This page intentionally left blank

CHAPTER 3

Introduction to SQL

There are a number of database query languages in use, either commercially or experimentally. In this chapter, as well as in Chapters 4 and 5, we study the most widely used 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.

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, and most recently SQL:2008. The bibliographic notes provide references to these standards.

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 ending 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, which were introduced in SQL:1999, and subsequent versions of SQL. Later, in Chapter 22, we outline object-oriented extensions to SQL, which were introduced in SQL:1999.

Although most SQL implementations support the standard features we describe here, you should be aware that there are differences between implementations. Most implementations support some nonstandard features, while omitting support for some of the more advanced 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 must be specified to the system by means of 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 Chapters 4 and 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 or 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 store a string "Avi" in this attribute, 7 spaces are appended to the string to make it 10 characters long. In contrast, if attribute B were of type **varchar**(10), and we store "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 added 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, 2 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
(A_1 D_1,
A_2 D_2,
. . . ,
A_n D_n,
⟨integrity-constraint$_1$⟩,
. . . ,
⟨integrity-constraint$_k$⟩);

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

- **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. We can use the **insert** command to load data into the relation. For example, if we wish to insert the fact that there is an instructor named Smith in the Biology department with *instructor_id* 10211 and a salary of $66,000, we write:

```
insert into instructor
        values (10211, 'Smith', 'Biology', 66000);
```

The values are specified in the *order* in which the corresponding attributes are listed in the relation schema. The insert command has a number of useful features, and is covered in more detail later, in Section 3.9.2.

We can use the **delete** command to delete tuples from a relation. The command

```
delete from student;
```

```
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.

would delete all tuples from the *student* relation. Other forms of the delete command allow specific tuples to be deleted; the delete command is covered in more detail later, in Section 3.9.1.

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

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**. The 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 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

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 3.2 Result of "**select** *name* **from** *instructor*".

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.

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 relations as well as in the results of SQL expressions. 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*;

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*".

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

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.

An 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 Figures 2.1 and 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

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."

department.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**.[1]

The **from** clause by itself defines a Cartesian product of the relations listed in the clause. It is defined formally in terms of set theory, but is perhaps best understood as an iterative process that generates tuples for the result relation of the **from** clause.

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

The result relation has all attributes from all the relations in the **from** clause. Since the same attribute name may appear in both r_i and r_j, as we saw earlier, we prefix the 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.[2]

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.

[1] 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 Chapters 12 and 13.

[2] Note that we renamed *instructor.ID* as *inst.ID* to reduce the width of the table in Figure 3.6.

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Physics	95000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Physics	95000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Physics	95000	10101	FIN-201	1	Spring	2010
10101	Srinivasan	Physics	95000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Physics	95000	10101	CS-101	1	Fall	2009
12121	Wu	Physics	95000	10101	CS-315	1	Spring	2010
12121	Wu	Physics	95000	10101	CS-347	1	Fall	2009
12121	Wu	Physics	95000	10101	FIN-201	1	Spring	2010
12121	Wu	Physics	95000	15151	MU-199	1	Spring	2010
12121	Wu	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Physics	95000	10101	CS-101	1	Fall	2009
15151	Mozart	Physics	95000	10101	CS-315	1	Spring	2010
15151	Mozart	Physics	95000	10101	CS-347	1	Fall	2009
15151	Mozart	Physics	95000	10101	FIN-201	1	Spring	2010
15151	Mozart	Physics	95000	15151	MU-199	1	Spring	2010
15151	Mozart	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2009
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2010
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2009
22222	Einstein	Physics	95000	10101	FIN-201	1	Spring	2010
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

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 expect 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;
```

Note that the above 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 4.1.2.

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 the above result.

If we only wished to find 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:

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."

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.

The above 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 later, in Chapters 12 and 13.

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 would 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 3 courses, so we have 600 tuples in the *teaches* relation. Then the above iterative process generates $200 * 600 = 120{,}000$ tuples in the result.

3.3.3 The Natural Join

In our example query that combined information from the *instructor* and *teaches* table, the matching condition required *instructor.ID* to be equal to *teaches.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.

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 *instructor* and *teaches*, computing *instructor* **natural join** *teaches* considers only those pairs of tuples where both the tuple from *instructor* and the tuple from *teaches* have the same value on the common attribute, *ID*.

ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Figure 3.8 The natural join of the *instructor* relation with the *teaches* relation.

The result relation, shown in Figure 3.8, has only 13 tuples, the ones that give information about an instructor and a course that that instructor actually teaches. 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.

Consider 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", which we wrote earlier as:

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

This query can be written more concisely using the natural-join operation in SQL as:

select *name*, *course_id*
from *instructor* **natural join** *teaches*;

Both of the above queries generate the same result.

As we saw earlier, the result of the natural join operation is a relation. Conceptually, expression "*instructor* **natural join** *teaches*" in the **from** clause is replaced

by the relation obtained by evaluating the natural join.[3] The **where** and **select** clauses are then evaluated on this relation, as we saw earlier 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** ... **natural join** r_m
where P;

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

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 instructors along with the the titles of courses that they teach." The query can be written in SQL as follows:

select *name*, *title*
from *instructor* **natural join** *teaches*, *course*
where *teaches.course_id*= *course.course_id*;

The natural join of *instructor* and *teaches* 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 *teaches.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 *teaches* relation.

In contrast the following SQL query does *not* compute the same result:

select *name*, *title*
from *instructor* **natural join** *teaches* **natural join** *course*;

To see why, note that the natural join of *instructor* and *teaches* contains the attributes (*ID*, *name*, *dept_name*, *salary*, *course_id*, *sec_id*), while the *course* relation contains the attributes (*course_id*, *title*, *dept_name*, *credits*). As a result, the natural join of these two would require that the *dept_name* attribute values from the two inputs be the same, in addition to requiring that the *course_id* values be the same. This query would then omit all (instructor name, course title) pairs where the instructor teaches a course in a department other than the instructor's own department. The previous query, on the other hand, correctly outputs such pairs.

[3] As a consequence, it is not possible to use attribute names containing the original relation names, for instance *instructor.name*or *teaches.course_id*, to refer to attributes in the natural join result; we can, however, use attribute names such as *name* and *course_id*, without the relation names.

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 (*instructor* **natural join** *teaches*) **join** *course* **using** (*course_id*);

The operation **join** . . . **using** requires a list of attribute names to be specified. Both inputs 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 *teaches.dept_name* and *course.dept_name* to differ, and the SQL query gives the correct answer.

3.4 Additional Basic Operations

There are number of additional basic operations that 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 used 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:

old-name **as** *new-name*

The **as** clause can appear in both the **select** and **from** clauses.[4]

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 is also commonly referred to as a **table alias**, or a **correlation variable**, or a **tuple variable**.

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 can in fact be expressed directly in SQL as we shall see in Section 3.8.2.

[4]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.

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 databases. 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; 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.

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 databases provide variants of the **like** operation which do not distinguish lower and upper case.

SQL:1999 also offers a **similar to** operation, which 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 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.

We can extend the preceding query that finds instructor names along with course identifiers, which we saw earlier, and consider a more complicated case in which we require also that the instructors be from the Biology department: "Find the instructor names and the courses they taught for all instructors in the Biology department who have taught some course." To write this query, we can modify either of the SQL queries we saw earlier, by adding an extra condition in the **where** clause. We show below the modified form of the SQL query that does not use natural join.

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

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 comparison operators can be used on tuples, and the ordering is defined lexicographically. For example, $(a_1, a_2) <= (b_1, b_2)$

course_id
CS-101
CS-347
PHY-101

Figure 3.9 The *c1* relation, listing courses taught in Fall 2009.

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 preceding SQL query 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-theory operations $\cup$, $\cap$, and $-$. We shall now construct queries involving the **union**, **intersect**, and **except** operations over two sets.

- The set of all courses taught in the Fall 2009 semester:

select *course_id*
from *section*
where *semester* = 'Fall' **and** *year*= 2009;

- The set of all courses taught in the Spring 2010 semester:

select *course_id*
from *section*
where *semester* = 'Spring' **and** *year*= 2010;

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 Figures 3.9 and 3.10. Observe that *c2* contains two tuples corresponding to *course_id* CS-319, since two sections of the course have been offered in Spring 2010.

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

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

Figure 3.10 The *c2* relation, listing courses taught in Spring 2010.

3.5.1 The Union Operation

To find the set of all courses taught either in Fall 2009 or in Spring 2010, or both, we write:[6]

```
(select course_id
 from section
 where semester = 'Fall' and year= 2009)
union
(select course_id
 from section
 where semester = 'Spring' and year= 2010);
```

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 2010, and a section of CS-101 is offered in the Fall 2009 as well as in the Fall 2010 semester, CS-101 and CS-319 appear only once in the result, shown in Figure 3.11.

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= 2009)
union all
(select course_id
 from section
 where semester = 'Spring' and year= 2010);
```

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 be listed twice. As a further example, if it were the case that 4 sections of ECE-101 were taught in the Fall 2009 semester and 2 sections of ECE-101

[6]The parentheses we include around each **select-from-where** statement are optional, but useful for ease of reading.

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

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

were taught in the Fall 2010 semester, then there would be 6 tuples with ECE-101 in the result.

3.5.2 The Intersect Operation

To find the set of all courses taught in the Fall 2009 as well as in Spring 2010 we write:

```
(select course_id
 from section
 where semester = 'Fall' and year= 2009)
intersect
(select course_id
 from section
 where semester = 'Spring' and year= 2010);
```

The result relation, shown in Figure 3.12, contains only one tuple with CS-101. The **intersect** operation automatically eliminates duplicates. For example, if it were the case that 4 sections of ECE-101 were taught in the Fall 2009 semester and 2 sections of ECE-101 were taught in the Spring 2010 semester, then there would be only 1 tuple with ECE-101 in the result.

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

course_id
CS-101

Figure 3.12 The result relation for *c1* intersect *c2*.

```
(select course_id
 from section
 where semester = 'Fall' and year= 2009)
intersect all
(select course_id
 from section
 where semester = 'Spring' and year= 2010);
```

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 4 sections of ECE-101 were taught in the Fall 2009 semester and 2 sections of ECE-101 were taught in the Spring 2010 semester, then there would be 2 tuples with ECE-101 in the result.

3.5.3 The Except Operation

To find all courses taught in the Fall 2009 semester but not in the Spring 2010 semester, we write:

```
(select course_id
 from section
 where semester = 'Fall' and year= 2009)
except
(select course_id
 from section
 where semester = 'Spring' and year= 2010);
```

The result of this query is shown in Figure 3.13. Note that this is exactly relation *c1* of Figure 3.9 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 performs set difference. The operation automatically eliminates duplicates in the inputs before performing set difference. For example, if 4 sections of ECE-101 were taught in the Fall 2009 semester and 2 sections of ECE-101 were taught in the Spring 2010 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= 2009)
except all
(select course_id
 from section
 where semester = 'Spring' and year= 2010);
```

[7]Some SQL implementations, notably Oracle, use the keyword **minus** in place of **except**.

course_id
CS-347
PHY-101

Figure 3.13 The result relation for *c1* except *c2*.

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 4 sections of ECE-101 were taught in the Fall 2009 semester and 2 sections of ECE-101 were taught in Spring 2010, then there are 2 tuples with ECE-101 in the result. If, however, there were two or fewer sections of ECE-101 in the the Fall 2009 semester, and two sections of ECE-101 in the Spring 2010 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.

Some implementations of SQL also allow 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**.

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 above 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.

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 built-in aggregate functions:

- 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 arbitrary name to the result relation attribute that is generated by aggregation; 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 balance 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 duplicates 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 2010 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 = 2010;
```

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;
```

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

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 wish 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*;

Figure 3.14 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 3.15.

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

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

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Figure 3.15 The result relation for the query "Find the average salary in each department".

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

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 2010 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* **natural join** *teaches*
where *semester* = 'Spring' **and** *year* = 2010
group by *dept_name*;

The result is shown in Figure 3.16.

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 must appear only inside an aggregate function if it appears in the **select** clause, 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:

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

Figure 3.16 The result relation for the query "Find the number of instructors in each department who teach a course in the Spring 2010 semester."

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

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.

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. 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 3.17.

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 treated as 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.

dept_name	*avg(avg_salary)*
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

Figure 3.17 The result relation for the query "Find the average salary of instructors in those departments where the average salary is more than $42,000."

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 2009, find the average total credits (*tot_cred*) of all students enrolled in the section, if the section had at least 2 students."

select *course_id*, *semester*, *year*, *sec_id*, **avg** (*tot_cred*)
from *takes* **natural join** *student*
where *year* = 2009
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 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**, which mean exactly what you would intuitively expect, can be applied on a collection of Boolean values.

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 Sections 3.8.1 through 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 2009 and Spring 2010 semesters." Earlier, we wrote such a query by intersecting two sets: the set of courses taught in Fall 2009 and the set of courses taught in Spring 2010. We can take the alternative approach of finding all courses that were taught in Fall 2009 and that are also members of the set of courses taught in Spring 2010. Clearly, 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 2010, and we write the subquery

```
(select course_id
 from section
 where semester = 'Spring' and year= 2010)
```

We then need to find those courses that were taught in the Fall 2009 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= 2009 and
      course_id in (select course_id
                    from section
                    where semester = 'Spring' and year= 2010);
```

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 2009 semester but not in the Spring 2010 semester, we can write:

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
course_id not in (select course_id
                  from section
                  where semester = 'Spring' and year= 2010);
```

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);
```

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**.[8]

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**.

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

[8]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.

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 2009 semester and in the Spring 2010 semester" in still another way:

```
select course_id
from section as S
where semester = 'Fall' and year= 2009 and
      exists (select *
              from section as T
              where semester = 'Spring' and year= 2010 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 distinct 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.

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[9] 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 2009" 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 = 2009);
```

Note that if a course is not offered in 2009, the subquery would return an empty result, and the **unique** predicate would evaluate to true on the empty set.

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

[9]This construct is not yet widely implemented.

```
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 = 2009);
```

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 2009" 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 = 2009);
```

Formally, the **unique** test on a relation is defined to fail if and only if the relation contains two 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.

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. However, some SQL implementations, notably Oracle, do not support renaming of the result relation in the **from** clause.

As another example, suppose we wish to find the maximum across all departments of the total salary at 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 **from** clause. However, 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 **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. Currently, only a few SQL implementations, such as IBM DB2, 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 defines the temporary relation *max_budget*, which is used in the immediately following query. The **with** clause, introduced in SQL:1999, is supported by many, but not all, database systems.

We could have written the above 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 a view definition 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, of course, 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;
```

The subquery in the above 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.

3.9 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 fail 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. Obviously, 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 4 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 an salary of $18,000.

It is important that we evaluate the **select** statement fully before we carry out any insertions. If we carry out some insertions even as the **select** statement is 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. Consider the query:

```
select student
from student
where tot_cred > 45;
```

Since the *tot_cred* value of student "3003" is not known, we cannot determine whether it is greater than 45.

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 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 *instructor* relation.

If a salary increase is to be paid only to instructors with 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 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 an over 8 percent raise.

SQL provides a **case** construct that we can use to perform both the 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 also useful in SQL update statements, where they can be used in the **set** clause. 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 not 'F' or null. To specify this update, we need to use a subquery in the **set** clause, as shown below:

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

Observe that the subquery uses a correlation variable *S* from the **update** statement. In case a student has not successfully completed any course, the above update 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 by 0; a better alternative is to replace the clause "**select sum**(*credits*)" in the preceding subquery by the following **select** clause using a **case** expression:

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

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, and support for the natural join operation.
- 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-theory 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
- Natural join operation
- **as** clause
- **order** by clause
- Correlation name (correlation variable, tuple variable)
- Set operations
  - **union**
  - **intersect**
  - **except**
- Null values
  - Truth value "unknown"
- 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
  - Deletion
  - Insertion
  - Updating

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 Autumn 2009.

f. Find the maximum enrollment, across all sections, in Autumn 2009.

g. Find the sections that had the maximum enrollment in Autumn 2009.

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

Figure 3.18 Insurance database for Exercises 3.4 and 3.14.

3.2 Suppose you are given a relation *grade_points*(*grade*, *points*), which 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 above relation, and our university schema, write each of the following queries in SQL. You can 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 every student.

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 (that is, 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 3.18, 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 2009.

b. Add a new accident to the database; assume any values for required attributes.

c. Delete the Mazda belonging to "John Smith".

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

Figure 3.19 Banking database for Exercises 3.8 and 3.15.

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, 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 distinct $p.a1$
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 one of $r1$ or $r2$ may be empty.

3.8 Consider the bank database of Figure 3.19, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find all customers of the bank who have an account but not a loan.

b. Find the names of all customers who live on the same street and in the same city as "Smith".

c. Find the names of all branches with customers who have an account in the bank and who live in "Harrison".

3.9 Consider the employee database of Figure 3.20, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find the names and cities of residence of all employees who work for "First Bank Corporation".

employee (*employee_name*, *street*, *city*)
works (*employee_name*, *company_name*, *salary*)
company (*company_name*, *city*)
manages (*employee_name*, *manager_name*)

Figure 3.20 Employee database for Exercises 3.9, 3.10, 3.16, 3.17, and 3.20.

b. Find the names, street addresses, and cities of residence of all employees who work for "First Bank Corporation" and earn more than $10,000.

c. Find all employees in the database who do not work for "First Bank Corporation".

d. Find all employees in the database who earn more than each employee of "Small Bank Corporation".

e. Assume that the companies may be located in several cities. Find all companies located in every city in which "Small Bank Corporation" is located.

f. Find the company that has the most employees.

g. Find those companies whose employees earn a higher salary, on average, than the average salary at "First Bank Corporation".

3.10 Consider the relational database of Figure 3.20. Give an expression in SQL for each of the following queries.

a. Modify the database so that "Jones" now lives in "Newtown".

b. Give all managers of "First Bank Corporation" a 10 percent raise unless the salary becomes greater than $100,000; 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 names of all students who have taken at least one Comp. Sci. course; make sure there are no duplicate names in the result.

b. Find the IDs and names of all students who have not taken any course offering before Spring 2009.

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 following queries in SQL, using the university schema.

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

b. Create a section of this course in Autumn 2009, with *sec_id* of 1.

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

d. Delete enrollments in the above section where the student's name is Chavez.

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 "database" 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 3.18. Make any reasonable assumptions about data types, and be sure to declare primary and foreign keys.

3.14 Consider the insurance database of Figure 3.18, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find the number of accidents in which the cars belonging to "John Smith" were involved.

b. Update the damage amount for the car with the license number "AABB2000" in the accident with report number "AR2197" to $3000.

3.15 Consider the bank database of Figure 3.19, where the primary keys are underlined. Construct the following SQL queries for this relational database.

a. Find all customers who have an account at *all* the branches located in "Brooklyn".

b. Find out 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 3.20, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find the names of all employees who work for "First Bank Corporation".

b. Find all employees in the database who live in the same cities as the companies for which they work.

c. Find all employees in the database who live in the same cities and on the same streets as do their managers.

d. Find all employees who earn more than the average salary of all employees of their company.

e. Find the company that has the smallest payroll.

3.17 Consider the relational database of Figure 3.20. 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 List two reasons why null values might be introduced into the database.

3.19 Show that, in SQL, <> **all** is identical to **not in**.

3.20 Give an SQL schema definition for the employee database of Figure 3.20. Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema.

3.21 Consider the library database of Figure 3.21. Write the following queries in SQL.

a. Print the names of members who have borrowed any book published by "McGraw-Hill".

b. Print the names of members who have borrowed all books published by "McGraw-Hill".

c. For each publisher, print the names of members who have borrowed more than five books of that publisher.

d. Print the average number of books borrowed per member. Take into account that if an member does not borrow any books, then that member does not appear in the *borrowed* relation at all.

3.22 Rewrite the **where** clause

where unique (**select** *title* **from** *course*)

without using the **unique** construct.

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

Figure 3.21 Library database for Exercise 3.21.

3.23 Consider the query:

```
select course_id, semester, year, sec_id, avg (tot_cred)
from takes natural join student
where year = 2009
group by course_id, semester, year, sec_id
having count (ID) >= 2;
```

Explain why joining *section* as well in the **from** clause would not change the result.

3.24 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.

Tools

A number of relational database systems are available commercially, including IBM DB2, IBM Informix, Oracle, Sybase, and Microsoft SQL Server. In addition several database systems can be downloaded and used free of charge, including PostgreSQL, MySQL (free except for certain kinds of commercial use), and Oracle Express edition.

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. Commercial IDEs for SQLthat work across multiple database platforms, include Embarcadero's RAD Studio and Aqua Data Studio.

For PostgreSQL, the pgAdmin tool provides GUI functionality, while for MySQL, phpMyAdmin provides GUI functionality. The NetBeans IDE provides a GUI front end that works with a number of different databases, but with limited functionality, while the Eclipse IDE supports similar functionality through several different plugins such as the Data Tools Platform (DTP) and JBuilder.

SQL schema definitions and sample data for the university schema are provided on the Web site for this book, db-book.com. The Web site also contains

instructions on how to set up and access some popular database systems. The SQL constructs discussed in this chapter are part of the SQL standard, but certain features are not supported by some databases. The Web site lists these incompatibilities, which you will need to take into account when executing queries on those databases.

Bibliographical Notes

The original version of SQL, called Sequel 2, is described by Chamberlin et al. [1976]. Sequel 2 was derived from the language Square (Boyce et al. [1975] and Chamberlin and Boyce [1974]). The American National Standard SQL-86 is described in ANSI [1986]. The IBM Systems Application Architecture definition of SQL is defined by IBM [1987]. The official standards for SQL-89 and SQL-92 are available as ANSI [1989] and ANSI [1992], respectively.

Textbook descriptions of the SQL-92 language include Date and Darwen [1997], Melton and Simon [1993], and Cannan and Otten [1993]. Date and Darwen [1997] and Date [1993a] include a critique of SQL-92 from a programming-languages perspective.

Textbooks on SQL:1999 include Melton and Simon [2001] and Melton [2002]. Eisenberg and Melton [1999] provide an overview of SQL:1999. Donahoo and Speegle [2005] covers SQL from a developers' perspective. Eisenberg et al. [2004] provides an overview of SQL:2003.

The SQL:1999, SQL:2003, SQL:2006 and SQL:2008 standards are published as a collection of ISO/IEC standards documents, which are described in more detail in Section 24.4. The standards documents are densely packed with information and hard to read, and of use primarily for database system implementers. The standards documents are available from the Web site http://webstore.ansi.org, but only for purchase.

Many database products support SQL features beyond those specified in the standard, and may not support some features of the standard. More information on these features may be found in the SQL user manuals of the respective products.

The processing of SQL queries, including algorithms and performance issues, is discussed in Chapters 12 and 13. Bibliographic references on these matters appear in those chapters.