CHAPTER 2

Introduction to the Relational Model

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

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

2.1 Structure of Relational Databases

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

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

Figure 2.1 The *instructor* relation.

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

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

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

Figure 2.2 The *course* relation.

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

Figure 2.3 The *prereq* relation.

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

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

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

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

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

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

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

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

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

Figure 2.4 Unsorted display of the *instructor* relation.

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

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

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

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

2.2 Database Schema

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

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

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

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

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

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

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

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

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

Figure 2.5 The *department* relation.

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

Figure 2.6 The *section* relation.

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

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

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

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

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

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

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

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

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

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

Figure 2.7 The *teaches* relation.

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

2.3 Keys

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

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

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

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

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

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

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

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

Consider the *classroom* relation:

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

Figure 2.8 Schema of the university database.

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

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

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

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

2.4 Schema Diagrams

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

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

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

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

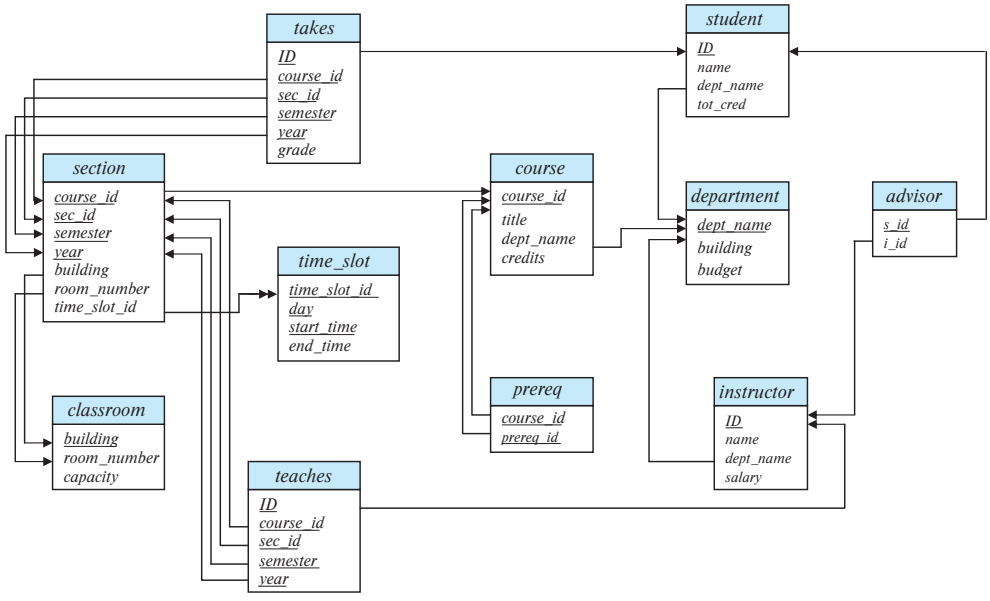

Figure 2.9 Schema diagram for the university database.

2.5 Relational Query Languages

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

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

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

There are a number of "pure" query languages.

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

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

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

2.6 The Relational Algebra

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

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

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

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

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

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

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

2.6.1 The Select Operation

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

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

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

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

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

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

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

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

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

2.6.2 The Project Operation

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

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

Figure 2.11 shows the relation that results from this query.

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

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

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

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

to get the monthly salary of each instructor.

2.6.3 Composition of Relational Operations

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

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

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

In general, since the result of a relational-algebra operation is of the same type (relation) as its inputs, relational-algebra operations can be composed together into a **relational-algebra expression**. Composing relational-algebra operations into relational-algebra expressions is just like composing arithmetic operations (such as $+$, $-$, $*$, and $\div$) 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 $r_1 \times r_2$.

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

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

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

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

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

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

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

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

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

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

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

2.6.5 The Join Operation

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

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

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

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

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

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

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

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

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

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

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

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

2.6.6 Set Operations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

course_id
CS-101

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

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

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

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

The result relation for this query appears in Figure 2.15.

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

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

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

The result relation for this query appears in Figure 2.16.

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

2.6.7 The Assignment Operation

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

course_id
CS-347
PHY-101

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

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

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

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

2.6.8 The Rename Operation

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

$$\rho_x (E)$$

returns the result of expression E under the name x.

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

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

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

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

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

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

Note 2.1 OTHER RELATIONAL OPERATIONS

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

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

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

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

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

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

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

2.6.9 Equivalent Queries

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

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

Now consider an alternative query:

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

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

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

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

2.7 Summary

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

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

Review Terms

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

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

Figure 2.17 Employee database.

Practice Exercises

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2.18 Bank database.

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

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

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

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

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

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

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

Given the above definition:

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

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

Exercises

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

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

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

a. What are the appropriate primary keys?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Further Reading

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

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

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

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

Bibliography

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

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

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

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

Credits

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

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** $(A_{j_1}, A_{j_2}, \ldots, A_{j_m})$: The **primary-key** specification says that attributes $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$ 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** $(A_{k_1}, A_{k_2}, \ldots, A_{k_n})$ **references** *s*: The **foreign key** specification says that the values of attributes $(A_{k_1}, A_{k_2}, \ldots, A_{k_n})$ 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.

name
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.

dept_name
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:

name
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 <, <=, >, >=, =, and <>. 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-*

name	*dept_name*	*building*
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 $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
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 t1 in relation r1
        for each tuple t2 in relation r2
                ...
                for each tuple tm in relation rm
                        Concatenate t1, t2, ..., tm 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.

[2] In 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.

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

Figure 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*;

name	*course_id*
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 MULTISET 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_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
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 $t_1.t_2$ 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 $A_1, A_2, \ldots, A_n$
from $r_1, r_2, \ldots, r_m$
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:

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

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**.

[3]Early 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.

[4] Except 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 $(v_1, v_2, \ldots, v_n)$ to denote a tuple of arity n containing values $v_1, v_2, \ldots, v_n$; 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) <= (b_1, b_2) is true if a_1 <= b_1 **and** a_2 <= 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';
```

course_id
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 ∪, ∩, 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.

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

course_id
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

course_id
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.

course_id
CS-101

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

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

course_id
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

[7] Some 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 "$1 <$ **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** $(1 <$ **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',null), ('A',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.

[8]The **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-

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

plicates 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;
```

ID	*name*	*dept_name*	*salary*
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.

dept_name	*avg_salary*
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."

dept_name	*avg_salary*
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 MULTISET 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 ${}_{dept\_name}\gamma_{\textbf{average}(salary)}(instructor)$ 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, $\gamma_{\textbf{average}(salary)}(instructor)$ 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:

$$ {}_{dept\_name}\gamma_{\textbf{average}(salary)\ \textbf{as}\ \text{avg\_salary}}(instructor) $$

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

select A_1, A_2, **sum**(A_3)
from r_1, r_2, ..., r_m
where P
group by A_1, A_2 **having count**(A_4) > 2

is equivalent to:

$$ t1 \leftarrow \sigma_P(r_1 \times r_2 \times \cdots \times r_m) $$
$$ \Pi_{A_1, A_2, SumA3}(\sigma_{countA4 > 2}({}_{A_1, A_2}\gamma_{\text{sum}(A_3)\ \textbf{as}\ SumA3,\ \textbf{count}(A_4)\ \textbf{as}\ countA4}(t1))) $$

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 the 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**.

[10]The 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** construct[11] 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.

[11] This 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;
```

[12] The 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.

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

Note 3.3 SQL AND MULTISET 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 $\ltimes$, and *antijoin*, denoted $\overline{\ltimes}$, which are supported internally by many database implementations (the symbol $\triangleright$ is sometimes used in place of $\overline{\ltimes}$ to denote antijoin). For example, given relations r and s, $r \ltimes_{r.A=s.B} s$ 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, $r \overline{\ltimes}_{r.A=s.B} s$ 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 **select**s). 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 $pred_1, pred_2, \ldots, pred_n$ that is satisfied; if none of the predicates is satisfied, the operation returns $result_0$. 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 *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 ≤ *score* < 60, grade *B* if 60 ≤ *score* < 80, and grade *A* if 80 ≤ *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.

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".

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:

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.

Exercises

3.11 Write the following queries in SQL, using the university schema.

a. 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.

b. Find the ID and name of each student who has not taken any course offered before 2017.

c. For each department, find the maximum salary of instructors in that department. You may assume that every department has at least one instructor.

d. 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:

a. Create a new course "CS-001", titled "Weekly Seminar", with 0 credits.

b. Create a section of this course in Fall 2017, with *sec_id* of 1, and with the location of this section not yet specified.

c. Enroll every student in the Comp. Sci. department in the above section.

d. Delete enrollments in the above section where the student's ID is 12345.

e. Delete the course CS-001. What will happen if you run this **delete** statement without first deleting offerings (sections) of this course?

f. 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.

a. Find the number of accidents involving a car belonging to a person named "John Smith".

b. 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.

a. Find each customer who has an account at *every* branch located in "Brooklyn".

b. Find the total sum of all loan amounts in the bank.

c. 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.

a. Find ID and name of each employee who lives in the same city as the location of the company for which the employee works.

b. Find ID and name of each employee who lives in the same city and on the same street as does her or his manager.

c. Find ID and name of each employee who earns more than the average salary of all employees of her or his company.

d. 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.

a. Give all employees of "First Bank Corporation" a 10 percent raise.

b. Give all managers of "First Bank Corporation" a 10 percent raise.

c. 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.

a. Find the member number and name of each member who has borrowed at least one book published by "McGraw-Hill".

b. Find the member number and name of each member who has borrowed every book published by "McGraw-Hill".

c. For each publisher, find the member number and name of each member who has borrowed more than five books of that publisher.

d. 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 Nesvadba/Shutterstock.

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.

ID	*name*	*dept_name*	*tot_cred*
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.

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
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	*null*

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]

[1] For 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*".

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
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	*null*

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 $A_1, A_2, \ldots, A_n$
from r_1 **natural join** r_2 **natural join** $\ldots$ **natural join** r_m
where P;

More generally, a **from** clause can be of the form

[2]As 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 $E_1, E_2, \ldots, E_n$

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 **join** r_2 **using**(A_1, A_2). The operation is similar to r_1 **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).

[3]We show null values in tables using *null*, but most systems display null values as a blank field.

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
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

[4]In those systems, notably MySQL, that implement only left and right outer join, this is exactly how one has to write a full outer join.

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*	*name*	*dept_name*	*tot_cred*
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	*null*	*null*	*null*	*null*	*null*	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	*null*	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.

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	*null*	*null*	*null*	*null*	*null*
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	*null*	*null*	*null*	*null*	*null*
76653	*null*	*null*	*null*	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*, *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.

[5] Some 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 MULTISET RELATIONAL ALGEBRA - PART 4

The relational algebra supports the left outer-join operation, denoted by $\mathbin{⟕}_\theta$, the right outer-join operation, denoted by $\mathbin{⟖}_\theta$, and the full outer-join operation, denoted by $\mathbin{⟗}_\theta$. 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 $\mathbin{⟕}$, $\mathbin{⟖}$, and $\mathbin{⟗}$. 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).

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

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 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.

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
69987	White	*null*	*null*

instructor

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

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]

[6]There 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.

[7] Oracle 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:

$$\textbf{unique}\ (A_{j_1}, A_{j_2}, \ldots, A_{j_m})$$

The **unique** specification says that attributes $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$ 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** (*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.

[8] Some 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]

[9] We 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.

[10] Many 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

[11] The 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.

[12]This 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 *d*atabase in place of the term *c*atalog.)

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 *dean*s. 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

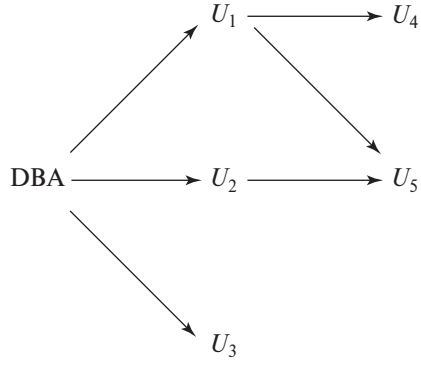

Figure 4.11 Authorization-grant graph (U_1, U_2, ..., U_5 are users and DBA refers to the database administrator).

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:

$$ID = \textbf{sys\_context}\ (\text{'USERENV'}, \text{'SESSION\_USER'})$$

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:

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.

b. Write the same query as in part a, but using a scalar subquery and not using outer join.

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 "—".

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.

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.

a. **select** * **from** *student* **natural left outer join** *takes*

b. **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**.

a. Give instances of relations r, s, and t such that in the result of
(r **natural left outer join** s) **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 **natural left outer join** (s **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.

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).

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 *r*1 with a foreign key referencing another relation *r*2. What authorization privilege does the user need on *r*2? 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 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.

[1]There are multiple versions of the getConnection() method, which differ in the parameters that they accept. We present the most commonly used version.

[2]Prior 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

[3]This 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' 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, HSQLBD, 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;
```

[4] If 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

[5]We discuss database backup and recovery from failures in detail in Chapter 19.

course_id	*prereq_id*
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

[6]This 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 union[8] 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

[7] Some systems treat the **recursive** keyword as optional; others disallow it.
[8] Some 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.

[9] Staring 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

[10] The 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;
```

[11] There 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 partition[12] 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.

[12]The 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 **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, **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;
```

[13] If 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.

[14] We 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 since 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;
```

item_name	*color*	*clothes_size*	*quantity*
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.

item_name	*clothes_size*	*dark*	*pastel*	*white*
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"))

[15] Some 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:

sales (*item_name*, *color*, *clothes_size*, *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

item_name	*color*	*quantity*
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	*null*	53
dress	*null*	35
shirt	*null*	49
pants	*null*	27
null	*null*	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.

[16]The 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 **rollup**s and **cube**s 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, cust omer_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

part(*part_id*, *name*, *cost*)
subpart(*part_id*, *subpart_id*, *count*)

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 prodedure 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

prereq_depth(*course_id*, *prereq_id*, *depth*)

building	*room_number*	*time_slot_id*	*course_id*	*sec_id*
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.