JDBC statement) is opened on the connection, and the query is executed using the cursor. Note that the first argument of `cursor.execute` is the query, with parameters marked by “%s”, and the second argument is a list of values for the parameters. The result of the database query is then displayed by calling a function `result_set_to_html()`, which iterates over the result set fetched from the database and outputs the results in HTML format to a string; the string is then returned as an `HttpResponse`.

Django provides support for a number of other features, such as creating HTML forms and validating data entered in the forms, annotations to simplify checking of authentication, and templates for creating HTML pages, which are somewhat similar to JSP pages. Django also supports an object-relation mapping system, which we describe in Section 9.6.2.2.

9.5 Client-Side Code and Web Services

The two most widely used classes of user interfaces today are the web interfaces and mobile application interfaces.

While early generation web browsers only displayed HTML code, the need was soon felt to allow code to run on the browsers. **Client-side scripting languages** are languages designed to be executed on the client’s web browser. The primary motivation for such scripting languages is flexible interaction with the user, providing features beyond the limited interaction power provided by HTML and HTML forms. Further, executing programs at the client site speeds up interaction greatly compared to every interaction being sent to a server site for processing.

The **JavaScript** language is by far the most widely used client-side scripting language. The current generation of web interfaces uses the JavaScript scripting language extensively to construct sophisticated user interfaces.

Any client-side interface needs to store and retrieve data from the back end. Directly accessing a database is not a good idea, since it not only exposes low-level details, but it also exposes the database to attacks. Instead, back ends provide access to store and retrieve data through web services. We discuss web services in Section 9.5.2.

Mobile applications are very widely used, and user interfaces for mobile devices are very important today. Although we do not cover mobile application development in this book, we offer pointers to some mobile application development frameworks in Section 9.5.4.

9.5.1 JavaScript

JavaScript is used for a variety of tasks, including validation, flexible user interfaces, and interaction with web services, which we now describe.

9.5.1.1 Input Validation

Functions written in JavaScript can be used to perform error checks (validation) on user input, such as a date string being properly formatted, or a value entered (such as age) being in an appropriate range. These checks are carried out on the browser as data are entered even before the data are sent to the web server.

With HTML5, many validation constraints can be specified as part of the input tag. For example, the following HTML code:

```
<input type="number" name="credits" size="2" min="1" max="15">
```

ensures that the input for the parameter “credits” is a number between 1 and 15. More complex validations that cannot be performed using HTML5 features are best done using JavaScript.

Figure 9.11 shows an example of a form with a JavaScript function used to validate a form input. The function is declared in the head section of the HTML document. The form accepts a start and an end date. The validation function ensures that the start date

```
<html>
<head>
<script type="text/javascript">
    function validate() {
        var startdate = new Date (document.getElementById("start").value);
        var enddate = new Date (document.getElementById("end").value);
        if(startdate > enddate) {
            alert("Start date is > end date");
            return false;
        }
    }
</script>
</head>

<body>
<form action="submitDates" onsubmit="return validate()">
    Start Date: <input type="date" id="start"><br />
    End Date : <input type="date" id="end"><br />
    <input type="submit" value="Submit">
</form>
</body>
</html>
```

Figure 9.11 Example of JavaScript used to validate form input.

is not greater than the end date. The `form` tag specifies that the validation function is to be invoked when the form is submitted. If the validation fails, an alert box is shown to the user, and if it succeeds, the form is submitted to the server.

9.5.1.2 Responsive User Interfaces

The most important benefit of JavaScript is the ability to create highly responsive user interfaces within a browser using JavaScript. The key to building such a user interface is the ability to dynamically modify the HTML code being displayed by using JavaScript. The browser parses HTML code into an in-memory tree structure defined by a standard called the **Document Object Model (DOM)**. JavaScript code can modify the tree structure to carry out certain operations. For example, suppose a user needs to enter a number of rows of data, for example multiple items in a single bill. A table containing text boxes and other form input methods can be used to gather user input. The table may have a default size, but if more rows are needed, the user may click on a button labeled (for example) “Add Item.” This button can be set up to invoke a JavaScript function that modifies the DOM tree by adding an extra row in the table.

Although the JavaScript language has been standardized, there are differences between browsers, particularly in the details of the DOM model. As a result, JavaScript code that works on one browser may not work on another. To avoid such problems, it is best to use a JavaScript library, such as the JQuery library, which allows code to be written in a browser-independent way. Internally, the functions in the library can find out which browser is in use and send appropriately generated JavaScript to the browser.

JavaScript libraries such as JQuery provide a number of UI elements, such as menus, tabs, widgets such as sliders, and features such as autocomplete, that can be created and executed using library functions.

The HTML5 standard supports a number of features for rich user interaction, including drag-and-drop, geolocation (which allows the user’s location to be provided to the application with user permission), allowing customization of the data/interface based on location. HTML5 also supports Server-Side Events (SSE), which allows a back-end to notify the front end when some event occurs.

9.5.1.3 Interfacing with Web Services

Today, JavaScript is widely used to create dynamic web pages, using several technologies that are collectively called **Ajax**. Programs written in JavaScript can communicate with the web server asynchronously (that is, in the background, without blocking user interaction with the web browser), and can fetch data and display it. The *JavaScript Object Notation*, or JSON, representation described in Section 8.1.2 is the most widely used data format for transferring data, although other formats such as XML are also used.

The role of the code for the above tasks, which runs at the application server, is to send data to the JavaScript code, which then renders the data on the browser.

Such backend services, which serve the role of functions which can be invoked to fetch required data, are known as *web services*. Such services can be implemented using Java Servlets, Python, or any of a number of other language frameworks.

As an example of the use of Ajax, consider the autocomplete feature implemented by many web applications. As the user types a value in a text box, the system suggests completions for the value being typed. Such autocomplete is very useful for helping a user choose a value from a large number of values where a drop-down list would not be feasible. Libraries such as jQuery provide support for autocomplete by associating a function with a text box; the function takes partial input in the box, connected to a web back end to get possible completions, and displays them as suggestions for the autocomplete.

The JavaScript code shown in Figure 9.12 uses the jQuery library to implement autocomplete and the DataTables plug-in for the jQuery library to provide a tabular display of data. The HTML code has a text input box for name, which has an id attribute set to name. The script associates an autocomplete function from the jQuery library with the text box by using `$("#name")` syntax of jQuery to locate the DOM node for text box with id “name”, and then associating the autocomplete function with the node. The attribute source passed to the function identifies the web service that must be invoked to get values for the autocomplete functionality. We assume that a servlet `/autocomplete_name` has been defined, which accepts a parameter `term` containing the letters typed so far by the user, even as they are being typed. The servlet should return a JSON array of names of students/instructors that match the letters in the `term` parameter.

The JavaScript code also illustrates how data can be retrieved from a web service and then displayed. Our sample code uses the DataTables jQuery plug-in; there are a number of other alternative libraries for displaying tabular data. We assume that the `person_query_ajax` Servlet, which is not shown, returns the ID, name, and department name of students or instructors with a given name, as we saw earlier in Figure 9.7, but encoded in JSON as an object with attribute `data` containing an array of rows; each row is a JSON object with attributes `id`, `name`, and `dept_name`.

The line starting with `myTable` shows how the jQuery plug-in `DataTable` is associated with the HTML table shown later in the figure, whose identifier is `personTable`. When the button “Show details” is clicked, the function `loadTableAsync()` is invoked. This function first creates a URL string `url` that is used to invoke `person_query_ajax` with values for person type and name. The function `ajax.url(url).load()` invoked on `myTable` fills the rows of the table using the JSON data fetched from the web service whose URL we created above. This happens asynchronously; that is, the function returns immediately, but when the data have been fetched, the table rows are filled with the returned data.

Figure 9.13 shows a screenshot of a browser displaying the result of the code in Figure 9.12.

As another example of the use of Ajax, consider a web site with a form that allows you to select a country, and once a country has been selected, you are allowed to select

```
<html> <head>
<script src="https://code.jquery.com/jquery-3.3.1.js"> </script>
<script src="https://cdn.datatables.net/1.10.19/js/jquery.dataTables.min.js"></script>
<script src="https://code.jquery.com/ui/1.12.1/jquery-ui.min.js"></script>
<script src="https://cdn.datatables.net/1.10.19/js/jquery.dataTables.min.js"></script>
<link rel="stylesheet"
      href="https://code.jquery.com/ui/1.12.1/themes/base/jquery-ui.css" />
<link rel="stylesheet"
      href="https://cdn.datatables.net/1.10.19/css/jquery.dataTables.min.css"/>
<script>
    var myTable;
    $(document).ready(function() {
        $("#name").autocomplete({ source: "/autocomplete_name" });
        myTable = $("#personTable").DataTable({
            columns: [{data:"id"}, {data:"name"}, {data:"dept_name"}]
        });
    });
    function loadTableAsync() {
        var params = {persontype:$("#persontype").val(), name:$("name").val()};
        var url = "/person_query_ajax?" + jQuery.param(params);
        myTable.ajax.url(url).load();
    }
</script>
</head> <body>
Search for:
<select id="persontype">
    <option value="student" selected>Student </option>
    <option value="instructor"> Instructor </option>
</select> <br>
Name: <input type="text" size=20 id="name">
<button onclick="loadTableAsync()"> Show details </button>
<table id="personTable" border="1">
    <thead>
        <tr> <th>ID</th> <th>Name</th> <th>Dept. Name</th> </tr>
    </thead>
</table>
</body> </html>
```

Figure 9.12 HTML page using JavaScript and Ajax.

a state from a list of states in that country. Until the country is selected, the drop-down list of states is empty. The Ajax framework allows the list of states to be downloaded

The screenshot shows a web application interface for searching student records. At the top left, there is a dropdown menu labeled "Search for: Student" with a downward arrow. Below it is a text input field with the prefix "Name: Br". To the right of the input field is a button labeled "Show details". A dropdown menu is open, listing "Brandt" and "Brown". To the right of the dropdown is a search bar with the placeholder "Search: []". Below these controls is a table with three columns: "ID", "Name", and "Dept. Name". The table contains three rows of data:

ID	Name	Dept. Name
76543	Brown	Comp. Sci.
77777	Brown	Biology
88888	Brown	Finance

Below the table, a message says "Showing 1 to 3 of 3 entries". At the bottom left is a "Previous" link, at the bottom center is a page number "1" in a highlighted box, and at the bottom right is a "Next" link.

Figure 9.13 Screenshot of display generated by Figure 9.12.

from the web site in the background when the country is selected, and as soon as the list has been fetched, it is added to the drop-down list, which allows you to select the state.

9.5.2 Web Services

A **web service** is an application component that can be invoked over the web and functions, in effect, like an application programming interface. A web service request is sent using the HTTP protocol, it is executed at an application server, and the results are sent back to the calling program.

Two approaches are widely used to implement web services. In the simpler approach, called **Representation State Transfer** (or **REST**), web service function calls are executed by a standard HTTP request to a URL at an application server, with parameters sent as standard HTTP request parameters. The application server executes the request (which may involve updating the database at the server), generates and encodes the result, and returns the result as the result of the HTTP request. The most widely used encoding for the results today is the JSON representation, although XML, which we saw earlier in Section 8.1.3, is also used. The requestor parses the returned page to access the returned data.

In many applications of such RESTful web services (i.e., web services using REST), the requestor is JavaScript code running in a web browser; the code updates the browser screen using the result of the function call. For example, when you scroll the display on a map interface on the web, the part of the map that needs to be newly displayed may be fetched by JavaScript code using a RESTful interface and then displayed on the screen.

While some web services are not publicly documented and are used only internally by specific applications, other web services have their interfaces documented and can be used by any application. Such services may allow use without any restriction,

may require users to be logged in before accessing the service, or may require users or application developers to pay the web service provider for the privilege of using the service.

Today, a very large variety of RESTful web services are available, and most front-end applications use one or more such services to perform backend activities. For example, your web-based email system, your social media web page, or your web-based map service would almost surely be built with JavaScript code for rendering and would use backend web services to fetch data as well as to perform updates. Similarly, any mobile app that stores data at the back end almost surely uses web services to fetch data and to perform updates.

Web services are also increasingly used at the backend, to make use of functionalities provided by other backend systems. For example, web-based storage systems provide a web service API for storing and retrieving data; such services are provided by a number of providers, such as Amazon S3, Google Cloud Storage, and Microsoft Azure. They are very popular with application developers since they allow storage of very large amounts of data, and they support a very large number of operations per second, allowing scalability far beyond what a centralized database can support.

There are many more such web-service APIs. For example, text-to-speech, speech recognition, and vision web-service APIs allow developers to construct applications incorporating speech and image recognition with very little development effort.

A more complex and less frequently used approach, sometimes referred to as “Big Web Services,” uses XML encoding of parameters as well as results, has a formal definition of the web API using a special language, and uses a protocol layer built on top of the HTTP protocol.

9.5.3 Disconnected Operation

Many applications wish to support some operations even when a client is disconnected from the application server. For example, a student may wish to complete an application form even if her laptop is disconnected from the network but have it saved back when the laptop is reconnected. As another example, if an email client is built as a web application, a user may wish to compose an email even if her laptop is disconnected from the network and have it sent when it is reconnected. Building such applications requires local storage in the client machine.

The HTML5 standard supports local storage, which can be accessed using JavaScript. The code:

```
if (typeof(Storage) !== "undefined") { // browser supports local storage  
...  
}
```

checks if the browser supports local storage. If it does, the following functions can be used to store, load, or delete values for a given key.

```
localStorage.setItem(key, value)
localStorage.getItem(key)
localStorage.removeItem(key)
```

To avoid excessive data storage, the browser may limit a web site to storing at most some amount of data; the default maximum is typically 5 megabytes.

The above interface only allows storage/retrieval of key/value pairs. Retrieval requires that a key be provided; otherwise the entire set of key/value pairs will need to be scanned to find a required value. Applications may need to store tuples indexed on multiple attributes, allowing efficient access based on values of any of the attributes. HTML5 supports IndexedDB, which allows storage of JSON objects with indices on multiple attributes. IndexedDB also supports schema versions and allows the developer to provide code to migrate data from one schema version to the next version.

9.5.4 Mobile Application Platforms

Mobile applications (or mobile apps, for short) are widely used today, and they form the primary user interface for a large class of users. The two most widely used mobile platforms today are Android and iOS. Each of these platforms provides a way of building applications with a graphical user interface, tailored to small touch-screen devices. The graphical user interface provides a variety of standard GUI features such as menus, lists, buttons, check boxes, progress bars, and so on, and the ability to display text, images, and video.

Mobile apps can be downloaded and stored and used later. Thus, the user can download apps when connected to a high-speed network and then use the app with a lower-speed network. In contrast, web apps may get downloaded when they are used, resulting in a lot of data transfer when a user may be connected to a lower-speed network or a network where data transfer is expensive. Further, mobile apps can be better tuned to small-sized devices than web apps, with user interfaces that work well on small devices. Mobile apps can also be compiled to machine code, resulting in lower power demands than web apps. More importantly, unlike (earlier generation) web apps, mobile apps can store data locally, allowing offline usage. Further, mobile apps have a well-developed authorization model, allowing them to use information and device features such as location, cameras, contacts, and so on with user authorization.

However, one of the drawbacks of using mobile-app interfaces is that code written for the Android platform can only run on that platform and not on iOS, and vice versa. As a result, developers are forced to code every application twice, once for Android and once for iOS, unless they decide to ignore one of the platforms completely, which is not very desirable.

The ability to create applications where the same high-level code can run on either Android or iOS is clearly very important. The *React Native* framework based on JavaScript, developed by Facebook, and the *Flutter* framework based on the *Dart* language developed by Google, are designed to allow cross-platform development. (*Dart*

is a language optimized for developing user interfaces, providing features such as asynchronous function invocation and functions on streams.) Both frameworks allow much of the application code to be common for both Android and iOS, but some functionality can be made specific to the underlying platform in case it is not supported in the platform-independent part of the framework.

With the wide availability of high-speed mobile networks, some of the motivation for using mobile apps instead of web apps, such as the ability to download ahead of time, is not as important anymore. A new generation of web apps, called **Progressive Web Apps (PWA)** that combine the benefits of mobile apps with web apps is seeing increasing usage. Such apps are built using JavaScript and HTML5 and are tailored for mobile devices.

A key enabling feature for PWAs is the HTML5 support for local data storage, which allows apps to be used even when the device is offline. Another enabling feature is the support for compilation of JavaScript code; compilation is restricted to code that follows a restricted syntax, since compilation of arbitrary JavaScript code is not practical. Such compilation is typically done just-in-time, that is, it is done when the code needs to be executed, or if it has already been executed multiple times. Thus, by writing CPU-heavy parts of a web application using only JavaScript features that allow compilation, it is possible to ensure CPU and energy-efficient execution of the code on a mobile device.

PWAs also make use of HTML5 service workers, which allow a script to run in the background in the browser, separate from a web page. Such service workers can be used to perform background synchronization operations between the local store and a web service, or to receive or push notifications from a backend service. HTML5 also allows apps to get device location (after user authorization), allowing PWAs to use location information.

Thus, PWAs are likely to see increasing use, replacing many (but certainly not all) of the use cases for mobile apps.

9.6 Application Architectures

To handle their complexity, large applications are often broken into several layers:

- The *presentation* or *user-interface* layer, which deals with user interaction. A single application may have several different versions of this layer, corresponding to distinct kinds of interfaces such as web browsers and user interfaces of mobile phones, which have much smaller screens.

In many implementations, the presentation/user-interface layer is itself conceptually broken up into layers, based on the **model-view-controller** (MVC) architecture.

The **model** corresponds to the business-logic layer, described below. The **view** defines the presentation of data; a single underlying model can have different views depending on the specific software/device used to access the application. The **controller** receives events (user actions), executes actions on the model, and returns

a view to the user. The MVC architecture is used in a number of web application frameworks.

- The **business-logic** layer, which provides a high-level view of data and actions on data. We discuss the business-logic layer in more detail in Section 9.6.1.
- The **data-access** layer, which provides the interface between the business-logic layer and the underlying database. Many applications use an object-oriented language to code the business-logic layer and use an object-oriented model of data, while the underlying database is a relational database. In such cases, the data-access layer also provides the mapping from the object-oriented data model used by the business logic to the relational model supported by the database. We discuss such mappings in more detail in Section 9.6.2.

Figure 9.14 shows these layers, along with a sequence of steps taken to process a request from the web browser. The labels on the arrows in the figure indicate the order of the steps. When the request is received by the application server, the controller sends a request to the model. The model processes the request, using business logic, which may involve updating objects that are part of the model, followed by creating a result object. The model in turn uses the data-access layer to update or retrieve information from a database. The result object created by the model is sent to the view module, which creates an HTML view of the result to be displayed on the web browser. The view may be tailored based on the characteristics of the device used to view the result—for example, whether it is a computer monitor with a large screen or a small screen on a phone. Increasingly, the view layer is implemented by code running at the client, instead of at the server.

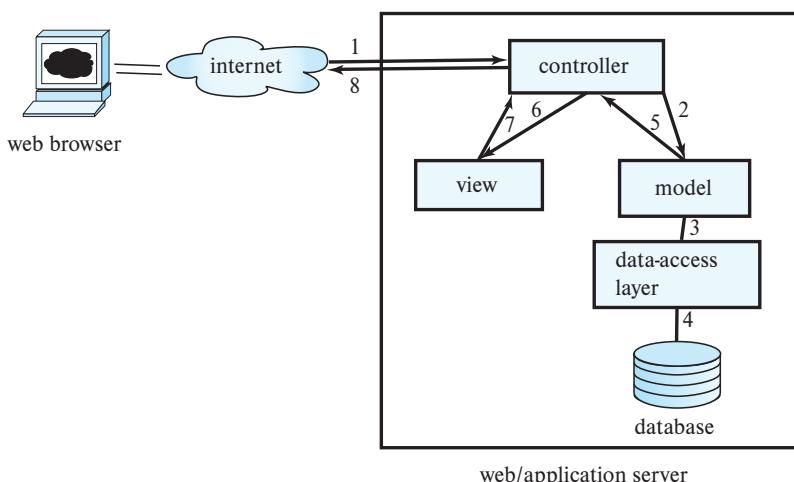


Figure 9.14 Web application architecture.

9.6.1 The Business-Logic Layer

The business-logic layer of an application for managing a university may provide abstractions of entities such as students, instructors, courses, sections, etc., and actions such as admitting a student to the university, enrolling a student in a course, and so on. The code implementing these actions ensures that **business rules** are satisfied; for example, the code would ensure that a student can enroll for a course only if she has already completed course prerequisites and has paid her tuition fees.

In addition, the business logic includes **workflows**, which describe how a particular task that involves multiple participants is handled. For example, if a candidate applies to the university, there is a workflow that defines who should see and approve the application first, and if approved in the first step, who should see the application next, and so on until either an offer is made to the student, or a rejection note is sent out. Workflow management also needs to deal with error situations; for example, if a deadline for approval/rejection is not met, a supervisor may need to be informed so she can intervene and ensure the application is processed.

9.6.2 The Data-Access Layer and Object-Relational Mapping

In the simplest scenario, where the business-logic layer uses the same data model as the database, the data-access layer simply hides the details of interfacing with the database. However, when the business-logic layer is written using an object-oriented programming language, it is natural to model data as objects, with methods invoked on objects.

In early implementations, programmers had to write code for creating objects by fetching data from the database and for storing updated objects back in the database. However, such manual conversions between data models is cumbersome and error prone. One approach to handling this problem was to develop a database system that natively stores objects, and relationships between objects, and allows objects in the database to be accessed in exactly the same way as in-memory objects. Such databases, called **object-oriented databases**, were discussed in Section 8.2. However, object-oriented databases did not achieve commercial success for a variety of technical and commercial reasons.

An alternative approach is to use traditional relational databases to store data, but to automate the mapping of data in relation to in-memory objects, which are created on demand (since memory is usually not sufficient to store all data in the database), as well as the reverse mapping to store updated objects back as relations in the database.

Several systems have been developed to implement such **object-relational mappings**. We describe the Hibernate and Django ORMs next.

9.6.2.1 Hibernate ORM

The **Hibernate** system is widely used for mapping from Java objects to relations. Hibernate provides an implementation of the Java Persistence API (JPA). In Hibernate, the mapping from each Java class to one or more relations is specified in a mapping file.

The mapping file can specify, for example, that a Java class called `Student` is mapped to the relation `student`, with the Java attribute `ID` mapped to the attribute `student.ID`, and so on. Information about the database, such as the host on which it is running and user name and password for connecting to the database, are specified in a *properties* file. The program has to open a *session*, which sets up the connection to the database. Once the session is set up, a `Student` object `stud` created in Java can be stored in the database by invoking `session.save(stud)`. The Hibernate code generates the SQL commands required to store corresponding data in the `student` relation.

While entities in an E-R model naturally correspond to objects in an object-oriented language such as Java, relationships often do not. Hibernate supports the ability to map such relationships as sets associated with objects. For example, the *takes* relationship between `student` and `section` can be modeled by associating a set of `sections` with each `student`, and a set of `students` with each `section`. Once the appropriate mapping is specified, Hibernate populates these sets automatically from the database relation *takes*, and updates to the sets are reflected back to the database relation on commit.

As an example of the use of Hibernate, we create a Java class corresponding to the `student` relation as follows:

```
@Entity public class Student {
    @Id String ID;
    String name;
    String department;
    int tot_cred;
}
```

To be precise, the class attributes should be declared as private, and getter/setter methods should be provided to access the attributes, but we omit these details.

The mapping of the class attributes of `Student` to attributes of the relation `student` can be specified in a mapping file, in an XML format, or more conveniently, by means of annotations of the Java code. In the example above, the annotation `@Entity` denotes that the class is mapped to a database relation, whose name by default is the class name, and whose attributes are by default the same as the class attributes. The default relation name and attribute names can be overridden using `@Table` and `@Column` annotations. The `@Id` annotation in the example specifies that `ID` is the primary key attribute.

The following code snippet then creates a `Student` object and saves it to the database.

```
Session session = getSessionFactory().openSession();
Transaction txn = session.beginTransaction();
Student stud = new Student("12328", "John Smith", "Comp. Sci.", 0);
session.save(stud);
txn.commit();
session.close();
```

Hibernate automatically generates the required SQL `insert` statement to create a *student* tuple in the database.

Objects can be retrieved either by primary key or by a query, as illustrated in the following code snippet:

```
Session session = getSessionFactory().openSession();
Transaction txn = session.beginTransaction();
// Retrieve student object by identifier
Student stud1 = session.get(Student.class, "12328");
    .. print out the Student information ..
List students =
    session.createQuery("from Student as s order by s.ID asc").list();
for ( Iterator iter = students.iterator(); iter.hasNext(); ) {
    Student stud = (Student) iter.next();
    .. print out the Student information ..
}
txn.commit();
session.close();
```

A single object can be retrieved using the `session.get()` method by providing its class and its primary key. The retrieved object can be updated in memory; when the transaction on the ongoing Hibernate session is committed, Hibernate automatically saves the updated objects by making corresponding updates on relations in the database.

The preceding code snippet also shows a query in Hibernate's HQL query language, which is based on SQL but designed to allow objects to be used directly in the query. The HQL query is automatically translated to SQL by Hibernate and executed, and the results are converted into a list of `Student` objects. The `for` loop iterates over the objects in this list.

These features help to provide the programmer with a high-level model of data without bothering about the details of the relational storage. However, Hibernate, like other object-relational mapping systems, also allows queries to be written using SQL on the relations stored in the database; such direct database access, bypassing the object model, can be quite useful for writing complex queries.

9.6.2.2 The Django ORM

Several ORMs have been developed for the Python language. The ORM component of the Django framework is one of the most popular such ORMs, while SQLAlchemy is another popular Python ORM.

Figure 9.15 shows a model definition for `Student` and `Instructor` in Django. Observe that all of the fields of *student* and *instructor* have been defined as fields in the class `Student` and `Instructor`, with appropriate type definitions.

In addition, the relation *advisor* has been modeled here as a many-to-many relationship between `Student` and `Instructor`. The relationship is accessed by an attribute

```
from django.db import models

class student(models.Model):
    id = models.CharField(primary_key=True, max_length=5)
    name = models.CharField(max_length=20)
    dept_name = models.CharField(max_length=20)
    tot_cred = models.DecimalField(max_digits=3, decimal_places=0)

class instructor(models.Model):
    id = models.CharField(primary_key=True, max_length=5)
    name = models.CharField(max_length=20)
    dept_name = models.CharField(max_length=20)
    salary = models.DecimalField(max_digits=8, decimal_places=2)
    advisees = models.ManyToManyField(student, related_name="advisors")
```

Figure 9.15 Model definition in Django.

called `advisees` in `Instructor`, which stores a set of references to `Student` objects. The reverse relationship from `Student` to `Instructor` is created automatically, and the model specifies that the reverse relationship attribute in the `Student` class is named `advisors`; this attribute stores a set of references to `Instructor` objects.

The Django view `person_query_model` shown in Figure 9.16 illustrates how to access database objects directly from the Python language, without using SQL. The expression `Student.objects.filter()` returns all student objects that satisfy the specified filter condition; in this case, students with the given name. The student names are printed out along with the names of their advisors. The expression `Student.advisors.all()` returns a list of advisors (advisor objects) of a given student, whose names are then retrieved and returned by the `get_names()` function. The case for instructors is similar, with instructor names being printed out along with the names of their advisees.

Django provides a tool called *migrate*, which creates database relations from a given model. Models can be given version numbers. When *migrate* is invoked on a model with a new version number, while an earlier version number is already in the database, the *migrate* tool also generates SQL code for migrating the existing data from the old database schema to the new database schema. It is also possible to create Django models from existing database schemas.

9.7 Application Performance

Web sites may be accessed by millions of people from across the globe, at rates of thousands of requests per second, or even more, for the most popular sites. Ensuring

```

from models import Student, Instructor
def get_names(persons):
    res = ""
    for p in persons:
        res += p.name + ", "
    return res.rstrip(", ")

def person_query_model(request):
    persontype = request.GET.get('persontype')
    personname = request.GET.get('personname')
    html = ""

    if persontype == 'student':
        students = Student.objects.filter(name=personname)
        for student in students:
            advisors = student.advisors.all()
            html = html + "Advisee: " + student.name + "<br>Advisors: "
            + get_names(advisors) + "<br> \n"
    else:
        instructors = Instructor.objects.filter(name=personname)
        for instructor in instructors:
            advisees = instructor.advisees.all()
            html = html + "Advisor: " + instructor.name + "<br>Advisees: "
            + get_names(advisees) + "<br> \n"
    return HttpResponseRedirect(html)

```

Figure 9.16 View definition in Django using models.

that requests are served with low response times is a major challenge for web-site developers. To do so, application developers try to speed up the processing of individual requests by using techniques such as caching, and they exploit parallel processing by using multiple application servers. We describe these techniques briefly next. Tuning of database applications is another way to improve performance and is described in Section 25.1.

9.7.1 Reducing Overhead by Caching

Suppose that the application code for servicing each user request connects to a database through JDBC. Creating a new JDBC connection may take several milliseconds, so opening a new connection for each user request is not a good idea if very high transaction rates are to be supported.

The **connection pooling** method is used to reduce this overhead; it works as follows: The connection pool manager (typically a part of the application server) creates a pool (that is, a set) of open ODBC/JDBC connections. Instead of opening a new connection to the database, the code servicing a user request (typically a servlet) asks for (requests) a connection from the connection pool and returns the connection to the pool when the code (servlet) completes its processing. If the pool has no unused connections when a connection is requested, a new connection is opened to the database (taking care not to exceed the maximum number of connections that the database system can support concurrently). If there are many open connections that have not been used for a while, the connection pool manager may close some of the open database connections. Many application servers and newer ODBC/JDBC drivers provide a built-in connection pool manager.

Details of how to create a connection pool vary by application server or JDBC driver, but most implementations require the creation of a **DataSource** object using the JDBC connection details such as the machine, port, database, user-id and password, as well as other parameters related to connection pooling. The `getConnection()` method invoked on the **DataSource** object gets a connection from the connection pool. Closing the connection returns the connection to the pool.

Certain requests may result in exactly the same query being resubmitted to the database. The cost of communication with the database can be greatly reduced by caching the results of earlier queries and reusing them, so long as the query result has not changed at the database. Some web servers support such query-result caching; caching can otherwise be done explicitly in application code.

Costs can be further reduced by caching the final web page that is sent in response to a request. If a new request comes with exactly the same parameters as a previous request, the request does not perform any updates, and the resultant web page is in the cache, that page can be reused to avoid the cost of recomputing the page. Caching can be done at the level of fragments of web pages, which are then assembled to create complete web pages.

Cached query results and cached web pages are forms of materialized views. If the underlying database data change, the cached results must be discarded, or recomputed, or even incrementally updated, as in materialized-view maintenance (described in Section 16.5). Some database systems (such as Microsoft SQL Server) provide a way for the application server to register a query with the database and get a **notification** from the database when the result of the query changes. Such a notification mechanism can be used to ensure that query results cached at the application server are up-to-date.

There are several widely used main-memory caching systems; among the more popular ones are **memcached** and **Redis**. Both systems allow applications to store data with an associated key and retrieve data for a specified key. Thus, they act as hash-map data structures that allow data to be stored in the main memory but also provide cache eviction of infrequently used data.

For example, with memcached, data can be stored using `memcached.add(key, data)` and fetched using `memcached.fetch(key)`. Instead of issuing a database query to fetch user data with a specified key, say `key1`, from a relation `r`, an application would first check if the required data are already cached by issuing a `fetch("r:"+key1)` (here, the key is appended to the relation name, to distinguish data from different relations that may be stored in the same memcached instance). If the fetch returns null, the database query is issued, a copy of the data fetched from the database is stored in memcached, and the data are then returned to the user. If the fetch does find the requested data, it can be used without accessing the database, leading to much faster access.

A client can connect to multiple memcached instances, which may run on different machines and store/retrieve data from any of them. How to decide what data are stored on which instance is left to the client code. By partitioning the data storage across multiple machines, an application can benefit from the aggregate main memory available across all the machines.

Memcached does not support automatic invalidation of cached data, but the application can track database changes and issue updates (using `memcached.set(key, newvalue)`) or deletes (using `memcached.delete(key)`) for the key values affected by update or deletion in the database. Redis offers very similar functionality. Both memcached and Redis provide APIs in multiple languages.

9.7.2 Parallel Processing

A commonly used approach to handling such very heavy loads is to use a large number of application servers running in parallel, each handling a fraction of the requests. A web server or a network router can be used to route each client request to one of the application servers. All requests from a particular client session must go to the same application server, since the server maintains state for a client session. This property can be ensured, for example, by routing all requests from a particular IP address to the same application server. The underlying database is, however, shared by all the application servers, so users see a consistent view of the database.

While the above architecture ensures that application servers do not become bottlenecks, it cannot prevent the database from becoming a bottleneck, since there is only one database server. To avoid overloading the database, application designers often use caching techniques to reduce the number of requests to the database. In addition, parallel database systems, described in Chapter 21 through Chapter 23, are used when the database needs to handle very large amounts of data, or a very large query load. Parallel data storage systems that are accessible via web service APIs are also popular in applications that need to scale to a very large number of users.

9.8 Application Security

Application security has to deal with several security threats and issues beyond those handled by SQL authorization.

The first point where security has to be enforced is in the application. To do so, applications must authenticate users and ensure that users are only allowed to carry out authorized tasks.

There are many ways in which an application's security can be compromised, even if the database system is itself secure, due to badly written application code. In this section, we first describe several security loopholes that can permit hackers to carry out actions that bypass the authentication and authorization checks carried out by the application, and we explain how to prevent such loopholes. Later in the section, we describe techniques for secure authentication, and for fine-grained authorization. We then describe audit trails that can help in recovering from unauthorized access and from erroneous updates. We conclude the section by describing issues in data privacy.

9.8.1 SQL Injection

In **SQL injection** attacks, the attacker manages to get an application to execute an SQL query created by the attacker. In Section 5.1.1.5, we saw an example of an SQL injection vulnerability if user inputs are concatenated directly with an SQL query and submitted to the database. As another example of SQL injection vulnerability, consider the form source text shown in Figure 9.3. Suppose the corresponding servlet shown in Figure 9.7 creates an SQL query string using the following Java expression:

```
String query = "select * from student where name like '%"
                + name + '%' "
```

where `name` is a variable containing the string input by the user, and then executes the query on the database. A malicious attacker using the web form can then type a string such as “`'; <some SQL statement>; --`”, where `<some SQL statement>` denotes any SQL statement that the attacker desires, in place of a valid student name. The servlet would then execute the following string.

```
select * from student where name like '%'; <some SQL statement>; -- %'
```

The quote inserted by the attacker closes the string, the following semicolon terminates the query, and the following text inserted by the attacker gets interpreted as a second SQL query, while the closing quote has been commented out. Thus, the malicious user has managed to insert an arbitrary SQL statement that is executed by the application. The statement can cause significant damage, since it can perform any action on the database, bypassing all security measures implemented in the application code.

As discussed in Section 5.1.1.5, to avoid such attacks, it is best to use prepared statements to execute SQL queries. When setting a parameter of a prepared query, JDBC automatically adds escape characters so that the user-supplied quote would no longer be able to terminate the string. Equivalently, a function that adds such escape

characters could be applied on input strings before they are concatenated with the SQL query, instead of using prepared statements.

Another source of SQL-injection risk comes from applications that create queries dynamically, based on selection conditions and ordering attributes specified in a form. For example, an application may allow a user to specify what attribute should be used for sorting the results of a query. An appropriate SQL query is constructed, based on the attribute specified. Suppose the application takes the attribute name from a form, in the variable `orderAttribute`, and creates a query string such as the following:

```
String query = "select * from takes order by " + orderAttribute;
```

A malicious user can send an arbitrary string in place of a meaningful `orderAttribute` value, even if the HTML form used to get the input tried to restrict the allowed values by providing a menu. To avoid this kind of SQL injection, the application should ensure that the `orderAttribute` variable value is one of the allowed values (in our example, attribute names) before appending it.

9.8.2 Cross-Site Scripting and Request Forgery

A web site that allows users to enter text, such as a comment or a name, and then stores it and later displays it to other users, is potentially vulnerable to a kind of attack called a **cross-site scripting (XSS)** attack. In such an attack, a malicious user enters code written in a client-side scripting language such as JavaScript or Flash instead of entering a valid name or comment. When a different user views the entered text, the browser executes the script, which can carry out actions such as sending private cookie information back to the malicious user or even executing an action on a different web server that the user may be logged into.

For example, suppose the user happens to be logged into her bank account at the time the script executes. The script could send cookie information related to the bank account login back to the malicious user, who could use the information to connect to the bank's web server, fooling it into believing that the connection is from the original user. Or the script could access appropriate pages on the bank's web site, with appropriately set parameters, to execute a money transfer. In fact, this particular problem can occur even without scripting by simply using a line of code such as

```
<img src=
  "https://mybank.com/transfermoney?amount=1000&toaccount=14523">
```

assuming that the URL `mybank.com/transfermoney` accepts the specified parameters and carries out a money transfer. This latter kind of vulnerability is also called **cross-site request forgery** or **XSRF** (sometimes also called **CSRF**).

XSS can be done in other ways, such as luring a user into visiting a web site that has malicious scripts embedded in its pages. There are other more complex kinds of XSS

and XSSR attacks, which we shall not get into here. To protect against such attacks, two things need to be done:

- **Prevent your web site from being used to launch XSS or XSSR attacks.**

The simplest technique is to disallow any HTML tags whatsoever in text input by users. There are functions that detect or strip all such tags. These functions can be used to prevent HTML tags, and as a result, any scripts, from being displayed to other users. In some cases HTML formatting is useful, and in that case functions that parse the text and allow limited HTML constructs but disallow other dangerous constructs can be used instead; these must be designed carefully, since something as innocuous as an image include could potentially be dangerous in case there is a bug in the image display software that can be exploited.

- **Protect your web site from XSS or XSSR attacks launched from other sites.**

If the user has logged into your web site and visits a different web site vulnerable to XSS, the malicious code executing on the user's browser could execute actions on your web site or pass session information related to your web site back to the malicious user, who could try to exploit it. This cannot be prevented altogether, but you can take a few steps to minimize the risk.

- The HTTP protocol allows a server to check the **referer** of a page access, that is, the URL of the page that had the link that the user clicked on to initiate the page access. By checking that the referer is valid, for example, that the referer URL is a page on the same web site, XSS attacks that originated on a different web page accessed by the user can be prevented.
- Instead of using only the cookie to identify a session, the session could also be restricted to the IP address from which it was originally authenticated. As a result, even if a malicious user gets a cookie, he may not be able to log in from a different computer.
- Never use a GET method to perform any updates. This prevents attacks using `` such as the one we saw earlier. In fact, the HTTP standard specifies that GET methods should not perform any updates.
- If you use a web application framework like Django, make sure to use the XSSR/CSRF protection mechanisms provided by the framework.

9.8.3 Password Leakage

Another problem that application developers must deal with is storing passwords in clear text in the application code. For example, programs such as JSP scripts often contain passwords in clear text. If such scripts are stored in a directory accessible by a web server, an external user may be able to access the source code of the script and get access to the password for the database account used by the application. To avoid such problems, many application servers provide mechanisms to store passwords in

encrypted form, which the server decrypts before passing it on to the database. Such a feature removes the need for storing passwords as clear text in application programs. However, if the decryption key is also vulnerable to being exposed, this approach is not fully effective.

As another measure against compromised database passwords, many database systems allow access to the database to be restricted to a given set of internet addresses, typically, the machines running the application servers. Attempts to connect to the database from other internet addresses are rejected. Thus, unless the malicious user is able to log into the application server, she cannot do any damage even if she gains access to the database password.

9.8.4 Application-Level Authentication

Authentication refers to the task of verifying the identity of a person/software connecting to an application. The simplest form of authentication consists of a secret password that must be presented when a user connects to the application. Unfortunately, passwords are easily compromised, for example, by guessing, or by sniffing of packets on the network if the passwords are not sent encrypted. More robust schemes are needed for critical applications, such as online bank accounts. Encryption is the basis for more robust authentication schemes. Authentication through encryption is addressed in Section 9.9.3.

Many applications use **two-factor authentication**, where two independent *factors* (i.e., pieces of information or processes) are used to identify a user. The two factors should not share a common vulnerability; for example, if a system merely required two passwords, both could be vulnerable to leakage in the same manner (by network sniffing, or by a virus on the computer used by the user, for example). While biometrics such as fingerprints or iris scanners can be used in situations where a user is physically present at the point of authentication, they are not very meaningful across a network.

Passwords are used as the first factor in most such two-factor authentication schemes. Smart cards or other encryption devices connected through the USB interface, which can be used for authentication based on encryption techniques (see Section 9.9.3), are widely used as second factors.

One-time password devices, which generate a new pseudo-random number (say) every minute are also widely used as a second factor. Each user is given one of these devices and must enter the number displayed by the device at the time of authentication, along with the password, to authenticate himself. Each device generates a different sequence of pseudo-random numbers. The application server can generate the same sequence of pseudo-random numbers as the device given to the user, stopping at the number that would be displayed at the time of authentication, and verify that the numbers match. This scheme requires that the clock in the device and at the server are synchronized reasonably closely.

Yet another second-factor approach is to send an SMS with a (randomly generated) one-time password to the user's phone (whose number is registered earlier) whenever

the user wishes to log in to the application. The user must possess a phone with that number to receive the SMS and then enter the one-time password, along with her regular password, to be authenticated.

It is worth noting that even with two-factor authentication, users may still be vulnerable to **man-in-the-middle attacks**. In such attacks, a user attempting to connect to the application is diverted to a fake web site, which accepts the password (including second factor passwords) from the user and uses it immediately to authenticate to the original application. The HTTPS protocol, described in Section 9.9.3.2, is used to authenticate the web site to the user (so the user does not connect to a fake site believing it to be the intended site). The HTTPS protocol also encrypts data and prevents man-in-the-middle attacks.

When users access multiple web sites, it is often annoying for a user to have to authenticate herself to each site separately, often with different passwords on each site. There are systems that allow the user to authenticate herself to one central authentication service, and other web sites and applications can authenticate the user through the central authentication service; the same password can then be used to access multiple sites. The LDAP protocol is widely used to implement such a central point of authentication for applications within a single organization; organizations implement an LDAP server containing user names and password information, and applications use the LDAP server to authenticate users.

In addition to authenticating users, a central authentication service can provide other services, for example, providing information about the user such as name, email, and address information, to the application. This obviates the need to enter this information separately in each application. LDAP can be used for this task, as described in Section 25.5.2. Other directory systems such Microsoft's Active Directories also provide mechanisms for authenticating users as well as for providing user information.

A **single sign-on** system further allows the user to be authenticated once, and multiple applications can then verify the user's identity through an authentication service without requiring reauthentication. In other words, once a user is logged in at one site, he does not have to enter his user name and password at other sites that use the same single sign-on service. Such single sign-on mechanisms have long been used in network authentication protocols such as Kerberos, and implementations are now available for web applications.

The **Security Assertion Markup Language (SAML)** is a protocol for exchanging authentication and authorization information between different security domains, to provide cross-organization single sign-on. For example, suppose an application needs to provide access to all students from a particular university, say Yale. The university can set up a web-based service that carries out authentication. Suppose a user connects to the application with a username such as "joe@yale.edu". The application, instead of directly authenticating a user, diverts the user to Yale University's authentication service, which authenticates the user and then tells the application who the user is and

may provide some additional information such as the category of the user (student or instructor) or other relevant information. The user's password and other authentication factors are never revealed to the application, and the user need not register explicitly with the application. However, the application must trust the university's authentication service when authenticating a user.

The **OpenID** protocol is an alternative for single sign-on across organizations, which works in a manner similar to SAML. The **OAuth** protocol is another protocol that allows users to authorize access to certain resources, via sharing of an authorization token.

9.8.5 Application-Level Authorization

Although the SQL standard supports a fairly flexible system of authorization based on roles (described in Section 4.7), the SQL authorization model plays a very limited role in managing user authorizations in a typical application. For instance, suppose you want all students to be able to see their own grades, but not the grades of anyone else. Such authorization cannot be specified in SQL for at least two reasons:

- 1. Lack of end-user information.** With the growth in the web, database accesses come primarily from web application servers. The end users typically do not have individual user identifiers on the database itself, and indeed there may only be a single user identifier in the database corresponding to all users of an application server. Thus, authorization specification in SQL cannot be used in the above scenario.

It is possible for an application server to authenticate end users and then pass the authentication information on to the database. In this section we will assume that the function `syscontext.user_id()` returns the identifier of the application user on whose behalf a query is being executed.⁶

- 2. Lack of fine-grained authorization.** Authorization must be at the level of individual tuples if we are to authorize students to see only their own grades. Such authorization is not possible in the current SQL standard, which permits authorization only on an entire relation or view, or on specified attributes of relations or views.

We could try to get around this limitation by creating for each student a view on the *takes* relation that shows only that student's grades. While this would work in principle, it would be extremely cumbersome since we would have to create one such view for every single student enrolled in the university, which is completely impractical.⁷

An alternative is to create a view of the form

⁶In Oracle, a JDBC connection using Oracle's JDBC drivers can set the end user identifier using the method `OracleConnection.setClientIdentifier(userId)`, and an SQL query can use the function `sys_context('USERENV', 'CLIENT_IDENTIFIER')` to retrieve the user identifier.

⁷Database systems are designed to manage large relations but to manage schema information such as views in a way that assumes smaller data volumes so as to enhance overall performance.

```
create view studentTakes as
select *
from takes
where takes.ID= syscontext.user_id()
```

Users are then given authorization to this view, rather than to the underlying *takes* relation. However, queries executed on behalf of students must now be written on the view *studentTakes*, rather than on the original *takes* relation, whereas queries executed on behalf of instructors may need to use a different view. The task of developing applications becomes more complex as a result.

The task of authorization is often typically carried out entirely in the application, bypassing the authorization facilities of SQL. At the application level, users are authorized to access specific interfaces, and they may further be restricted to view or update certain data items only.

While carrying out authorization in the application gives a great deal of flexibility to application developers, there are problems, too.

- The code for checking authorization becomes intermixed with the rest of the application code.
- Implementing authorization through application code, rather than specifying it declaratively in SQL, makes it hard to ensure the absence of loopholes. Because of an oversight, one of the application programs may not check for authorization, allowing unauthorized users access to confidential data.

Verifying that all application programs make all required authorization checks involves reading through all the application-server code, a formidable task in a large system. In other words, applications have a very large “surface area,” making the task of protecting the application significantly harder. And in fact, security loopholes have been found in a variety of real-life applications.

In contrast, if a database directly supported fine-grained authorization, authorization policies could be specified and enforced at the SQL level, which has a much smaller surface area. Even if some of the application interfaces inadvertently omit required authorization checks, the SQL-level authorization could prevent unauthorized actions from being executed.

Some database systems provide mechanisms for row-level authorization as we saw in Section 4.7.7. For example, the Oracle **Virtual Private Database (VPD)** allows a system administrator to associate a function with a relation; the function returns a predicate that must be added to any query that uses the relation (different functions can be defined for relations that are being updated). For example, using our syntax for retrieving application user identifiers, the function for the *takes* relation can return a predicate such as:

$$ID = \text{sys_context.user_id}()$$

This predicate is added to the **where** clause of every query that uses the *takes* relation. As a result (assuming that the application program sets the *user_id* value to the student's *ID*), each student can see only the tuples corresponding to courses that she took.

As we discussed in Section 4.7.7, a potential pitfall with adding a predicate as described above is that it may change the meaning of a query. For example, if a user wrote a query to find the average grade over all courses, she would end up getting the average of *her* grades, not all grades. Although the system would give the "right" answer for the rewritten query, that answer would not correspond to the query the user may have thought she was submitting.

PostgreSQL and Microsoft SQL Server offer row-level authorization support with similar functionality to Oracle VPD. More information on Oracle VPD and PostgreSQL and SQL Server row-level authorization may be found in their respective system manuals available online.

9.8.6 Audit Trails

An **audit trail** is a log of all changes (inserts, deletes, and updates) to the application data, along with information such as which user performed the change and when the change was performed. If application security is breached, or even if security was not breached, but some update was carried out erroneously, an audit trail can (a) help find out what happened, and who may have carried out the actions, and (b) aid in fixing the damage caused by the security breach or erroneous update.

For example, if a student's grade is found to be incorrect, the audit log can be examined to locate when and how the grade was updated, as well as to find which user carried out the updates. The university could then also use the audit trail to trace all the updates performed by this user in order to find other incorrect or fraudulent updates, and then correct them.

Audit trails can also be used to detect security breaches where a user's account is compromised and accessed by an intruder. For example, each time a user logs in, she may be informed about all updates in the audit trail that were done from that login in the recent past; if the user sees an update that she did not carry out, it is likely the account has been compromised.

It is possible to create a database-level audit trail by defining appropriate triggers on relation updates (using system-defined variables that identify the user name and time). However, many database systems provide built-in mechanisms to create audit trails that are much more convenient to use. Details of how to create audit trails vary across database systems, and you should refer to the database-system manuals for details.

Database-level audit trails are usually insufficient for applications, since they are usually unable to track who was the end user of the application. Further, updates are recorded at a low level, in terms of updates to tuples of a relation, rather than at a higher level, in terms of the business logic. Applications, therefore, usually create a

higher-level audit trail, recording, for example, what action was carried out, by whom, when, and from which IP address the request originated.

A related issue is that of protecting the audit trail itself from being modified or deleted by users who breach application security. One possible solution is to copy the audit trail to a different machine, to which the intruder would not have access, with each record in the trail copied as soon as it is generated. A more robust solution is to use blockchain techniques, which are described in Chapter 26; blockchain techniques store logs in multiple machines and use a hashing mechanism that makes it very difficult for an intruder to modify or delete data without being detected.

9.8.7 Privacy

In a world where an increasing amount of personal data are available online, people are increasingly worried about the privacy of their data. For example, most people would want their personal medical data to be kept private and not revealed publicly. However, the medical data must be made available to doctors and emergency medical technicians who treat the patient. Many countries have laws on privacy of such data that define when and to whom the data may be revealed. Violation of privacy law can result in criminal penalties in some countries. Applications that access such private data must be built carefully, keeping the privacy laws in mind.

On the other hand, aggregated private data can play an important role in many tasks such as detecting drug side effects, or in detecting the spread of epidemics. How to make such data available to researchers carrying out such tasks without compromising the privacy of individuals is an important real-world problem. As an example, suppose a hospital hides the name of the patient but provides a researcher with the date of birth and the postal code of the patient (both of which may be useful to the researcher). Just these two pieces of information can be used to uniquely identify the patient in many cases (using information from an external database), compromising his privacy. In this particular situation, one solution would be to give the year of birth but not the date of birth, along with the address, which may suffice for the researcher. This would not provide enough information to uniquely identify most individuals.⁸

As another example, web sites often collect personal data such as address, telephone, email, and credit-card information. Such information may be required to carry out a transaction such as purchasing an item from a store. However, the customer may not want the information to be made available to other organizations, or may want part of the information (such as credit-card numbers) to be erased after some period of time as a way to prevent it from falling into unauthorized hands in the event of a security breach. Many web sites allow customers to specify their privacy preferences, and those web sites must then ensure that these preferences are respected.

⁸For extremely old people, who are relatively rare, even the year of birth plus postal code may be enough to uniquely identify the individual, so a range of values, such as 90 years or older, may be provided instead of the actual age for people older than 90 years.

9.9 Encryption and Its Applications

Encryption refers to the process of transforming data into a form that is unreadable, unless the reverse process of decryption is applied. Encryption algorithms use an encryption key to perform encryption, and they require a decryption key (which could be the same as the encryption key, depending on the encryption algorithm used) to perform decryption.

The oldest uses of encryption were for transmitting messages, encrypted using a secret key known only to the sender and the intended receiver. Even if the message is intercepted by an enemy, the enemy, not knowing the key, will not be able to decrypt and understand the message. Encryption is widely used today for protecting data in transit in a variety of applications such as data transfer on the internet, and on cell-phone networks. Encryption is also used to carry out other tasks, such as authentication, as we will see in Section 9.9.3.

In the context of databases, encryption is used to store data in a secure way, so that even if the data are acquired by an unauthorized user (e.g., a laptop computer containing the data is stolen), the data will not be accessible without a decryption key.

Many databases today store sensitive customer information, such as credit-card numbers, names, fingerprints, signatures, and identification numbers such as, in the United States, social security numbers. A criminal who gets access to such data can use them for a variety of illegal activities, such as purchasing goods using a credit-card number, or even acquiring a credit card in someone else's name. Organizations such as credit-card companies use knowledge of personal information as a way of identifying who is requesting a service or goods. Leakage of such personal information allows a criminal to impersonate someone else and get access to service or goods; such impersonation is referred to as **identity theft**. Thus, applications that store such sensitive data must take great care to protect them from theft.

To reduce the chance of sensitive information being acquired by criminals, many countries and states today require by law that any database storing such sensitive information must store the information in an encrypted form. A business that does not protect its data thus could be held criminally liable in case of data theft. Thus, encryption is a critical component of any application that stores such sensitive information.

9.9.1 Encryption Techniques

There are a vast number of techniques for the encryption of data. Simple encryption techniques may not provide adequate security, since it may be easy for an unauthorized user to break the code. As an example of a weak encryption technique, consider the substitution of each character with the next character in the alphabet. Thus,

Perryridge

becomes

Qfsszsjehf

If an unauthorized user sees only “Qfsszsjehf,” she probably has insufficient information to break the code. However, if the intruder sees a large number of encrypted branch names, she could use statistical data regarding the relative frequency of characters to guess what substitution is being made (for example, *E* is the most common letter in English text, followed by *T, A, O, N, I*, and so on).

A good encryption technique has the following properties:

- It is relatively simple for authorized users to encrypt and decrypt data.
- It depends not on the secrecy of the algorithm, but rather on a parameter of the algorithm called the *encryption key*, which is used to encrypt data. In a **symmetric-key** encryption technique, the encryption key is also used to decrypt data. In contrast, in **public-key** (also known as **asymmetric-key**) encryption techniques, there are two different keys, the public key and the private key, used to encrypt and decrypt the data.
- Its decryption key is extremely difficult for an intruder to determine, even if the intruder has access to encrypted data. In the case of asymmetric-key encryption, it is extremely difficult to infer the private key even if the public key is available.

The **Advanced Encryption Standard (AES)** is a symmetric-key encryption algorithm that was adopted as an encryption standard by the U.S. government in 2000 and is now widely used. The standard is based on the **Rijndael algorithm** (named for the inventors V. Rijmen and J. Daemen). The algorithm operates on a 128-bit block of data at a time, while the key can be 128, 192, or 256 bits in length. The algorithm runs a series of steps to jumble up the bits in a data block in a way that can be reversed during decryption, and it performs an XOR operation with a 128-bit “round key” that is derived from the encryption key. A new round key is generated from the encryption key for each block of data that is encrypted. During decryption, the round keys are generated again from the encryption key and the encryption process is reversed to recover the original data. An earlier standard called the *Data Encryption Standard (DES)*, adopted in 1977, was very widely used earlier.

For any symmetric-key encryption scheme to work, authorized users must be provided with the encryption key via a secure mechanism. This requirement is a major weakness, since the scheme is no more secure than the security of the mechanism by which the encryption key is transmitted.

Public-key encryption is an alternative scheme that avoids some of the problems faced by symmetric-key encryption techniques. It is based on two keys: a *public key* and a *private key*. Each user U_i has a public key E_i and a private key D_i . All public keys are published: They can be seen by anyone. Each private key is known to only the one user to whom the key belongs. If user U_1 wants to store encrypted data, U_1 encrypts them using public key E_1 . Decryption requires the private key D_1 .

Because the encryption key for each user is public, it is possible to exchange information securely by this scheme. If user U_1 wants to share data with U_2 , U_1 encrypts

the data using E_2 , the public key of U_2 . Since only user U_2 knows how to decrypt the data, information can be transferred securely.

For public-key encryption to work, there must be a scheme for encryption such that it is infeasible (that is, extremely hard) to deduce the private key, given the public key. Such a scheme does exist and is based on these conditions:

- There is an efficient algorithm for testing whether or not a number is prime.
- No efficient algorithm is known for finding the prime factors of a number.

For purposes of this scheme, data are treated as a collection of integers. We create a public key by computing the product of two large prime numbers: P_1 and P_2 . The private key consists of the pair (P_1, P_2) . The decryption algorithm cannot be used successfully if only the product P_1P_2 is known; it needs the individual values P_1 and P_2 . Since all that is published is the product P_1P_2 , an unauthorized user would need to be able to factor P_1P_2 to steal data. By choosing P_1 and P_2 to be sufficiently large (over 100 digits), we can make the cost of factoring P_1P_2 prohibitively high (on the order of years of computation time, on even the fastest computers).

The details of public-key encryption and the mathematical justification of this technique's properties are referenced in the bibliographical notes.

Although public-key encryption by this scheme is secure, it is also computationally very expensive. A hybrid scheme widely used for secure communication is as follows: a symmetric encryption key (based, for example, on AES) is randomly generated and exchanged in a secure manner using a public-key encryption scheme, and symmetric-key encryption using that key is used on the data transmitted subsequently.

Encryption of small values, such as identifiers or names, is made complicated by the possibility of **dictionary attacks**, particularly if the encryption key is publicly available. For example, if date-of-birth fields are encrypted, an attacker trying to decrypt a particular encrypted value e could try encrypting every possible date of birth until he finds one whose encrypted value matches e . Even if the encryption key is not publicly available, statistical information about data distributions can be used to figure out what an encrypted value represents in some cases, such as age or address. For example, if the age 18 is the most common age in a database, the encrypted age value that occurs most often can be inferred to represent 18.

Dictionary attacks can be deterred by adding extra random bits to the end of the value before encryption (and removing them after decryption). Such extra bits, referred to as an **initialization vector** in AES, or as *salt* bits in other contexts, provide good protection against dictionary attack.

9.9.2 Encryption Support in Databases

Many file systems and database systems today support encryption of data. Such encryption protects the data from someone who is able to access the data but is not able to access the decryption key. In the case of file-system encryption, the data to be encrypted are usually large files and directories containing information about files.

In the context of databases, encryption can be done at several different levels. At the lowest level, the disk blocks containing database data can be encrypted, using a key available to the database-system software. When a block is retrieved from disk, it is first decrypted and then used in the usual fashion. Such disk-block-level encryption protects against attackers who can access the disk contents but do not have access to the encryption key.

At the next higher level, specified (or all) attributes of a relation can be stored in encrypted form. In this case, each attribute of a relation could have a different encryption key. Many databases today support encryption at the level of specified attributes as well as at the level of an entire relation, or all relations in a database. Encryption of specified attributes minimizes the overhead of decryption by allowing applications to encrypt only attributes that contain sensitive values such as credit-card numbers. Encryption also then needs to use extra random bits to prevent dictionary attacks, as described earlier. However, databases typically do not allow primary and foreign key attributes to be encrypted, and they do not support indexing on encrypted attributes.

A decryption key is obviously required to get access to encrypted data. A single master encryption key may be used for all the encrypted data; with attribute level encryption, different encryption keys could be used for different attributes. In this case, the decryption keys for different attributes can be stored in a file or relation (often referred to as “wallet”), which is itself encrypted using a master key.

A connection to the database that needs to access encrypted attributes must then provide the master key; unless this is provided, the connection will not be able to access encrypted data. The master key would be stored in the application program (typically on a different computer), or memorized by the database user, and provided when the user connects to the database.

Encryption at the database level has the advantage of requiring relatively low time and space overhead and does not require modification of applications. For example, if data in a laptop computer database need to be protected from theft of the computer itself, such encryption can be used. Similarly, someone who gets access to backup tapes of a database would not be able to access the data contained in the backups without knowing the decryption key.

An alternative to performing encryption in the database is to perform it *before* the data are sent to the database. The application must then encrypt the data before sending it to the database and decrypt the data when they are retrieved. This approach to data encryption requires significant modifications to be done to the application, unlike encryption performed in a database system.

9.9.3 Encryption and Authentication

Password-based authentication is used widely by operating systems as well as database systems. However, the use of passwords has some drawbacks, especially over a network. If an eavesdropper is able to “sniff” the data being sent over the network, she may be able to find the password as it is being sent across the network. Once the eavesdropper

has a user name and password, she can connect to the database, pretending to be the legitimate user.

A more secure scheme involves a **challenge-response** system. The database system sends a challenge string to the user. The user encrypts the challenge string using a secret password as encryption key and then returns the result. The database system can verify the authenticity of the user by decrypting the string with the same secret password and checking the result with the original challenge string. This scheme ensures that no passwords travel across the network.

Public-key systems can be used for encryption in challenge – response systems. The database system encrypts a challenge string using the user's public key and sends it to the user. The user decrypts the string using her private key and returns the result to the database system. The database system then checks the response. This scheme has the added benefit of not storing the secret password in the database, where it could potentially be seen by system administrators.

Storing the private key of a user on a computer (even a personal computer) has the risk that if the computer is compromised, the key may be revealed to an attacker who can then masquerade as the user. **Smart cards** provide a solution to this problem. In a smart card, the key can be stored on an embedded chip; the operating system of the smart card guarantees that the key can never be read, but it allows data to be sent to the card for encryption or decryption, using the private key.⁹

9.9.3.1 Digital Signatures

Another interesting application of public-key encryption is in **digital signatures** to verify authenticity of data; digital signatures play the electronic role of physical signatures on documents. The private key is used to “sign,” that is, encrypt, data, and the signed data can be made public. Anyone can verify the signature by decrypting the data using the public key, but no one could have generated the signed data without having the private key. (Note the reversal of the roles of the public and private keys in this scheme.) Thus, we can **authenticate** the data; that is, we can verify that the data were indeed created by the person who is supposed to have created them.

Furthermore, digital signatures also serve to ensure **nonrepudiation**. That is, in case the person who created the data later claims she did not create them (the electronic equivalent of claiming not to have signed the check), we can prove that that person must have created the data (unless her private key was leaked to others).

9.9.3.2 Digital Certificates

Authentication is, in general, a two-way process, where each of a pair of interacting entities authenticates itself to the other. Such pairwise authentication is needed even

⁹Smart cards provide other functionality too, such as the ability to store cash digitally and make payments, which is not relevant in our context.

when a client contacts a web site, to prevent a malicious site from masquerading as a legal web site. Such masquerading could be done, for example, if the network routers were compromised and data rerouted to the malicious site.

For a user to ensure that she is interacting with an authentic web site, she must have the site's public key. This raises the problem of how the user can get the public key—if it is stored on the web site, the malicious site could supply a different key, and the user would have no way of verifying if the supplied public key is itself authentic. Authentication can be handled by a system of **digital certificates**, whereby public keys are signed by a certification agency, whose public key is well known. For example, the public keys of the root certification authorities are stored in standard web browsers. A certificate issued by them can be verified by using the stored public keys.

A two-level system would place an excessive burden of creating certificates on the root certification authorities, so a multilevel system is used instead, with one or more root certification authorities and a tree of certification authorities below each root. Each authority (other than the root authorities) has a digital certificate issued by its parent.

A digital certificate issued by a certification authority A consists of a public key K_A and an encrypted text E that can be decoded by using the public key K_A . The encrypted text contains the name of the party to whom the certificate was issued and her public key K_c . In case the certification authority A is not a root certification authority, the encrypted text also contains the digital certificate issued to A by its parent certification authority; this certificate authenticates the key K_A itself. (That certificate may in turn contain a certificate from a further parent authority, and so on.)

To verify a certificate, the encrypted text E is decrypted by using the public key K_A to retrieve the name of the party (i.e., the name of the organization owning the web site); additionally, if A is not a root authority whose public key is known to the verifier, the public key K_A is verified recursively by using the digital certificate contained within E ; recursion terminates when a certificate issued by the root authority is reached. Verifying the certificate establishes the chain through which a particular site was authenticated and provides the name and authenticated public key for the site.

Digital certificates are widely used to authenticate web sites to users, to prevent malicious sites from masquerading as other web sites. In the HTTPS protocol (the secure version of the HTTP protocol), the site provides its digital certificate to the browser, which then displays it to the user. If the user accepts the certificate, the browser then uses the provided public key to encrypt data. A malicious site will have access to the certificate, but not the private key, and will thus not be able to decrypt the data sent by the browser. Only the authentic site, which has the corresponding private key, can decrypt the data sent by the browser. We note that public-/private-key encryption and decryption costs are much higher than encryption/decryption costs using symmetric private keys. To reduce encryption costs, HTTPS actually creates a one-time symmetric key after authentication and uses it to encrypt data for the rest of the session.

Digital certificates can also be used for authenticating users. The user must submit a digital certificate containing her public key to a site, which verifies that the certificate has been signed by a trusted authority. The user's public key can then be used in a challenge-response system to ensure that the user possesses the corresponding private key, thereby authenticating the user.

9.10 Summary

- Application programs that use databases as back ends and interact with users have been around since the 1960s. Application architectures have evolved over this period. Today most applications use web browsers as their front end, and a database as their back end, with an application server in between.
- HTML provides the ability to define interfaces that combine hyperlinks with forms facilities. Web browsers communicate with web servers by the HTTP protocol. Web servers can pass on requests to application programs and return the results to the browser.
- Web servers execute application programs to implement desired functionality. Servlets are a widely used mechanism to write application programs that run as part of the web server process, in order to reduce overhead. There are also many server-side scripting languages that are interpreted by the web server and provide application-program functionality as part of the web server.
- There are several client-side scripting languages—JavaScript is the most widely used—that provide richer user interaction at the browser end.
- Complex applications usually have a multilayer architecture, including a model implementing business logic, a controller, and a view mechanism to display results. They may also include a data access layer that implements an object-relational mapping. Many applications implement and use web services, allowing functions to be invoked over HTTP.
- Techniques such as caching of various forms, including query result caching and connection pooling, and parallel processing are used to improve application performance.
- Application developers must pay careful attention to security, to prevent attacks such as SQL injection attacks and cross-site scripting attacks.
- SQL authorization mechanisms are coarse grained and of limited value to applications that deal with large numbers of users. Today application programs implement fine-grained, tuple-level authorization, dealing with a large number of application users, completely outside the database system. Database extensions to provide tuple-level access control and to deal with large numbers of application users have been developed, but are not standard as yet.

- Protecting the privacy of data are an important task for database applications. Many countries have legal requirements on protection of certain kinds of data, such as credit-card information or medical data.
- Encryption plays a key role in protecting information and in authentication of users and web sites. Symmetric-key encryption and public-key encryption are two contrasting but widely used approaches to encryption. Encryption of certain sensitive data stored in databases is a legal requirement in many countries and states.
- Encryption also plays a key role in authentication of users to applications, of Web sites to users, and for digital signatures.

Review Terms

- Application programs
- Web interfaces to databases
- HTML
- Hyperlinks
- Uniform resource locator (URL)
- Forms
- HyperText Transfer Protocol (HTTP)
- Connectionless protocols
- Cookie
- Session
- Servlets and Servlet sessions
- Server-side scripting
- Java Server Pages (JSP)
- PHP
- Client-side scripting
- JavaScript
- Document Object Model (DOM)
- Ajax
- Progressive Web Apps
- Application architecture
- Presentation layer
- Model-view-controller (MVC) architecture
- Business-logic layer
- Data-access layer
- Object-relational mapping
- Hibernate
- Django
- Web services
- RESTful web services
- Web application frameworks
- Connection pooling
- Query result caching
- Application security
- SQL injection
- Cross-site scripting (XSS)
- Cross-site request forgery (XSRF)
- Authentication
- Two-factor authentication
- Man-in-the-middle attack
- Central authentication
- Single sign-on
- OpenID
- Authorization
- Virtual Private Database (VPD)
- Audit trail

- Encryption
- Symmetric-key encryption
- Public-key encryption
- Dictionary attack
- Challenge – response
- Digital signatures
- Digital certificates

Practice Exercises

- 9.1 What is the main reason why servlets give better performance than programs that use the common gateway interface (CGI), even though Java programs generally run slower than C or C++ programs?
- 9.2 List some benefits and drawbacks of connectionless protocols over protocols that maintain connections.
- 9.3 Consider a carelessly written web application for an online-shopping site, which stores the price of each item as a hidden form variable in the web page sent to the customer; when the customer submits the form, the information from the hidden form variable is used to compute the bill for the customer. What is the loophole in this scheme? (There was a real instance where the loophole was exploited by some customers of an online-shopping site before the problem was detected and fixed.)
- 9.4 Consider another carelessly written web application which uses a servlet that checks if there was an active session but does not check if the user is authorized to access that page, instead depending on the fact that a link to the page is shown only to authorized users. What is the risk with this scheme? (There was a real instance where applicants to a college admissions site could, after logging into the web site, exploit this loophole and view information they were not authorized to see; the unauthorized access was, however, detected, and those who accessed the information were punished by being denied admission.)
- 9.5 Why is it important to open JDBC connections using the try-with-resources (try (...){} ...) syntax?
- 9.6 List three ways in which caching can be used to speed up web server performance.
- 9.7 The `netstat` command (available on Linux and on Windows) shows the active network connections on a computer. Explain how this command can be used to find out if a particular web page is not closing connections that it opened, or if connection pooling is used, not returning connections to the connection pool. You should account for the fact that with connection pooling, the connection may not get closed immediately.

- 9.8** Testing for SQL-injection vulnerability:
- Suggest an approach for testing an application to find if it is vulnerable to SQL injection attacks on text input.
 - Can SQL injection occur with forms of HTML input other than text boxes? If so, how would you test for vulnerability?
- 9.9** A database relation may have the values of certain attributes encrypted for security. Why do database systems not support indexing on encrypted attributes? Using your answer to this question, explain why database systems do not allow encryption of primary-key attributes.
- 9.10** Exercise 9.9 addresses the problem of encryption of certain attributes. However, some database systems support encryption of entire databases. Explain how the problems raised in Exercise 9.9 are avoided if the entire database is encrypted.
- 9.11** Suppose someone impersonates a company and gets a certificate from a certificate-issuing authority. What is the effect on things (such as purchase orders or programs) certified by the impersonated company, and on things certified by other companies?
- 9.12** Perhaps the most important data items in any database system are the passwords that control access to the database. Suggest a scheme for the secure storage of passwords. Be sure that your scheme allows the system to test passwords supplied by users who are attempting to log into the system.

Exercises

- 9.13** Write a servlet and associated HTML code for the following very simple application: A user is allowed to submit a form containing a value, say n , and should get a response containing n “*” symbols.
- 9.14** Write a servlet and associated HTML code for the following simple application: A user is allowed to submit a form containing a number, say n , and should get a response saying how many times the value n has been submitted previously. The number of times each value has been submitted previously should be stored in a database.
- 9.15** Write a servlet that authenticates a user (based on user names and passwords stored in a database relation) and sets a session variable called *userid* after authentication.
- 9.16** What is an SQL injection attack? Explain how it works and what precautions must be taken to prevent SQL injection attacks.
- 9.17** Write pseudocode to manage a connection pool. Your pseudocode must include a function to create a pool (providing a database connection string, database user name, and password as parameters), a function to request a connection

from the pool, a connection to release a connection to the pool, and a function to close the connection pool.

- 9.18** Explain the terms CRUD and REST.
- 9.19** Many web sites today provide rich user interfaces using Ajax. List two features each of which reveals if a site uses Ajax, without having to look at the source code. Using the above features, find three sites which use Ajax; you can view the HTML source of the page to check if the site is actually using Ajax.
- 9.20** XSS attacks:
- What is an XSS attack?
 - How can the referer field be used to detect some XSS attacks?
- 9.21** What is multifactor authentication? How does it help safeguard against stolen passwords?
- 9.22** Consider the Oracle Virtual Private Database (VPD) feature described in Section 9.8.5 and an application based on our university schema.
- What predicate (using a subquery) should be generated to allow each faculty member to see only *takes* tuples corresponding to course sections that they have taught?
 - Give an SQL query such that the query with the predicate added gives a result that is a subset of the original query result without the added predicate.
 - Give an SQL query such that the query with the predicate added gives a result containing a tuple that is not in the result of the original query without the added predicate.
- 9.23** What are two advantages of encrypting data stored in the database?
- 9.24** Suppose you wish to create an audit trail of changes to the *takes* relation.
- Define triggers to create an audit trail, logging the information into a relation called, for example, *takes_trail*. The logged information should include the user-id (assume a function *user_id()* provides this information) and a timestamp, in addition to old and new values. You must also provide the schema of the *takes_trail* relation.
 - Can the preceding implementation guarantee that updates made by a malicious database administrator (or someone who manages to get the administrator's password) will be in the audit trail? Explain your answer.
- 9.25** Hackers may be able to fool you into believing that their web site is actually a web site (such as a bank or credit card web site) that you trust. This may be done by misleading email, or even by breaking into the network infrastructure

and rerouting network traffic destined for, say mybank.com, to the hacker's site. If you enter your user name and password on the hacker's site, the site can record it and use it later to break into your account at the real site. When you use a URL such as <https://mybank.com>, the HTTPS protocol is used to prevent such attacks. Explain how the protocol might use digital certificates to verify authenticity of the site.

- 9.26** Explain what is a challenge - response system for authentication. Why is it more secure than a traditional password-based system?

Project Suggestions

Each of the following is a large project, which can be a semester-long project done by a group of students. The difficulty of the project can be adjusted easily by adding or deleting features.

You can choose to use either a web front-end using HTML5, or a mobile front-end on Android or iOS for your project.

Project 9.1 Pick your favorite interactive web site, such as Bebo, Blogger, Facebook, Flickr, Last.FM, Twitter, Wikipedia; these are just a few examples, there are many more. Most of these sites manage a large amount of data and use databases to store and process the data. Implement a subset of the functionality of the web site you picked. Implementing even a significant subset of the features of such a site is well beyond a course project, but it is possible to find a set of features that is interesting to implement yet small enough for a course project.

Most of today's popular web sites make extensive use of Javascript to create rich interfaces. You may wish to go easy on this for your project, at least initially, since it takes time to build such interfaces, and then add more features to your interfaces, as time permits.

Make use of web application development frameworks, or Javascript libraries available on the web, such as the jQuery library, to speed up your front-end development. Alternatively, implement the application as a mobile app on Android or iOS.

Project 9.2 Create a "mashup" which uses web services such as Google or Yahoo map APIs to create an interactive web site. For example, the map APIs provide a way to display a map on the web page, with other information overlaid on the maps. You could implement a restaurant recommendation system, with users contributing information about restaurants such as location, cuisine, price range, and ratings. Results of user searches could be displayed on the map. You could allow Wikipedia-like features, such as allowing users to add information and edit

information added by other users, along with moderators who can weed out malicious updates. You could also implement social features, such as giving more importance to ratings provided by your friends.

Project 9.3 Your university probably uses a course-management system such as Moodle, Blackboard, or WebCT. Implement a subset of the functionality of such a course-management system. For example, you can provide assignment submission and grading functionality, including mechanisms for students and teachers/teaching assistants to discuss grading of a particular assignment. You could also provide polls and other mechanisms for getting feedback.

Project 9.4 Consider the E-R schema of Practice Exercise 6.3 (Chapter 6), which represents information about teams in a league. Design and implement a web-based system to enter, update, and view the data.

Project 9.5 Design and implement a shopping cart system that lets shoppers collect items into a shopping cart (you can decide what information is to be supplied for each item) and purchase together. You can extend and use the E-R schema of Exercise 6.21 of Chapter 6. You should check for availability of the item and deal with nonavailable items as you feel appropriate.

Project 9.6 Design and implement a web-based system to record student registration and grade information for courses at a university.

Project 9.7 Design and implement a system that permits recording of course performance information—specifically, the marks given to each student in each assignment or exam of a course, and computation of a (weighted) sum of marks to get the total course marks. The number of assignments/exams should not be predefined; that is, more assignments/exams can be added at any time. The system should also support grading, permitting cutoffs to be specified for various grades.

You may also wish to integrate it with the student registration system of Project 9.6 (perhaps being implemented by another project team).

Project 9.8 Design and implement a web-based system for booking classrooms at your university. Periodic booking (fixed days/times each week for a whole semester) must be supported. Cancellation of specific lectures in a periodic booking should also be supported.

You may also wish to integrate it with the student registration system of Project 9.6 (perhaps being implemented by another project team) so that classrooms can be booked for courses, and cancellations of a lecture or addition of extra lectures can be noted at a single interface and will be reflected in the classroom booking and communicated to students via email.

Project 9.9 Design and implement a system for managing online multiple-choice tests.

You should support distributed contribution of questions (by teaching assistants, for example), editing of questions by whoever is in charge of the course, and creation of tests from the available set of questions. You should also be able to administer tests online, either at a fixed time for all students or at any time but with a time limit from start to finish (support one or both), and the system should give students feedback on their scores at the end of the allotted time.

Project 9.10 Design and implement a system for managing email customer service.

Incoming mail goes to a common pool. There is a set of customer service agents who reply to email. If the email is part of an ongoing series of replies (tracked using the in-reply-to field of email) the mail should preferably be replied to by the same agent who replied earlier. The system should track all incoming mail and replies, so an agent can see the history of questions from a customer before replying to an email.

Project 9.11 Design and implement a simple electronic marketplace where items can be listed for sale or for purchase under various categories (which should form a hierarchy). You may also wish to support alerting services, whereby a user can register interest in items in a particular category, perhaps with other constraints as well, without publicly advertising her interest, and is notified when such an item is listed for sale.**Project 9.12** Design and implement a web-based system for managing a sports “ladder.” Many people register and may be given some initial rankings (perhaps based on past performance). Anyone can challenge anyone else to a match, and the rankings are adjusted according to the result. One simple system for adjusting rankings just moves the winner ahead of the loser in the rank order, in case the winner was behind earlier. You can try to invent more complicated rank-adjustment systems.**Project 9.13** Design and implement a publication-listing service. The service should permit entering of information about publications, such as title, authors, year, where the publication appeared, and pages. Authors should be a separate entity with attributes such as name, institution, department, email, address, and home page.

Your application should support multiple views on the same data. For instance, you should provide all publications by a given author (sorted by year, for example), or all publications by authors from a given institution or department. You should also support search by keywords, on the overall database as well as within each of the views.

Project 9.14 A common task in any organization is to collect structured information from a group of people. For example, a manager may need to ask employees to enter their vacation plans, a professor may wish to collect feedback on a particu-

lar topic from students, or a student organizing an event may wish to allow other students to register for the event, or someone may wish to conduct an online vote on some topic. Google Forms can be used for such activities; your task is to create something like Google Forms, but with authorization on who can fill a form.

Specifically, create a system that will allow users to easily create information collection events. When creating an event, the event creator must define who is eligible to participate; to do so, your system must maintain user information and allow predicates defining a subset of users. The event creator should be able to specify a set of inputs (with types, default values, and validation checks) that the users will have to provide. The event should have an associated deadline, and the system should have the ability to send reminders to users who have not yet submitted their information. The event creator may be given the option of automatic enforcement of the deadline based on a specified date/time, or choosing to login and declare the deadline is over. Statistics about the submissions should be generated—to do so, the event creator may be allowed to create simple summaries on the entered information. The event creator may choose to make some of the summaries public, viewable by all users, either continually (e.g., how many people have responded) or after the deadline (e.g., what was the average feedback score).

Project 9.15 Create a library of functions to simplify creation of web interfaces, using jQuery. You must implement at least the following functions: a function to display a JDBC result set (with tabular formatting), functions to create different types of text and numeric inputs (with validation criteria such as input type and optional range, enforced at the client by appropriate JavaScript code), and functions to create menu items based on a result set. Also implement functions to get input for specified fields of specified relations, ensuring that database constraints such as type and foreign-key constraints are enforced at the client side. Foreign key constraints can also be used to provide either autocomplete or drop-down menus, to ease the task of data entry for the fields.

For extra credit, use support CSS styles which allow the user to change style parameters such as colors and fonts. Build a sample database application to illustrate the use of these functions.

Project 9.16 Design and implement a web-based multiuser calendar system. The system must track appointments for each person, including multioccurrence events, such as weekly meetings, and shared events (where an update made by the event creator gets reflected to all those who share the event). Provide interfaces to schedule multiuser events, where an event creator can add a number of users who are invited to the event. Provide email notification of events. For extra credits implement a web service that can be used by a reminder program running on the client machine.

Tools

There are several integrated development environments that provide support for web application development. Eclipse (www.eclipse.org) and Netbeans (netbeans.org) are popular open-source IDEs. IntelliJ IDEA (www.jetbrains.com/idea/) is a popular commercial IDE which provides free licenses for students, teachers and non-commercial open source projects. Microsoft's Visual Studio (visualstudio.microsoft.com) also supports web application development. All these IDEs support integration with application servers, to allow web applications to be executed directly from the IDE.

The Apache Tomcat (jakarta.apache.org), Glassfish (javaee.github.io/glassfish/), JBoss Enterprise Application Platform (developers.redhat.com/products/eap/overview/), WildFly (wildfly.org) (which is the community edition of JBoss) and Caucho's Resin (www.caucho.com), are application servers that support servlets and JSP. The Apache web server (apache.org) is the most widely used web server today. Microsoft's IIS (Internet Information Services) is a web and application server that is widely used on Microsoft Windows platforms, supporting Microsoft's ASP.NET (msdn.microsoft.com/asp.net/).

The jQuery JavaScript library jquery.com is among the most widely used JavaScript libraries for creating interactive web interfaces.

Android Studio (developer.android.com/studio/) is a widely used IDE for developing Android apps. XCode (developer.apple.com/xcode/) from Apple and AppCode (www.jetbrains.com/objc/) are popular IDEs for iOS application development. Google's Flutter framework (flutter.io), which is based on the Dart language, and Facebook's React Native (facebook.github.io/react-native/) which is based on Javascript, are frameworks that support cross-platform application development across Android and iOS.

The Open Web Application Security Project (OWASP) (www.owasp.org) provides a variety of resources related to application security, including technical articles, guides, and tools.

Further Reading

The HTML tutorials at www.w3schools.com/html, the CSS tutorials at www.w3schools.com/css are good resources for learning HTML and CSS. A tutorial on Java Servlets can be found at docs.oracle.com/javaee/7/tutorial/servlets.htm. The JavaScript tutorials at www.w3schools.com/js are an excellent source of learning material on JavaScript. You can also learn more about JSON and Ajax as part of the JavaScript tutorial. The jQuery tutorial at www.w3schools.com/Jquery is a very good resource for learning how to use jQuery. These tutorials allow you to modify sample code and test it in the browser, with no software download. Information about the

.NET framework and about web application development using ASP.NET can be found at msdn.microsoft.com.

You can learn more about the Hibernate ORM and Django (including the Django ORM) from the tutorials and documentation at hibernate.org/orm and docs.djangoproject.com respectively.

The Open Web Application Security Project (OWASP) (www.owasp.org) provides a variety of technical material such as the OWASP Testing Guide, the OWASP Top Ten document which describes critical security risks, and standards for application security verification.

The concepts behind cryptographic hash functions and public-key encryption were introduced in [Diffie and Hellman (1976)] and [Rivest et al. (1978)]. A good reference for cryptography is [Katz and Lindell (2014)], while [Stallings (2017)] provides textbook coverage of cryptography and network security.

Bibliography

[Diffie and Hellman (1976)] W. Diffie and M. E. Hellman, “New Directions in Cryptography”, *IEEE Transactions on Information Theory*, Volume 22, Number 6 (1976), pages 644-654.

[Katz and Lindell (2014)] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 3rd edition, Chapman and Hall/CRC (2014).

[Rivest et al. (1978)] R. L. Rivest, A. Shamir, and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, *Communications of the ACM*, Volume 21, Number 2 (1978), pages 120-126.

[Stallings (2017)] W. Stallings, *Cryptography and Network Security - Principles and Practice*, 7th edition, Pearson (2017).

Credits

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

CHAPTER 9



Application Development

Practice Exercises

- 9.1** What is the main reason why servlets give better performance than programs that use the common gateway interface (CGI), even though Java programs generally run slower than C or C++ programs?

Answer:

The CGI interface starts a new process to service each request, which has a significant operating system overhead. On the other hand, servlets are run as threads of an existing process, avoiding this overhead. Further, the process running threads could be the web server process itself, avoiding interprocess communication, which can be expensive. Thus, for small to moderate-sized tasks, the overhead of Java is less than the overhead saved by avoiding process creation and communication.

For tasks involving a lot of CPU activity, this may not be the case, and using CGI with a C or C++ program may give better performance.

- 9.2** List some benefits and drawbacks of connectionless protocols over protocols that maintain connections.

Answer:

Most computers have limits on the number of simultaneous connections they can accept. With connectionless protocols, connections are broken as soon as the request is satisfied, and therefore other clients can open connections. Thus more clients can be served at the same time. A request can be routed to any one of a number of different servers to balance load, and if a server crashes, another can take over without the client noticing any problem.

The drawback of connectionless protocols is that a connection has to be reestablished every time a request is sent. Also, session information has to be sent each time in the form of cookies or hidden fields. This makes them slower than the protocols which maintain connections in case state information is required.

- 9.3** Consider a carelessly written web application for an online-shopping site, which stores the price of each item as a hidden form variable in the web page sent to the customer; when the customer submits the form, the information from the hidden form variable is used to compute the bill for the customer. What is the loophole in this scheme? (There was a real instance where the loophole was exploited by some customers of an online-shopping site before the problem was detected and fixed.)

Answer:

A hacker can edit the HTML source code of the web page and replace the value of the hidden variable price with another value, use the modified web page to place an order. The web application would then use the user-modified value as the price of the product.

- 9.4** Consider another carelessly written web application which uses a servlet that checks if there was an active session but does not check if the user is authorized to access that page, instead depending on the fact that a link to the page is shown only to authorized users. What is the risk with this scheme? (There was a real instance where applicants to a college admissions site could, after logging into the web site, exploit this loophole and view information they were not authorized to see; the unauthorized access was, however, detected, and those who accessed the information were punished by being denied admission.)

Answer:

Although the link to the page is shown only to authorized users, an unauthorized user may somehow come to know of the existence of the link (for example, from an unauthorized user, or via web proxy logs). The user may then log in to the system and access the unauthorized page by entering its URL in the browser. If the check for user authorization was inadvertently left out from that page, the user will be able to see the result of the page.

The HTTP referer attribute can be used to block a naive attempt to exploit such loopholes by ensuring the referer value is from a valid page of the web site. However, the referer attribute is set by the browser and can be spoofed, so a malicious user can easily work around the referer check.

- 9.5** Why is it important to open JDBC connections using the try-with-resources (try (...){} ...) syntax?

Answer:

This ensures connections are closed properly, and you will not run out of database connections.

- 9.6** List three ways in which caching can be used to speed up web server performance.

Answer:

Caching can be used to improve performance by exploiting the commonalities between transactions.

- a. If the application code for servicing each request needs to open a connection to the database, which is time consuming, then a pool of open connections may be created beforehand, and each request uses one from those.
 - b. The results of a query generated by a request can be cached. If the same request comes again, or generates the same query, then the cached result can be used instead of connecting to the database again.
 - c. The final web page generated in response to a request can be cached. If the same request comes again, then the cached page can be outputted.
- 9.7** The `netstat` command (available on Linux and on Windows) shows the active network connections on a computer. Explain how this command can be used to find out if a particular web page is not closing connections that it opened, or if connection pooling is used, not returning connections to the connection pool. You should account for the fact that with connection pooling, the connection may not get closed immediately.

Answer:

The tester should run `netstat` to find all connections open to the machine/socket used by the database. (If the application server is separate from the database server, the command may be executed at either of the machines). Then the web page being tested should be accessed repeatedly (this can be automated by using tools such as JMeter to generate page accesses). The number of connections to the database would go from 0 to some value (depending on the number of connections retained in the pool), but after some time the number of connections should stop increasing. If the number keeps increasing, the code underlying the web page is clearly not closing connections or returning the connection to the pool.

- 9.8** Testing for SQL-injection vulnerability:

- a. Suggest an approach for testing an application to find if it is vulnerable to SQL injection attacks on text input.
- b. Can SQL injection occur with forms of HTML input other than text boxes? If so, how would you test for vulnerability?

Answer:

- a. One approach is to enter a string containing a single quote in each of the input text boxes of each of the forms provided by the application to see

- if the application correctly saves the value. If it does not save the value correctly and/or gives an error message, it is vulnerable to SQL injection.
- b. Yes, SQL injection can even occur with selection inputs such as drop-down menus, by modifying the value sent back to the server when the input value is chosen—for example by editing the page directly, or in the browser’s DOM tree. Most modern browsers provide a way for users to edit the DOM tree. This feature can be able to modify the values sent to the application, inserting a single quote into the value.

- 9.9** A database relation may have the values of certain attributes encrypted for security. Why do database systems not support indexing on encrypted attributes? Using your answer to this question, explain why database systems do not allow encryption of primary-key attributes.

Answer:

It is not possible in general to index on an encrypted value, unless all occurrences of the value encrypt to the same value (and even in this case, only equality predicates would be supported). However, mapping all occurrences of a value to the same encrypted value is risky, since statistical analysis can be used to reveal common values, even without decryption; techniques based on adding random “salt” bits are used to prevent such analysis, but they make indexing impossible. One possible workaround is to store the index unencrypted, but then the index can be used to leak values. Another option is to keep the index encrypted, but then the database system should know the decryption key, to decrypt required parts of the index on the fly. Since this requires modifying large parts of the database system code, databases typically do not support this option.

The primary-key constraint has to be checked by the database when tuples are inserted, and if the values are encrypted as above, the database system will not be able to detect primary-key violations. Therefore, database systems that support encryption of specified attributes do not allow primary-key attributes, or for that matter foreign-key attributes, to be encrypted.

- 9.10** Exercise 9.9 addresses the problem of encryption of certain attributes. However, some database systems support encryption of entire databases. Explain how the problems raised in Exercise 9.9 are avoided if the entire database is encrypted.

Answer:

When the entire database is encrypted, it is easy for the database to perform decryption as data are fetched from disk into memory, so in-memory storage is unencrypted. With this option, everything in the database, including indices, is encrypted when on disk, but unencrypted in memory. As a result, only the data access layer of the database system code needs to be modified to perform encryption, leaving other layers untouched. Thus, indices can be used unchanged, and primary-key and foreign-key constraints enforced without any change to the corresponding layers of the database system code.

- 9.11** Suppose someone impersonates a company and gets a certificate from a certificate-issuing authority. What is the effect on things (such as purchase orders or programs) certified by the impersonated company, and on things certified by other companies?

Answer:

The key problem with digital certificates (when used offline, without contacting the certificate issuer) is that there is no way to withdraw them.

For instance (this actually happened, but names of the parties have been changed) person C claims to be an employee of company X and gets a new public key certified by the certifying authority A . Suppose the authority A incorrectly believed that C was acting on behalf of company X , and it gave C a certificate $cert$. Now C can communicate with person Y , who checks the certificate $cert$ presented by C and believes the public key contained in $cert$ really belongs to X . C can communicate with Y using the public key, and Y trusts the communication is from company X .

Person Y may now reveal confidential information to C or accept a purchase order from C or execute programs certified by C , based on the public key, thinking he is actually communicating with company X . In each case there is potential for harm to Y .

Even if A detects the impersonation, as long as Y does not check with A (the protocol does not require this check), there is no way for Y to find out that the certificate is forged.

If X was a certification authority itself, further levels of fake certificates could be created. But certificates that are not part of this chain would not be affected.

- 9.12** Perhaps the most important data items in any database system are the passwords that control access to the database. Suggest a scheme for the secure storage of passwords. Be sure that your scheme allows the system to test passwords supplied by users who are attempting to log into the system.

Answer:

A scheme for storing passwords would be to encrypt each password (after adding randomly generated “salt” bits to prevent dictionary attacks), and then use a hash index on the user-id to store/access the encrypted password. The password being used in a login attempt is then encrypted (if randomly generated “salt” bits were used initially, these bits should be stored with the user-id and used when encrypting the user-supplied password). The encrypted value is then compared with the stored encrypted value of the correct password. An advantage of this scheme is that passwords are not stored in clear text, and the code for decryption need not even exist. Thus, “one-way” encryption functions, such as secure hashing functions, which do not support decryption can be used for this task. The secure hashing algorithm SHA-1 is widely used for such one-way encryption.



Transactions

Often, a collection of several operations on the database appears to be a single unit from the point of view of the database user. For example, a transfer of funds from a checking account to a savings account is a single operation from the customer's standpoint; within the database system, however, it consists of several operations. It is essential that all these operations occur, or that, in case of a failure, none occur. It would be unacceptable if the checking account were debited but the savings account not credited.

Collections of operations that form a single logical unit of work are called **transactions**. A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency. In our funds-transfer example, a transaction computing the customer's total balance might see the checking-account balance before it is debited by the funds-transfer transaction, but see the savings balance after it is credited. As a result, it would obtain an incorrect result.

This chapter introduces the basic concepts of transaction processing. Details on concurrent transaction processing and recovery from failures are in Chapter 18 and Chapter 19, respectively.

17.1 Transaction Concept

A **transaction** is a **unit** of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language (typically SQL), or programming language (e.g., C++ or Java), with embedded database accesses in JDBC or ODBC. A transaction is delimited by statements (or function calls) of the form **begin transaction** and **end transaction**. The transaction consists of all operations executed between the **begin transaction** and **end transaction**.

This collection of steps must appear to the user as a single, indivisible unit. Since a transaction is indivisible, it either executes in its entirety or not at all. Thus, if a

transaction begins to execute but fails for whatever reason, any changes to the database that the transaction may have made must be undone. This requirement holds regardless of whether the transaction itself failed (e.g., if it divided by zero), the operating system crashed, or the computer itself stopped operating. As we shall see, ensuring that this requirement is met is difficult since some changes to the database may still be stored only in the main-memory variables of the transaction, while others may have been written to the database and stored on disk. This “all-or-none” property is referred to as **atomicity**.

Furthermore, since a transaction is a single unit, its actions cannot appear to be separated by other database operations not part of the transaction. While we wish to present this user-level impression of transactions, we know that reality is quite different. Even a single SQL statement involves many separate accesses to the database, and a transaction may consist of several SQL statements. Therefore, the database system must take special actions to ensure that transactions operate properly without interference from concurrently executing database statements. This property is referred to as **isolation**.

Even if the system ensures correct execution of a transaction, this serves little purpose if the system subsequently crashes and, as a result, the system “forgets” about the transaction. Thus, a transaction’s actions must persist across crashes. This property is referred to as **durability**.

Because of the above three properties, transactions are an ideal way of structuring interaction with a database. This leads us to impose a requirement on transactions themselves. A transaction must preserve database consistency—if a transaction is run atomically in isolation starting from a consistent database, the database must again be consistent at the end of the transaction. This consistency requirement goes beyond the data-integrity constraints we have seen earlier (such as primary-key constraints, referential integrity, check constraints, and the like). Rather, transactions are expected to go beyond that to ensure preservation of those application-dependent consistency constraints that are too complex to state using the SQL constructs for data integrity. How this is done is the responsibility of the programmer who codes a transaction. This property is referred to as **consistency**.

To restate the above more concisely, we require that the database system maintain the following properties of the transactions:

- **Atomicity.** Either all operations of the transaction are reflected properly in the database, or none are.
- **Consistency.** Execution of a transaction in isolation (i.e., with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation.** Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished.

Thus, each transaction is unaware of other transactions executing concurrently in the system.

- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the **ACID properties**; the acronym is derived from the first letter of each of the four properties.

As we shall see later, ensuring the isolation property may have a significant adverse effect on system performance. For this reason, some applications compromise on the isolation property. We shall study these compromises after first studying the strict enforcement of the ACID properties.

17.2 A Simple Transaction Model

Because SQL is a powerful and complex language, we begin our study of transactions with a simple database language that focuses on when data are moved from disk to main memory and from main memory to disk. In doing this, we ignore SQL **insert** and **delete** operations and defer considering them until Section 18.4. The only actual operations on the data are restricted in our simple language to arithmetic operations. Later we shall discuss transactions in a realistic, SQL-based context with a richer set of operations. The data items in our simplified model contain a single data value (a number in our examples). Each data item is identified by a name (typically a single letter in our examples, that is, *A*, *B*, *C*, etc.).

We shall illustrate the transaction concept using a simple bank application consisting of several accounts and a set of transactions that access and update those accounts. Transactions access data using two operations:

- **read(*X*)**, which transfers the data item *X* from the database to a variable, also called *X*, in a buffer in main memory belonging to the transaction that executed the **read** operation.
- **write(*X*)**, which transfers the value in the variable *X* in the main-memory buffer of the transaction that executed the **write** to the data item *X* in the database.

It is important to know if a change to a data item appears only in main memory or if it has been written to the database on disk. In a real database system, the **write** operation does not necessarily result in the immediate update of the data on the disk; the **write** operation may be temporarily stored elsewhere and executed on the disk later. For now, however, we shall assume that the **write** operation updates the database immediately. We discuss storage issues further in Section 17.3 and discuss the issue of when database data in main memory are written to the database on disk in Chapter 19.

Let T_i be a transaction that transfers \$50 from account A to account B . This transaction can be defined as:

```
 $T_i$ : read( $A$ );
 $A := A - 50$ ;
write( $A$ );
read( $B$ );
 $B := B + 50$ ;
write( $B$ ).
```

Let us now consider each of the ACID properties. (For ease of presentation, we consider them in an order different from the order A-C-I-D.)

- **Consistency:** The consistency requirement here is that the sum of A and B be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction. This task may be facilitated by automatic testing of integrity constraints, as we discussed in Section 4.4.

- **Atomicity:** Suppose that, just before the execution of transaction T_i , the values of accounts A and B are \$1000 and \$2000, respectively. Now suppose that, during the execution of transaction T_i , a failure occurs that prevents T_i from completing its execution successfully. Further, suppose that the failure happened after the **write**(A) operation but before the **write**(B) operation. In this case, the values of accounts A and B reflected in the database are \$950 and \$2000. The system destroyed \$50 as a result of this failure. In particular, we note that the sum $A + B$ is no longer preserved.

Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction T_i is executed to completion, there exists a point at which the value of account A is \$950 and the value of account B is \$2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account A is \$950, and the value of account B is \$2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are.

The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write. This information is written to a file called the *log*. If the transaction does not complete its execution, the database system restores the old values from the log to make it appear as though the transaction never executed. We discuss these ideas further in Section 17.4. Ensuring atomicity is the responsibility of the database system; specifically, it is handled by a component of the database called the **recovery system**, which we describe in detail in Chapter 19.

- **Durability:** Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure can result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. Protection against loss of data on disk is discussed in Chapter 19. We can guarantee durability by ensuring that either:

1. The updates carried out by the transaction have been written to disk before the transaction completes.
2. Information about the updates carried out by the transaction is written to disk, and such information is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

The recovery system of the database, described in Chapter 19, is responsible for ensuring durability, in addition to ensuring atomicity.

- **Isolation:** Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state.

For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from *A* to *B* is executing, with the deducted total written to *A* and the increased total yet to be written to *B*. If a second concurrently running transaction reads *A* and *B* at this intermediate point and computes *A* + *B*, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on *A* and *B* based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed.

A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits, as we shall see in Section

17.5. Other solutions have therefore been developed; they allow multiple transactions to execute concurrently.

We discuss the problems caused by concurrently executing transactions in Section 17.5. The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order. We shall discuss the principles of isolation further in Section 17.6. Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control system**, which we discuss in Chapter 18.

17.3 Storage Structure

To understand how to ensure the atomicity and durability properties of a transaction, we must gain a better understanding of how the various data items in the database may be stored and accessed.

In Chapter 12, we saw that storage media can be distinguished by their relative speed, capacity, and resilience to failure, and classified as volatile storage or non-volatile storage. We review these terms and introduce another class of storage, called stable storage.

- **Volatile storage.** Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main memory and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself and because it is possible to access any data item in volatile storage directly.
- **Non-volatile storage.** Information residing in non-volatile storage survives system crashes. Examples of non-volatile storage include secondary storage devices such as magnetic disk and flash storage, used for online storage, and tertiary storage devices such as optical media and magnetic tapes, used for archival storage. At the current state of technology, non-volatile storage is slower than volatile storage, particularly for random access. Both secondary and tertiary storage devices, however, are susceptible to failures that may result in loss of information.
- **Stable storage.** Information residing in stable storage is *never* lost (*never* should be taken with a grain of salt, since theoretically *never* cannot be guaranteed—for example, it is possible, although extremely unlikely, that a black hole may envelop the earth and permanently destroy all data!). Although stable storage is theoretically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely. To implement stable storage, we replicate the information in several non-volatile storage media (usually disk) with independent failure modes. Updates must be done with care to ensure that a failure during an update to stable storage does not cause a loss of information. Section 19.2.1 discusses stable-storage implementation.

The distinctions among the various storage types can be less clear in practice than in our presentation. For example, certain systems, for example some RAID controllers, provide battery backup, so that some main memory can survive system crashes and power failures.

For a transaction to be durable, its changes need to be written to stable storage. Similarly, for a transaction to be atomic, log records need to be written to stable storage before any changes are made to the database on disk. The degree to which a system ensures durability and atomicity depends on how stable its implementation of stable storage really is. In some cases, a single copy on disk is considered sufficient, but applications whose data are highly valuable and whose transactions are highly important require multiple copies, or, in other words, a closer approximation of the idealized concept of stable storage.

17.4 Transaction Atomicity and Durability

As we noted earlier, a transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Thus, any changes that the aborted transaction made to the database must be undone. Once the changes caused by an **aborted transaction have been undone**, we say that the transaction has been **rolled back**. It is part of the responsibility of the recovery scheme to manage transaction aborts. This is done typically by maintaining a **log**. Each database modification made by a transaction is first recorded in the log. We record the identifier of the transaction performing the modification, the identifier of the data item being modified, and both the old value (prior to modification) and the new value (after modification) of the data item. Only then is the database itself modified. Maintaining a log provides the possibility of redoing a modification to ensure atomicity and durability as well as the possibility of undoing a modification to ensure atomicity in case of a failure during transaction execution. Details of log-based recovery are discussed in Chapter 19.

A transaction that completes its execution successfully is said to be **committed**. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.

Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a compensating transaction. For instance, if a transaction added \$20 to an account, the compensating transaction would subtract \$20 from the account. However, it is not always possible to create such a compensating transaction. Therefore, the responsibility of writing and executing a compensating transaction is left to the user and is not handled by the database system.

We need to be more precise about what we mean by *successful completion* of a transaction. We therefore establish a simple abstract transaction model. A transaction must be in one of the following states:

- **Active**, the initial state; the transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed**, after successful completion.

The state diagram corresponding to a transaction appears in Figure 17.1. We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if it has either committed or aborted.

A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

As mentioned earlier, we assume for now that failures do not result in loss of data on disk. Chapter 19 discusses techniques to deal with loss of data on disk.

A transaction enters the failed state after the system determines that the transaction can no longer proceed with its normal execution (e.g., because of hardware or logical errors). Such a transaction must be rolled back. Then, it enters the aborted state. At this point, the system has two options:

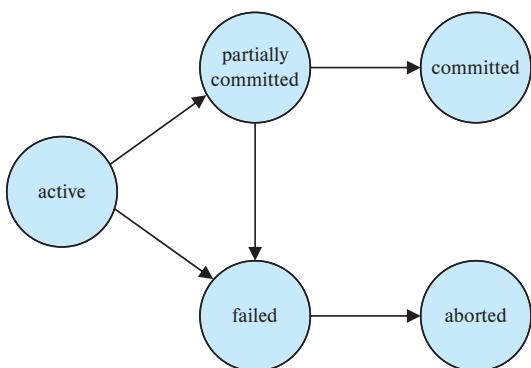


Figure 17.1 State diagram of a transaction.

- It can **restart** the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can **kill** the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

We must be cautious when dealing with **observable external writes**, such as **writes to a user's screen, or sending email**. Once such a write has occurred, it cannot be erased, since it may have been seen external to the database system. Most systems allow such writes to take place only after the transaction has entered the committed state. One way to implement such a scheme is for the database system to **store any value associated with such external writes temporarily in a special relation** in the database, and to perform the actual writes only after the transaction enters the committed state. If the system should fail after the transaction has entered the committed state, but before it could complete the external writes, the database system will carry out the external writes (using the data in non-volatile storage) when the system is restarted.

Handling external writes can be more complicated in some situations. For example, suppose the external action is that of dispensing cash at an automated teller machine, and the system fails just before the cash is actually dispensed (we assume that cash can be dispensed atomically). It makes no sense to dispense cash when the system is restarted, since the user may have left the machine. In such a case a **compensating transaction**, such as depositing the cash back into the user's account, needs to be executed when the system is restarted.

As another example, consider a user making a booking over the web. It is possible that the database system or the application server crashes just after the booking transaction commits. It is also possible that the network connection to the user is lost just after the booking transaction commits. In either case, even though the transaction has committed, the external write has not taken place. To handle such situations, the application must be designed such that when the user connects to the web application again, she will be able to see whether her transaction had succeeded or not.

For certain applications, it may be desirable to allow active transactions to display data to users, particularly for long-duration transactions that run for minutes or hours. Unfortunately, we cannot allow such output of observable data unless we are willing to compromise transaction atomicity.

17.5 Transaction Isolation

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data, as we saw earlier. Ensuring consistency in spite of

Note 17.1 TRENDS IN CONCURRENCY

Several current trends in the field of computing are giving rise to an increase in the amount of concurrency possible. As database systems exploit this concurrency to increase overall system performance, there will necessarily be an increasing number of transactions run concurrently.

Early computers had only one processor. Therefore, there was never any real concurrency in the computer. The only concurrency was apparent concurrency created by the operating system as it shared the processor among several distinct tasks or processes. Modern computers are likely to have many processors. Each processor is referred to as a *core*; a single processor chip may contain several cores, and several such chips may be connected together in a single system, which all share a common system memory. Further, parallel database systems may contain multiple such systems. Parallel database architectures are discussed in Chapter 20.

The parallelism provided by multiple processors and cores is used for two purposes. One is to execute different parts of a single long running query in parallel, to speed up query execution. The other is to allow a large number of queries (often much smaller queries) to execute concurrently, for example to support a very large number of concurrent users. Chapter 21 through Chapter 23 describe algorithms for building parallel database systems.

concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization.** A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. **While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction.** All of this increases the **throughput** of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.
- **Reduced waiting time.** There may be a mix of transactions running on a system, some short and some long. If transactions run serially, **a short transaction may have to wait for a preceding long transaction to complete**, which can lead to un-

predictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted.

The motivation for using concurrent execution in a database is essentially the same as the motivation for using **multiprogramming** in an operating system.

When several transactions run concurrently, the isolation property may be violated, resulting in database consistency being destroyed despite the correctness of each individual transaction. In this section, we present the concept of schedules to help identify those executions that are guaranteed to ensure the isolation property and thus database consistency.

The database system must **control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database**. It does so through a variety of mechanisms called **concurrency-control schemes**. We study concurrency-control schemes in Chapter 18; for now, we focus on the concept of correct concurrent execution.

Consider again the simplified banking system of Section 17.1, which has several accounts, and a set of transactions that access and update those accounts. Let T_1 and T_2 be two transactions that transfer funds from one account to another. Transaction T_1 transfers \$50 from account A to account B . It is defined as:

```
 $T_1$ : read( $A$ );
       $A := A - 50$ ;
      write( $A$ );
      read( $B$ );
       $B := B + 50$ ;
      write( $B$ ).
```

Transaction T_2 transfers 10 percent of the balance from account A to account B . It is defined as:

```
 $T_2$ : read( $A$ );
       $temp := A * 0.1$ ;
       $A := A - temp$ ;
      write( $A$ );
      read( $B$ );
       $B := B + temp$ ;
      write( $B$ ).
```

T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$\text{read}(A)$ $temp := A * 0.1$ $A := A - temp$ $\text{write}(A)$ $\text{read}(B)$ $B := B + temp$ $\text{write}(B)$ commit

Figure 17.2 Schedule 1—a serial schedule in which T_1 is followed by T_2 .

Suppose the current values of accounts A and B are \$1000 and \$2000, respectively. Suppose also that the two transactions are executed one at a time in the order T_1 followed by T_2 . This execution sequence appears in Figure 17.2. In the figure, the sequence of instruction steps is in chronological order from top to bottom, with instructions of T_1 appearing in the left column and instructions of T_2 appearing in the right column. The final values of accounts A and B , after the execution in Figure 17.2 takes place, are \$855 and \$2145, respectively. Thus, the total amount of money in accounts A and B —that is, the sum $A + B$ —is preserved after the execution of both transactions.

Similarly, if the transactions are executed one at a time in the order T_2 followed by T_1 , then the corresponding execution sequence is that of Figure 17.3. Again, as expected, the sum $A + B$ is preserved, and the final values of accounts A and B are \$850 and \$2150, respectively.

The execution sequences just described are called **schedules**. They represent the **chronological order** in which instructions are executed in the system. Clearly, a schedule for a set of transactions must consist of all instructions of those transactions and they must preserve the order in which the instructions appear in each individual transaction. For example, in transaction T_1 , the instruction $\text{write}(A)$ must appear before the instruction $\text{read}(B)$, in any valid schedule. Note that we include in our schedules the **commit** operation to indicate that the transaction has entered the committed state. In the following discussion, we shall refer to the first execution sequence (T_1 followed by T_2) as schedule 1, and to the second execution sequence (T_2 followed by T_1) as schedule 2.

T_1	T_2
<pre> read(A) temp := A * 0.1 A := A - temp write(A) read(B) B := B + temp write(B) commit </pre> <pre> read(A) A := A - 50 write(A) read(B) B := B + 50 write(B) commit </pre>	<pre> read(A) temp := A * 0.1 A := A - temp write(A) read(B) B := B + temp write(B) commit </pre>

Figure 17.3 Schedule 2—a serial schedule in which T_2 is followed by T_1 .

These schedules are **serial**: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Recalling a well-known formula from combinatorics, we note that, for a set of n transactions, there exist n factorial ($n!$) different valid serial schedules.

When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction.¹

Returning to our previous example, suppose that the two transactions are executed concurrently. One possible schedule appears in Figure 17.4. After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order T_1 followed by T_2 . The sum $A + B$ is indeed preserved.

¹The number of possible schedules for a set of n transactions is very large. There are $n!$ different serial schedules. Considering all the possible ways that steps of transactions might be interleaved, the total number of possible schedules is much larger than $n!$.

T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$	$\text{read}(A)$ $temp := A * 0.1$ $A := A - temp$ $\text{write}(A)$
$\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$\text{read}(B)$ $B := B + temp$ $\text{write}(B)$ commit

Figure 17.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure 17.5. After the execution of this schedule, we arrive at a state where the final values of accounts A and B are \$950 and \$2100, respectively. This final state is an *inconsistent state*, since we have gained \$50 in the process of the concurrent execution. Indeed, the sum $A + B$ is not preserved by the execution of the two transactions.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The **concurrency-control** component of the database system carries out this task.

We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called **serializable** schedules.

17.6

Serializability

Before we can consider how the concurrency-control component of the database system can ensure serializability, we consider how to determine when a schedule is serializable. Certainly, **serial schedules are serializable**, but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable. Since trans-

T_1	T_2
$\text{read}(A)$ $A := A - 50$	$\text{read}(A)$ $temp := A * 0.1$ $A := A - temp$ $\text{write}(A)$ $\text{read}(B)$
$\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$B := B + temp$ $\text{write}(B)$ commit

Figure 17.5 Schedule 4—a concurrent schedule resulting in an inconsistent state.

actions are programs, it is difficult to determine exactly what operations a transaction performs and how operations of various transactions interact. For this reason, we shall not consider the various types of operations that a transaction can perform on a data item, but instead consider only two operations: **read** and **write**. We assume that, between a $\text{read}(Q)$ instruction and a $\text{write}(Q)$ instruction on a data item Q , a transaction may perform an arbitrary sequence of operations on the copy of Q that is residing in the local buffer of the transaction. In this model, the only significant operations of a transaction, from a scheduling point of view, are its **read** and **write** instructions. Commit operations, though relevant, are not considered until Section 17.7. We therefore may show only **read** and **write** instructions in schedules, as we do for schedule 3 in Figure 17.6.

In this section, we discuss different forms of schedule equivalence but focus on a particular form called **conflict serializability**.

Let us consider a schedule S in which there are two consecutive instructions, I and J , of transactions T_i and T_j , respectively ($i \neq j$). If I and J refer to different data items, then we can swap I and J without affecting the results of any instruction in the schedule. However, if I and J refer to the same data item Q , then the order of the two steps may matter. Since we are dealing with only **read** and **write** instructions, there are four cases that we need to consider:

1. $I = \text{read}(Q)$, $J = \text{read}(Q)$. The order of I and J does not matter, since the same value of Q is read by T_i and T_j , regardless of the order.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 17.6 Schedule 3—showing only the read and write instructions.

2. $I = \text{read}(Q)$, $J = \text{write}(Q)$. If I comes before J , then T_i does not read the value of Q that is written by T_j in instruction J . If J comes before I , then T_i reads the value of Q that is written by T_j . Thus, the order of I and J matters.
3. $I = \text{write}(Q)$, $J = \text{read}(Q)$. The order of I and J matters for reasons similar to those of the previous case.
4. $I = \text{write}(Q)$, $J = \text{write}(Q)$. Since both instructions are **write** operations, the order of these instructions does not affect either T_i or T_j . However, the value obtained by the next $\text{read}(Q)$ instruction of S is affected, since the result of only the latter of the two **write** instructions is preserved in the database. If there is no other $\text{write}(Q)$ instruction after I and J in S , then the order of I and J directly affects the final value of Q in the database state that results from schedule S .

Thus, only in the case where both I and J are **read** instructions does the relative order of their execution not matter.

We say that I and J **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a **write** operation.

To illustrate the concept of conflicting instructions, we consider schedule 3 in Figure 17.6. The **write(A)** instruction of T_1 conflicts with the **read(A)** instruction of T_2 . However, the **write(A)** instruction of T_2 does not conflict with the **read(B)** instruction of T_1 because the two instructions access different data items.

Let I and J be consecutive instructions of a schedule S . If I and J are instructions of different transactions and I and J do not conflict, then we can swap the order of I and J to produce a new schedule S' . S is equivalent to S' , since all instructions appear in the same order in both schedules except for I and J , whose order does not matter.

Since the **write(A)** instruction of T_2 in schedule 3 of Figure 17.6 does not conflict with the **read(B)** instruction of T_1 , we can swap these instructions to generate an equivalent schedule, schedule 5, in Figure 17.7. Regardless of the initial system state, schedules 3 and 5 both produce the same final system state.

T_1	T_2
read(A)	
write(A)	
read(B)	read(A)
	write(A)
write(B)	read(B)
	write(B)

Figure 17.7 Schedule 5—schedule 3 after swapping of a pair of instructions.

We continue to swap nonconflicting instructions:

- Swap the `read(B)` instruction of T_1 with the `read(A)` instruction of T_2 .
- Swap the `write(B)` instruction of T_1 with the `write(A)` instruction of T_2 .
- Swap the `write(B)` instruction of T_1 with the `read(A)` instruction of T_2 .

The final result of these swaps, schedule 6 of Figure 17.8, is a serial schedule. Note that schedule 6 is exactly the same as schedule 1, but it shows only the `read` and `write` instructions. Thus, we have shown that schedule 3 is equivalent to a serial schedule. This equivalence implies that, regardless of the initial system state, schedule 3 produces the same final state as some serial schedule.

If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.²

T_1	T_2
read(A)	
write(A)	
read(B)	read(A)
write(B)	write(A)
	read(B)
	write(B)

Figure 17.8 Schedule 6—a serial schedule that is equivalent to schedule 3.

²We use the term *conflict equivalent* to distinguish the way we have just defined equivalence from other definitions that we shall discuss later on in this section.

T_3	T_4
read(Q)	
write(Q)	write(Q)

Figure 17.9 Schedule 7.

Not all serial schedules are conflict equivalent to each other. For example, schedules 1 and 2 are not conflict equivalent.

The concept of conflict equivalence leads to the concept of conflict serializability. We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule. Thus, schedule 3 is conflict serializable, since it is conflict equivalent to the serial schedule 1.

Finally, consider schedule 7 of Figure 17.9; it consists of only the significant operations (that is, the **read** and **write**) of transactions T_3 and T_4 . This schedule is not conflict serializable, since it is not equivalent to either the serial schedule $\langle T_3, T_4 \rangle$ or the serial schedule $\langle T_4, T_3 \rangle$.

We now present a simple and efficient method for determining the conflict serializability of a schedule. Consider a schedule S . We construct a directed graph, called a **precedence graph**, from S . This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

1. T_i executes **write**(Q) before T_j executes **read**(Q).
2. T_i executes **read**(Q) before T_j executes **write**(Q).
3. T_i executes **write**(Q) before T_j executes **write**(Q).

If an edge $T_i \rightarrow T_j$ exists in the precedence graph, then, in any serial schedule S' equivalent to S , T_i must appear before T_j .

For example, the precedence graph for schedule 1 in Figure 17.10a contains the single edge $T_1 \rightarrow T_2$, since all the instructions of T_1 are executed before the first instruction of T_2 is executed. Similarly, Figure 17.10b shows the precedence graph for

**Figure 17.10** Precedence graph for (a) schedule 1 and (b) schedule 2.

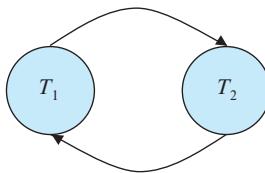


Figure 17.11 Precedence graph for schedule 4.

schedule 2 with the single edge $T_2 \rightarrow T_1$, since all the instructions of T_2 are executed before the first instruction of T_1 is executed.

The precedence graph for schedule 4 appears in Figure 17.11. It contains the edge $T_1 \rightarrow T_2$ because T_1 executes $\text{read}(A)$ before T_2 executes $\text{write}(A)$. It also contains the edge $T_2 \rightarrow T_1$ because T_2 executes $\text{read}(B)$ before T_1 executes $\text{write}(B)$.

If the precedence graph for S has a cycle, then schedule S is not conflict serializable. If the graph contains no cycles, then the schedule S is conflict serializable.

A **serializability order** of the transactions can be obtained by finding a linear order consistent with the partial order of the precedence graph. This process is called **topological sorting**. There are, in general, several possible linear orders that can be obtained through a topological sort. For example, the graph of Figure 17.12a has the two acceptable linear orderings shown in Figure 17.12b and Figure 17.12c.

Thus, to test for conflict serializability, we need to construct the precedence graph and to invoke a cycle-detection algorithm. Cycle-detection algorithms can be found in standard textbooks on algorithms. Cycle-detection algorithms, such as those based on depth-first search, require on the order of n^2 operations, where n is the number of vertices in the graph (that is, the number of transactions).³

Returning to our previous examples, note that the precedence graphs for schedules 1 and 2 (Figure 17.10) indeed do not contain cycles. The precedence graph for schedule 4 (Figure 17.11), on the other hand, contains a cycle, indicating that this schedule is not conflict serializable.

It is possible to have two schedules that produce the same outcome but that are not conflict equivalent. For example, consider transaction T_5 , which transfers \$10 from account B to account A . Let schedule 8 be as defined in Figure 17.13. We claim that schedule 8 is not conflict equivalent to the serial schedule $\langle T_1, T_5 \rangle$, since, in schedule 8, the $\text{write}(B)$ instruction of T_5 conflicts with the $\text{read}(B)$ instruction of T_1 . This creates an edge $T_5 \rightarrow T_1$ in the precedence graph. Similarly, we see that the $\text{write}(A)$ instruction of T_1 conflicts with the read instruction of T_5 , creating an edge $T_1 \rightarrow T_5$. This shows that the precedence graph has a cycle and that schedule 8 is not serializable. However, the final values of accounts A and B after the execution of either schedule 8 or the serial schedule $\langle T_1, T_5 \rangle$ are the same—\$960 and \$2040, respectively.

³If instead we measure complexity in terms of the number of edges, which corresponds to the number of actual conflicts between active transactions, then depth-first-based cycle detection is linear.

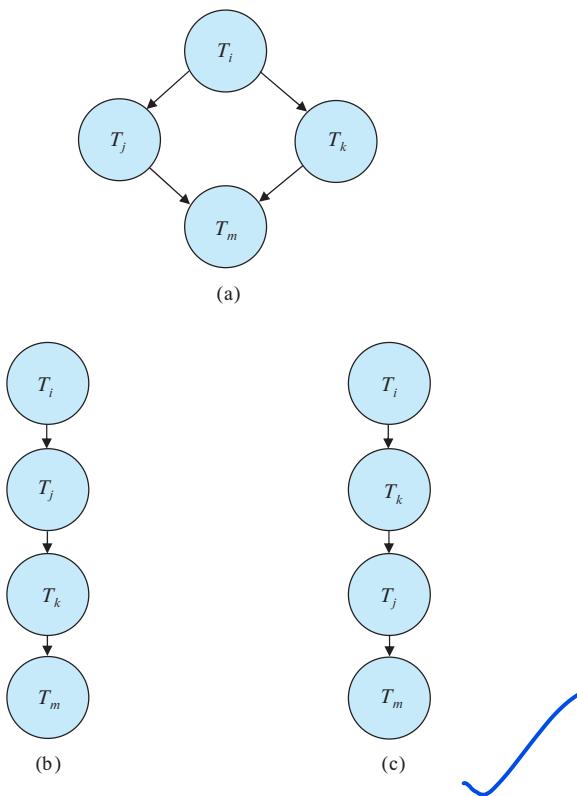


Figure 17.12 Illustration of topological sorting.

We can see from this example that there are less-stringent definitions of schedule equivalence than conflict equivalence. For the system to determine that schedule 8 produces the same outcome as the serial schedule $\langle T_1, T_5 \rangle$, it must analyze the computation performed by T_1 and T_5 , rather than just the **read** and **write** operations. In general, such analysis is hard to implement and is computationally expensive. In our example, the final result is the same as that of a serial schedule because of the mathematical fact that the increment and decrement operations are commutative. While this may be easy to see in our simple example, the general case is not so easy since a transaction may be expressed as a complex SQL statement, a Java program with JDBC calls, etc.

However, there are other definitions of schedule equivalence based purely on the **read** and **write** operations. One such definition is *view equivalence*, a definition that leads to the concept of *view serializability*. View serializability is not used in practice due to its high degree of computational complexity.⁴ We therefore defer discussion of

⁴Testing for view serializability has been proven to be NP-complete, which means that it is virtually certain that no efficient test for view serializability exists.

T_1	T_5
<code>read(A)</code>	
$A := A - 50$	
<code>write(A)</code>	
	<code>read(B)</code>
	$B := B - 10$
	<code>write(B)</code>
<code>read(B)</code>	
$B := B + 50$	
<code>write(B)</code>	
	<code>read(A)</code>
	$A := A + 10$
	<code>write(A)</code>

Figure 17.13 Schedule 8.

view serializability to Chapter 18, but, for completeness, note here that the example of schedule 8 is not view serializable.

17.7

Transaction Isolation and Atomicity

So far, we have studied schedules while assuming implicitly that there are no transaction failures. We now address the effect of transaction failures during concurrent execution.

If a transaction T_i fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, the atomicity property requires that any transaction T_j that is dependent on T_i (i.e., T_j has read data written by T_i) is also aborted. To achieve this, we need to place restrictions on the types of schedules permitted in the system.

In the following two subsections, we address the issue of what schedules are acceptable from the viewpoint of recovery from transaction failure. We describe in Chapter 18 how to ensure that only such acceptable schedules are generated.

17.7.1 Recoverable Schedules

Consider the partial schedule 9 in Figure 17.14, in which T_7 is a transaction that performs only one instruction: `read(A)`. We call this a **partial schedule** because we have not included a **commit** or **abort** operation for T_6 . Notice that T_7 commits immediately after executing the `read(A)` instruction. Thus, T_7 commits while T_6 is still in the active state. Now suppose that T_6 fails before it commits. T_7 has read the value of data item A written by T_6 . Therefore, we say that T_7 is **dependent** on T_6 . Because of this, we must abort T_7 to ensure atomicity. However, T_7 has already committed and cannot be

T_6	T_7
read(A)	
write(A)	
read(B)	read(A) commit

Figure 17.14 Schedule 9, a nonrecoverable schedule.

aborted. Thus, we have a situation where it is impossible to recover correctly from the failure of T_6 .

Schedule 9 is an example of a *nonrecoverable* schedule. A **recoverable schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i , the commit operation of T_i appears before the commit operation of T_j . For the example of schedule 9 to be recoverable, T_7 would have to delay committing until after T_6 commits.

17.7.2 Cascadeless Schedules

Even if a schedule is recoverable, to recover correctly from the failure of a transaction T_i , we may have to roll back several transactions. Such situations occur if transactions have read data written by T_i . As an illustration, consider the partial schedule of Figure 17.15. Transaction T_8 writes a value of A that is read by transaction T_9 . Transaction T_9 writes a value of A that is read by transaction T_{10} . Suppose that, at this point, T_8 fails. T_8 must be rolled back. Since T_9 is dependent on T_8 , T_9 must be rolled back. Since T_{10} is dependent on T_9 , T_{10} must be rolled back. This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**.

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)	read(A) write(A)	read(A)
abort		

Figure 17.15 Schedule 10.

Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work. It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called *cascadeless schedules*. Formally, a **cascadeless schedule** is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i , the commit operation of T_i appears before the read operation of T_j . It is easy to verify that every cascadeless schedule is also recoverable.

17.8 Transaction Isolation Levels

Serializability is a useful concept because it allows programmers to ignore issues related to concurrency when they code transactions. If every transaction has the property that it maintains database consistency if executed alone, then serializability ensures that concurrent executions maintain consistency. However, the protocols required to ensure serializability may allow too little concurrency for certain applications. In these cases, weaker levels of consistency are used. The use of weaker levels of consistency places additional burdens on programmers for ensuring database correctness.

The SQL standard also allows a transaction to specify that it may be executed in such a way that it becomes nonserializable with respect to other transactions. For instance, a transaction may operate at the isolation level of **read uncommitted**, which permits the transaction to read a data item even if it was written by a transaction that has not been committed. SQL provides such features for the benefit of long transactions whose results do not need to be precise. If these transactions were to execute in a serializable fashion, they could interfere with other transactions, causing the others' execution to be delayed.

The **isolation levels** specified by the SQL standard are as follows:

- **Serializable** usually ensures serializable execution. However, as we shall explain shortly, some database systems implement this isolation level in a manner that may, in certain cases, allow nonserializable executions.
- **Repeatable read** allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it. However, the transaction may not be serializable with respect to other transactions. For instance, when it is searching for data satisfying some conditions, a transaction may find some of the data inserted by a committed transaction, but may not find other data inserted by the same transaction.
- **Read committed** allows only committed data to be read, but does not require repeatable reads. For instance, between two reads of a data item by the transaction, another transaction may have updated the data item and committed.
- **Read uncommitted** allows uncommitted data to be read. It is the lowest isolation level allowed by SQL.

All the isolation levels above additionally disallow **dirty writes**, that is, they disallow writes to a data item that has already been written by another transaction that has not yet committed or aborted.

Many database systems run, by default, at the read-committed isolation level. In SQL, it is possible to set the isolation level explicitly, rather than accepting the system's default setting. For example, the statement

```
set transaction isolation level serializable
```

sets the isolation level to serializable; any of the other isolation levels may be specified instead. The preceding syntax is supported by Oracle, PostgreSQL, and SQL Server; Oracle uses the syntax

```
alter session set isolation_level = serializable
```

while DB2 uses the syntax “**change isolation level**” with its own abbreviations for isolation levels. Changing of the isolation level must be done as the first statement of a transaction.

By default, most databases commit individual statements as soon as they are executed. Such **automatic commit** of individual statements must be turned off to allow multiple statements to run as a single transaction. The command **start transaction** ensures that subsequent SQL statements, until a subsequent **commit** or **rollback**, are executed as a single transaction. As expected, the **commit** operation commits the preceding SQL statements, while **rollback** rolls back the preceding SQL statements. (SQL Server uses **begin transaction** in place of **start transaction**, while Oracle and PostgreSQL treat **begin** as identical to **start transaction**.)

APIs such as JDBC and ODBC provide functions to turn off automatic commit. In JDBC the **setAutoCommit** method of the **Connection** interface (which we saw earlier in Section 5.1.1.8) can be used to turn automatic commit off by invoking **setAutoCommit(false)**, or on by invoking **setAutoCommit(true)**. Further, in JDBC the method **setTransactionIsolation(int level)** of the **Connection** interface can be invoked with any one of

- `Connection.TRANSACTION_SERIALIZABLE`,
- `Connection.TRANSACTION_REPEATABLE_READ`,
- `Connection.TRANSACTION_READ_COMMITTED`, or
- `Connection.TRANSACTION_READ_UNCOMMITTED`

to set the transaction isolation level correspondingly.

An application designer may decide to accept a weaker isolation level in order to improve system performance. As we shall see in Section 17.9 and Chapter 18, ensuring serializability may force a transaction to wait for other transactions or, in some cases, to abort because the transaction can no longer be executed as part of a serializable execution. While it may seem shortsighted to risk database consistency for performance,

this trade-off makes sense if we can be sure that the inconsistency that may occur is not relevant to the application.

There are many means of implementing isolation levels. As long as the implementation ensures serializability, the designer of a database application or a user of an application does not need to know the details of such implementations, except perhaps for dealing with performance issues. Unfortunately, even if the isolation level is set to **serializable**, some database systems actually implement a weaker level of isolation, which does not rule out every possible nonserializable execution; we revisit this issue in Section 17.9. If weaker levels of isolation are used, either explicitly or implicitly, the application designer has to be aware of some details of the implementation, to avoid or minimize the chance of inconsistency due to lack of serializability.

17.9 Implementation of Isolation Levels

So far, we have seen what properties a schedule must have if it is to leave the database in a consistent state and allow transaction failures to be handled in a safe manner.

There are various **concurrency-control** policies that we can use to ensure that, even when multiple transactions are executed concurrently, only acceptable schedules are generated, regardless of how the operating system time-shares resources (such as CPU time) among the transactions.

As a trivial example of a concurrency-control policy, consider this: A transaction acquires a **lock** on the entire database before it starts and releases the lock after it has committed. While a transaction holds a lock, no other transaction is allowed to acquire the lock, and all must therefore wait for the lock to be released. As a result of the locking policy, only one transaction can execute at a time. Therefore, only serial schedules are generated. These are trivially serializable, and it is easy to verify that they are recoverable and cascadeless as well.

A concurrency-control policy such as this one leads to poor performance, since it forces transactions to wait for preceding transactions to finish before they can start. In other words, it provides a poor degree of concurrency (indeed, no concurrency at all). As we saw in Section 17.5, concurrent execution has substantial performance benefits.

The goal of concurrency-control policies is to provide a high degree of concurrency, while ensuring that all schedules that can be generated are conflict or view serializable, recoverable, and cascadeless.

Here we provide an overview of how some of most important concurrency-control mechanisms work, and we defer the details to Chapter 18.

17.9.1 Locking

Instead of locking the entire database, a transaction could instead lock only those data items that it accesses. Under such a policy, the transaction must hold locks long enough to ensure serializability, but for a period short enough not to harm performance exces-

Note 17.2 SERIALIZABILITY IN THE REAL WORLD

Serializable schedules are the ideal way to ensure consistency, but in our day-to-day lives, we don't impose such stringent requirements. A web site offering goods for sale may list an item as being in stock, yet by the time a user selects the item and goes through the checkout process, that item might no longer be available. Viewed from a database perspective, this would be a nonrepeatable read.

As another example, consider seat selection for air travel. Assume that a traveler has already booked an itinerary and now is selecting seats for each flight. Many airline web sites allow the user to step through the various flights and choose a seat, after which the user is asked to confirm the selection. It could be that other travelers are selecting seats or changing their seat selections for the same flights at the same time. The seat availability that the traveler was shown is thus actually changing, but the traveler is shown a snapshot of the seat availability as of when the traveler started the seat selection process.

Even if two travelers are selecting seats at the same time, most likely they will select different seats, and if so there would be no real conflict. However, the transactions are not serializable, since each traveler has read data that was subsequently updated by the other traveler, leading to a cycle in the precedence graph. If two travelers performing seat selection concurrently actually selected the same seat, one of them would not be able to get the seat they selected; however, the situation could be easily resolved by asking the traveler to perform the selection again, with updated seat availability information.

It is possible to enforce serializability by allowing only one traveler to do seat selection for a particular flight at a time. However, doing so could cause significant delays as travelers would have to wait for their flight to become available for seat selection; in particular a traveler who takes a long time to make a choice could cause serious problems for other travelers. Instead, any such transaction is typically broken up into a part that requires user interaction and a part that runs exclusively on the database. In the example above, the database transaction would check if the seats chosen by the user are still available, and if so update the seat selection in the database. Serializability is ensured only for the transactions that run on the database, without user interaction.

sively. Complicating matters are SQL statements where the data items accessed depend on a **where** clause, which we discuss in Section 17.10. In Chapter 18, we present the two-phase locking protocol, a simple, widely used technique that ensures serializability. Stated simply, two-phase locking requires a transaction to have two phases, one where it acquires locks but does not release any, and a second phase where the transaction releases locks but does not acquire any. (In practice, locks are usually released only when the transaction completes its execution and has been either committed or aborted.)

Further improvements to locking result if we have two kinds of locks: shared and exclusive. Shared locks are used for data that the transaction reads and exclusive locks are used for those it writes. Many transactions can hold shared locks on the same data item at the same time, but a transaction is allowed an exclusive lock on a data item only if no other transaction holds any lock (regardless of whether shared or exclusive) on the data item. This use of two modes of locks along with two-phase locking allows concurrent reading of data while still ensuring serializability.

17.9.2 Timestamps

Another category of techniques for the implementation of isolation assigns each transaction a **timestamp**, typically when it begins. For each data item, the system keeps two timestamps. The read timestamp of a data item holds the largest (that is, the most recent) timestamp of those transactions that read the data item. The write timestamp of a data item holds the timestamp of the transaction that wrote the current value of the data item. Timestamps are used to ensure that transactions access each data item in order of the transactions' timestamps if their accesses conflict. When this is not possible, offending transactions are aborted and restarted with a new timestamp.

17.9.3 Multiple Versions and Snapshot Isolation

By maintaining more than one version of a data item, it is possible to allow a transaction to read an old version of a data item rather than a newer version written by an uncommitted transaction or by a transaction that should come later in the serialization order. There are a variety of multiversion concurrency-control techniques. One in particular, called **snapshot isolation**, is widely used in practice.

In snapshot isolation, we can imagine that each transaction is given its own version, or snapshot, of the database when it begins.⁵ It reads data from this private version and is thus isolated from the updates made by other transactions. If the transaction updates the database, that update appears only in its own version, not in the actual database itself. Information about these updates is saved so that the updates can be applied to the “real” database if the transaction commits.

When a transaction T enters the partially committed state, it then proceeds to the committed state only if no other concurrent transaction has modified data that T intends to update. Transactions that, as a result, cannot commit abort instead.

Snapshot isolation ensures that attempts to read data never need to wait (unlike locking). Read-only transactions cannot be aborted; only those that modify data run a slight risk of aborting. Since each transaction reads its own version or snapshot of the database, reading data does not cause subsequent update attempts by other transactions to wait (unlike locking). Since most transactions are read-only (and most others read more data than they update), this is often a major source of performance improvement as compared to locking.

⁵In reality, the entire database is not copied. Multiple versions are kept only of those data items that are changed.

The problem with snapshot isolation is that, paradoxically, it provides *too much* isolation. Consider two transactions T and T' . In a serializable execution, either T sees all the updates made by T' or T' sees all the updates made by T , because one must follow the other in the serialization order. Under snapshot isolation, there are cases where neither transaction sees the updates of the other. This is a situation that cannot occur in a serializable execution. In many (indeed, most) cases, the data accesses by the two transactions do not conflict and there is no problem. However, if T reads some data item that T' updates and T' reads some data item that T updates, it is possible that both transactions fail to read the update made by the other. The result, as we shall see in Chapter 18, may be an inconsistent database state that, of course, could not be obtained in any serializable execution.

Oracle, PostgreSQL, and SQL Server offer the option of snapshot isolation. Oracle and PostgreSQL versions prior to PostgreSQL 9.1 implement the **serializable** isolation level using snapshot isolation. As a result, their implementation of serializability can, in exceptional circumstances, result in a nonserializable execution being allowed. SQL Server instead includes an additional isolation level beyond the standard ones, called **snapshot**, to offer the option of snapshot isolation. PostgreSQL versions subsequent to 9.1 implement a form of concurrency control called serializable snapshot isolation, which provides the benefits of snapshot isolation while ensuring serializability.

17.10 Transactions as SQL Statements

In Section 4.3, we presented the SQL syntax for specifying the beginning and end of transactions. Now that we have seen some of the issues in ensuring the ACID properties for transactions, we are ready to consider how those properties are ensured when transactions are specified as a sequence of SQL statements rather than the restricted model of simple reads and writes that we considered up to this point.

In our simple model, we assumed a set of data items exists. While our simple model allowed data-item values to be changed, it did not allow data items to be created or deleted. In SQL, however, **insert** statements create new data and **delete** statements delete data. These two statements are, in effect, **write** operations, since they change the database, but their interactions with the actions of other transactions are different from what we saw in our simple model. As an example, consider how insertion or deletion would conflict with the following SQL query, which finds all instructors who earn more than \$90,000:

```
select ID, name
from instructor
where salary > 90000;
```

Using our sample *instructor* relation (Section A.3), we find that only Einstein and Brandt satisfy the condition. Now assume that around the same time we are running our query, another user inserts a new instructor named “James” whose salary is \$100,000.

```
insert into instructor values ('11111', 'James', 'Marketing', 100000);
```

The result of our query depends on whether this insert comes before or after our query is run. In a concurrent execution of these transactions, it is intuitively clear that they conflict, but this is a conflict that may not be captured by our simple model. This situation is referred to as the **phantom phenomenon** because a conflict may exist on “phantom” data.

Our simple model of transactions required that operations operate on a specific data item given as an argument to the operation. In our simple model, we can look at the **read** and **write** steps to see which data items are referenced. But in an SQL statement, the specific data items (tuples) referenced may be determined by a **where** clause predicate. So the same transaction, if run more than once, might reference different data items each time it is run if the values in the database change between runs. In our example, the 'James' tuple is referenced only if our query comes after the insertion. Let T denote the query and let T' denote the insert. If T' comes first, then there is an edge $T' \rightarrow T$ in the precedence graph. However, in the case where the query T comes first, there is no edge in the precedence graph between T and T' despite the actual conflict on phantom data that forces T to be serialized before T' .

The above-mentioned problem demonstrates that it is not sufficient for concurrency control to consider only the tuples that are accessed by a transaction; the information used to find the tuples that are accessed by the transaction must also be considered for the purpose of concurrency control. The information used to find tuples could be updated by an insertion or deletion, or in the case of an index, even by an update to a search-key attribute. For example, if locking is used for concurrency control, the data structures that track the tuples in a relation, as well as index structures, must be appropriately locked. However, such locking can lead to poor concurrency in some situations; index-locking protocols that maximize concurrency, while ensuring serializability in spite of inserts, deletes, and predicates in queries, are discussed in Section 18.4.3.

Let us consider again the query:

```
select ID, name  
from instructor  
where salary > 90000;
```

and the following SQL update:

```
update instructor  
set salary = salary * 0.9  
where name = 'Wu';
```

We now face an interesting situation in determining whether our query conflicts with the update statement. If our query reads the entire *instructor* relation, then it reads the

tuple with Wu's data and conflicts with the update. However, if an index were available that allowed our query direct access to those tuples with $\text{salary} > 90000$, then our query would not have accessed Wu's data at all because Wu's salary is initially \$90,000 in our example instructor relation and reduces to \$81,000 after the update.

However, using the above approach, it would appear that the existence of a conflict depends on a low-level query processing decision by the system that is unrelated to a user-level view of the meaning of the two SQL statements! An alternative approach to concurrency control treats an insert, delete, or update as conflicting with a predicate on a relation, if it could affect the set of tuples selected by a predicate. In our example query above, the predicate is " $\text{salary} > 90000$ ", and an update of Wu's salary from \$90,000 to a value greater than \$90,000, or an update of Einstein's salary from a value greater than \$90,000 to a value less than or equal to \$90,000, would conflict with this predicate. Locking based on this idea is called **predicate locking**; predicate locking is often implemented using locks on index nodes as we see in Section 18.4.3.

17.11 Summary

- A *transaction* is a *unit* of program execution that accesses and possibly updates various data items. Understanding the concept of a transaction is critical for understanding and implementing updates of data in a database in such a way that concurrent executions and failures of various forms do not result in the database becoming inconsistent.
- Transactions are required to have the ACID properties: atomicity, consistency, isolation, and durability.
 - Atomicity ensures that either all the effects of a transaction are reflected in the database, or none are; a failure cannot leave the database in a state where a transaction is partially executed.
 - Consistency ensures that, if the database is initially consistent, the execution of the transaction (by itself) leaves the database in a consistent state.
 - Isolation ensures that concurrently executing transactions are isolated from one another, so that each has the impression that no other transaction is executing concurrently with it.
 - Durability ensures that, once a transaction has been committed, that transaction's updates do not get lost, even if there is a system failure.
- Concurrent execution of transactions improves throughput of transactions and system utilization and also reduces the waiting time of transactions.
- The various types of storage in a computer are volatile storage, non-volatile storage, and stable storage. Data in volatile storage, such as in RAM, are lost when the computer crashes. Data in non-volatile storage, such as disk, are not lost when

the computer crashes but may occasionally be lost because of failures such as disk crashes. Data in stable storage are never lost.

- Stable storage that must be accessible online is approximated with mirrored disks, or other forms of RAID, which provide redundant data storage. Offline, or archival, stable storage may consist of multiple tape copies of data stored in physically secure locations.
- When several transactions execute concurrently on the database, the consistency of data may no longer be preserved. It is therefore necessary for the system to control the interaction among the concurrent transactions.
 - Since a transaction is a unit that preserves consistency, a serial execution of transactions guarantees that consistency is preserved.
 - A *schedule* captures the key actions of transactions that affect concurrent execution, such as **read** and **write** operations, while abstracting away internal details of the execution of the transaction.
 - We require that any schedule produced by concurrent processing of a set of transactions will have an effect equivalent to a schedule produced when these transactions are run serially in some order.
 - A system that guarantees this property is said to ensure *serializability*.
 - There are several different notions of equivalence leading to the concepts of *conflict serializability* and *view serializability*.
- Serializability of schedules generated by concurrently executing transactions can be ensured through one of a variety of mechanisms called *concurrency-control* policies.
- We can test a given schedule for conflict serializability by constructing a *precedence graph* for the schedule and by searching for the absence of cycles in the graph. However, there are more efficient concurrency-control policies for ensuring serializability.
- Schedules must be recoverable, to make sure that if transaction *a* sees the effects of transaction *b*, and *b* then aborts, then *a* also gets aborted.
- Schedules should preferably be cascadeless, so that the abort of a transaction does not result in cascading aborts of other transactions. Cascadelessness is ensured by allowing transactions to only read committed data.
- The concurrency-control management component of the database is responsible for handling the concurrency-control policies. Techniques include locking, timestamp ordering, and snapshot isolation. Chapter 18 describes concurrency-control policies.

- Database systems offer isolation levels weaker than serializability to allow less restriction of concurrency and thus improved performance. This introduces a risk of inconsistency that some applications find acceptable.
- Ensuring correct concurrent execution in the presence of SQL **update**, **insert**, and **delete** operations requires additional care due to the phantom phenomenon.

Review Terms

- Transaction
 - Atomicity
 - Consistency
 - Isolation
 - Durability
- Inconsistent state
- Storage types
 - Volatile storage
 - Non-volatile storage
 - Stable storage
- Concurrency-control system
- Recovery system
- Transaction state
 - Active
 - Partially committed
 - Failed
 - Aborted
 - Committed
 - Terminated
- compensating transaction
- Transaction
 - Restart
 - Kill
- Observable external writes
- Concurrent executions
- Serial execution
- Schedules
- Conflict of operations
- Conflict equivalence
- Conflict serializability
- Serializability testing
- Precedence graph
- Serializability order
- Recoverable schedules
- Cascading rollback
- Cascadeless schedules
- Isolation levels
 - Serializable
 - Repeatable read
 - Read committed
 - Read uncommitted
- Dirty writes
- Automatic commit
- Concurrency control
- Locking
- Timestamp ordering
- Snapshot isolation
- Phantom phenomenon
- Predicate locking

Practice Exercises

- 17.1 Suppose that there is a database system that never fails. Is a recovery manager required for this system?
- 17.2 Consider a file system such as the one on your favorite operating system.
 - a. What are the steps involved in the creation and deletion of files and in writing data to a file?
 - b. Explain how the issues of atomicity and durability are relevant to the creation and deletion of files and to writing data to files.
- 17.3 Database-system implementers have paid much more attention to the ACID properties than have file-system implementers. Why might this be the case?
- 17.4 What class or classes of storage can be used to ensure durability? Why?
- 17.5 Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?
- 17.6 Consider the precedence graph of Figure 17.16. Is the corresponding schedule conflict serializable? Explain your answer.
- 17.7 What is a cascadeless schedule? Why is cascadelessness of schedules desirable? Are there any circumstances under which it would be desirable to allow noncascadeless schedules? Explain your answer.
- 17.8 The **lost update** anomaly is said to occur if a transaction T_j reads a data item, then another transaction T_k writes the data item (possibly based on a previous read), after which T_j writes the data item. The update performed by T_k has been lost, since the update done by T_j ignored the value written by T_k .

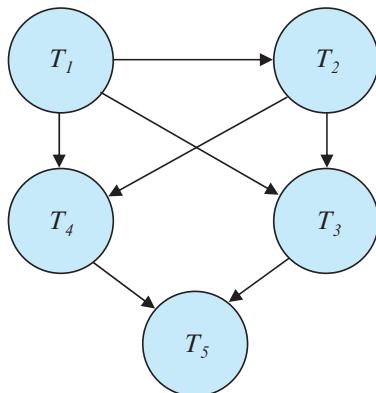


Figure 17.16 Precedence graph for Practice Exercise 17.6.

- a. Give an example of a schedule showing the lost update anomaly.
 - b. Give an example schedule to show that the lost update anomaly is possible with the **read committed** isolation level.
 - c. Explain why the lost update anomaly is not possible with the **repeatable read** isolation level.
- 17.9** Consider a database for a bank where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs that would present a problem for the bank.
- 17.10** Consider a database for an airline where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs, but the airline may be willing to accept it in order to gain better overall performance.
- 17.11** The definition of a schedule assumes that operations can be totally ordered by time. Consider a database system that runs on a system with multiple processors, where it is not always possible to establish an exact ordering between operations that executed on different processors. However, operations on a data item can be totally ordered.
 Does this situation cause any problem for the definition of conflict serializability? Explain your answer.

Exercises

- 17.12** List the ACID properties. Explain the usefulness of each.
- 17.13** During its execution, a transaction passes through several states, until it finally commits or aborts. List all possible sequences of states through which a transaction may pass. Explain why each state transition may occur.
- 17.14** Explain the distinction between the terms *serial schedule* and *serializable schedule*.
- 17.15** Consider the following two transactions:

```

 $T_{13}$ : read( $A$ );
    read( $B$ );
    if  $A = 0$  then  $B := B + 1$ ;
    write( $B$ ).
 $T_{14}$ : read( $B$ );
    read( $A$ );
    if  $B = 0$  then  $A := A + 1$ ;
    write( $A$ ).
  
```

Let the consistency requirement be $A = 0 \vee B = 0$, with $A = B = 0$ as the initial values.

- a. Show that every serial execution involving these two transactions preserves the consistency of the database.
 - b. Show a concurrent execution of T_{13} and T_{14} that produces a nonserializable schedule.
 - c. Is there a concurrent execution of T_{13} and T_{14} that produces a serializable schedule?
- 17.16** Give an example of a serializable schedule with two transactions such that the order in which the transactions commit is different from the serialization order.
- 17.17** What is a recoverable schedule? Why is recoverability of schedules desirable? Are there any circumstances under which it would be desirable to allow non-recoverable schedules? Explain your answer.
- 17.18** Why do database systems support concurrent execution of transactions, despite the extra effort needed to ensure that concurrent execution does not cause any problems?
- 17.19** Explain why the read-committed isolation level ensures that schedules are cascade-free.
- 17.20** For each of the following isolation levels, give an example of a schedule that respects the specified level of isolation but is not serializable:
- a. Read uncommitted
 - b. Read committed
 - c. Repeatable read
- 17.21** Suppose that in addition to the operations `read` and `write`, we allow an operation `pred_read(r, P)`, which reads all tuples in relation r that satisfy predicate P .
- a. Give an example of a schedule using the `pred_read` operation that exhibits the phantom phenomenon and is nonserializable as a result.
 - b. Give an example of a schedule where one transaction uses the `pred_read` operation on relation r and another concurrent transaction deletes a tuple from r , but the schedule does not exhibit a phantom conflict. (To do so, you have to give the schema of relation r and show the attribute values of the deleted tuple.)

Further Reading

[Gray and Reuter (1993)] provides detailed textbook coverage of transaction-processing concepts, techniques, and implementation details, including concurrency control and recovery issues. [Bernstein and Newcomer (2009)] provides textbook coverage of various aspects of transaction processing.

The concept of serializability was formalized by [Eswaran et al. (1976)] in connection with work on concurrency control for System R.

References covering specific aspects of transaction processing, such as concurrency control and recovery, are cited in Chapter 18 and Chapter 19.

Bibliography

[Bernstein and Newcomer (2009)] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, 2nd edition, Morgan Kaufmann (2009).

[Eswaran et al. (1976)] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, “The Notions of Consistency and Predicate Locks in a Database System”, *Communications of the ACM*, Volume 19, Number 11 (1976), pages 624–633.

[Gray and Reuter (1993)] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

Credits

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

CHAPTER 17



Transactions

Practice Exercises

- 17.1 Suppose that there is a database system that never fails. Is a recovery manager required for this system?

Answer:

Even in this case the recovery manager is needed to perform rollback of aborted transactions for cases where the transaction itself fails.

- 17.2 Consider a file system such as the one on your favorite operating system.

- a. What are the steps involved in the creation and deletion of files and in writing data to a file?
- b. Explain how the issues of atomicity and durability are relevant to the creation and deletion of files and to writing data to files.

Answer:

There are several steps in the creation of a file. A storage area is assigned to the file in the file system. (In UNIX, a unique i-number is given to the file and an i-node entry is inserted into the i-list.) Deletion of file involves exactly opposite steps.

For the file system user, durability is important for obvious reasons, but atomicity is not relevant generally as the file system doesn't support transactions. To the file system implementor, though, many of the internal file system actions need to have transaction semantics. All steps involved in creation/deletion of the file must be atomic, otherwise there will be unreferenceable files or unusable areas in the file system.

- 17.3 Database-system implementers have paid much more attention to the ACID properties than have file-system implementers. Why might this be the case?

Answer:

Database systems usually perform crucial tasks whose effects need to be atomic and durable, and whose outcome affects the real world in a permanent manner. Examples of such tasks are monetary transactions, seat bookings etc. Hence the ACID properties have to be ensured. In contrast, most users of file systems would not be willing to pay the price (monetary, disk space, time) of supporting ACID properties.

- 17.4** What class or classes of storage can be used to ensure durability? Why?

Answer:

Only stable storage ensures true durability. Even nonvolatile storage is susceptible to data loss, albeit less so than volatile storage. Stable storage is only an abstraction. It is approximated by redundant use of nonvolatile storage in which data are not only replicated but distributed physically to reduce to near zero the chance of a single event causing data loss.

- 17.5** Since every conflict-serializable schedule is view serializable, why do we emphasize conflict serializability rather than view serializability?

Answer:

Most of the concurrency control protocols (protocols for ensuring that only serializable schedules are generated) used in practice are based on conflict serializability—they actually permit only a subset of conflict serializable schedules. The general form of view serializability is very expensive to test, and only a very restricted form of it is used for concurrency control.

- 17.6** Consider the precedence graph of Figure 17.16. Is the corresponding schedule conflict serializable? Explain your answer.

Answer:

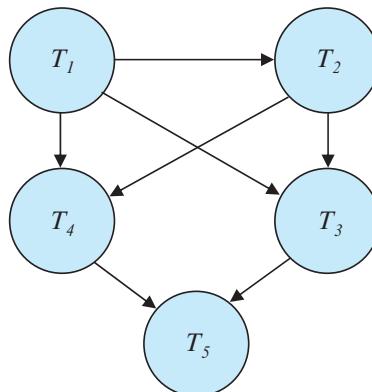


Figure 17.16 Precedence graph for Practice Exercise 17.6.

There is a serializable schedule corresponding to the precedence graph since the graph is acyclic. A possible schedule is obtained by doing a topological sort, that is, T_1, T_2, T_3, T_4, T_5 .

- 17.7 What is a cascadeless schedule? Why is cascadelessness of schedules desirable? Are there any circumstances under which it would be **desirable** to allow noncascadeless schedules? Explain your answer.

Answer:

A cascadeless schedule is one where, for each pair of transactions T_i and T_j such that T_j reads data items previously written by T_i , the commit operation of T_i appears before the read operation of T_j . Cascadeless schedules are desirable because the failure of a transaction does not lead to the aborting of any other transaction. Of course this comes at the cost of less concurrency. If failures occur rarely, so that we can pay the price of cascading aborts for the increased concurrency, noncascadeless schedules might be desirable.

- 17.8 The **lost update** anomaly is said to occur if a transaction T_j reads a data item, then another transaction T_k writes the data item (possibly based on a previous read), after which T_j writes the data item. The update performed by T_k has been lost, since the update done by T_j ignored the value written by T_k .

- Give an example of a schedule showing the lost update anomaly.
- Give an example schedule to show that the lost update anomaly is possible with the **read committed** isolation level.
- Explain why the lost update anomaly is not possible with the **repeatable read** isolation level.

Answer:

- A schedule showing the lost update anomaly:

T_1	T_2
read(A)	
	read(A) write(A)

In the above schedule, the value written by the transaction T_2 is lost because of the write of the transaction T_1 .

- Lost update anomaly in read-committed isolation level:

T_1	T_2
lock-S(A)	
read(A)	
unlock(A)	
.	
	lock-X(A)
	read(A)
	write(A)
	unlock(A)
	commit
lock-X(A)	
write(A)	
unlock(A)	
commit	

The locking in the above schedule ensures the read-committed isolation level. The value written by transaction T_2 is lost due to T_1 's write.

- c. Lost update anomaly is not possible in repeatable read isolation level. In repeatable read isolation level, a transaction T_1 reading a data item X holds a shared lock on X till the end. This makes it impossible for a newer transaction T_2 to write the value of X (which requires X-lock) until T_1 finishes. This forces the serialization order T_1, T_2 , and thus the value written by T_2 is not lost.

- 17.9 Consider a database for a bank where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs that would present a problem for the bank.

Answer:

Suppose that the bank enforces the integrity constraint that the sum of the balances in the checking and the savings account of a customer must not be negative. Suppose the checking and savings balances for a customer are \$100 and \$200 respectively.

Suppose that transaction T_1 withdraws \$200 from the checking account after verifying the integrity constraint by reading both the balances. Suppose that concurrent transaction T_2 withdraws \$200 from the checking account after verifying the integrity constraint by reading both the balances.

Since each of the transactions checks the integrity constraints on its own snapshot, if they run concurrently, each will believe that the sum of the balances after the withdrawal is \$100, and therefore its withdrawal does not violate the integrity constraint. Since the two transactions update different data items, they do not have any update conflict, and under snapshot isolation both

of them can commit. This is a nonserializable execution which results into a serious problem.

- 17.10** Consider a database for an airline where the database system uses snapshot isolation. Describe a particular scenario in which a nonserializable execution occurs, but the airline may be willing to accept it in order to gain better overall performance.

Answer:

Consider a web-based airline reservation system. There could be many concurrent requests to see the list of available flights and available seats in each flight and to book tickets. Suppose there are two users *A* and *B* concurrently accessing this web application, and only one seat is left on a flight.

Suppose that both user *A* and user *B* execute transactions to book a seat on the flight and suppose that each transaction checks the total number of seats booked on the flight, and inserts a new booking record if there are enough seats left. Let T_3 and T_4 be their respective booking transactions, which run concurrently. Now T_3 and T_4 will see from their snapshots that one ticket is available and will insert new booking records. Since the two transactions do not update any common data item (tuple), snapshot isolation allows both transactions to commit. This results in an extra booking, beyond the number of seats available on the flight.

However, this situation is usually not very serious since cancellations often resolve the conflict; even if the conflict is present at the time the flight is to leave, the airline can arrange a different flight for one of the passengers on the flight, giving incentives to accept the change. Using snapshot isolation improves the overall performance in this case since the booking transactions read the data from their snapshots only and do not block other concurrent transactions.

- 17.11** The definition of a schedule assumes that operations can be totally ordered by time. Consider a database system that runs on a system with multiple processors, where it is not always possible to establish an exact ordering between operations that executed on different processors. However, operations on a data item can be totally ordered.

Does this situation cause any problem for the definition of conflict serializability? Explain your answer.

Answer:

The given situation will not cause any problem for the definition of conflict serializability since the ordering of operations on each data item is necessary for conflict serializability, whereas the ordering of operations on different data items is not important.

T_1	T_2
read(A)	
write(B)	read(B)

For the above schedule to be conflict serializable, the only ordering requirement is **read(B) -> write(B)**. **read(A)** and **read(B)** can be in any order.

Therefore, as long as the operations on a data item can be totally ordered, the definition of conflict serializability should hold on the given multiprocessor system.