CHAPTER 3

SQL

The relational algebra described in Chapter 2 provides a concise, formal notation for representing queries. However, commercial database systems require a query language that is more user friendly. In this chapter, as well as Chapter 4, we study SQL, the most influential commercially marketed query language. SQL uses a combination of relational-algebra (Chapter 2) and relational-calculus (Chapter 5) constructs.

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 Background

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; the most recent version is SQL:2003. 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.

- **Interactive data-manipulation language** (DML). The SQL DML includes a query language based on both the relational algebra (2) and and the tuple relational calculus (5). It includes also commands 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++, Java, PL/I, Cobol, Pascal, and Fortran.
- **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. Our description is mainly based on the widely implemented SQL-92 standard, but we also cover some extensions from the SQL:1999 and SQL:2003 standards.

In Chapter 4 we provide a more detailed coverage of the SQL type system, integrity constraints, and authorization. In that chapter, we also briefly outline embedded and dynamic SQL, including the ODBC and JDBC standards for interacting with a database from programs written in the C and Java languages. In Chapter 9, we outline object-oriented extensions to SQL that were introduced in SQL:1999.

Many database systems support most of the SQL-92 standard and some of the new constructs in SQL:1999 and SQL:2003, although currently no database system supports all the new constructs. You should also be aware that many database systems do not support some features of SQL-92, and that many databases provide nonstandard features that we do not cover here. 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.

The enterprise that we use in the examples in this chapter, and later chapters, is a banking enterprise. Figure 3.1 gives the relational schema that we use in our examples, with primary key attributes underlined. Recall that in Chapter 2 we first

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

Figure 3.1 Schema of banking enterprise.

defined a relation schema R by listing its attributes, and then defined a relation r on the schema using the notation $r(R)$. The notation in Figure 3.1 omits the schema name, and defines the schema of a relation by directly listing its attributes.

3.2 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 domain 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 domain values; we defer discussion of the other SQL DDL features to Chapter 4.

3.2.1 Basic Domain Types

The SQL standard supports a variety of built-in domain 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 domain 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 domain values are covered in Section 4.1.

3.2.2 Basic Schema Definition in SQL

We define an SQL relation by using the **create table** command:

create table $r(A_1D_1, A_2D_2, \ldots, A_nD_n,$
$\langle$integrity-constraint$_1\rangle$,
$\ldots,$
$\langle$integrity-constraint$_k\rangle)$

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 type of values in the domain of attribute A_i. There are a number of different allowable integrity constraints. In this section we only discuss primary key,which takes the form:

- **primary key** $(A_{j_1}, A_{j_2}, \ldots, A_{j_m})$: The **primary-key** specification says that attributes $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$ form the primary key for the relation. The primary-key attributes are required to be *non null* 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.[1] Although the primary-key specification is optional, it is generally a good idea to specify a primary key for each relation.

Other integrity constraints that the **create table** command may include are covered later, in Section 4.2.

Figure 3.2 presents a partial SQL DDL definition of our bank database. Note that, as in earlier chapters, we do not attempt to model precisely the real world in the bank database example. In the real world, multiple people may have the same name, so *customer_name* would not be a primary key *customer*; a *customer_id* would more likely be used as a primary key. We use *customer_name* as a primary key to keep our database schema simple and short.

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.

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 account A-9732 at the Perryridge branch and that it has a balance of $1200, we write

insert into *account*
values ('A-9732', 'Perryridge', 1200)

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 3.10.2.

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

delete from *account*

1. In SQL-89, primary-key attributes were not implicitly declared to be **not null**; an explicit **not null** declaration was required.

```
create table customer
    (customer_name    char(20),
     customer_street  char(30),
     customer_city    char(30),
     primary key (customer_name))

create table branch
    (branch_name      char(15),
     branch_city      char(30),
     assets           numeric(16,2),
     primary key (branch_name))

create table account
    (account_number   char(10),
     branch_name      char(15),
     balance          numeric(12,2),
     primary key (account_number))

create table depositor
    (customer_name    char(20),
     account_number   char(10),
     primary key (customer_name, account_number))
```

Figure 3.2 SQL data definition for part of the bank database.

would delete all tuples from the *account* relation. Other forms of the delete command allow specific tuples to be deleted; the delete command is covered in more detail later, in Section 3.10.1.

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 domain 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

A relational database consists of a collection of relations, each of which is assigned a unique name. Each relation has a structure similar to that presented in Chapter 2. SQL allows the use of null values to indicate that the value either is unknown or does not exist. It allows a user to specify which attributes cannot be assigned null values, as we noted in Section 3.2.

The basic structure of an SQL expression consists of three clauses: **select**, **from**, and **where**.

- The **select** clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.
- The **from** clause corresponds to the Cartesian-product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.
- The **where** clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the **from** clause.

That the term *select* has different meaning in SQL than in the relational algebra is an unfortunate historical fact. We emphasize the different interpretations here to minimize potential confusion.

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. The query is equivalent to the relational-algebra expression

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

If the **where** clause is omitted, the predicate P is **true**. However, unlike the result of a relational-algebra expression, the result of the SQL query may contain multiple copies of some tuples; we shall return to this issue in Section 3.3.8.

SQL forms the Cartesian product of the relations named in the **from** clause, performs a relational-algebra selection using the **where** clause predicate, and then

projects the result onto the attributes of the **select** clause. 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 13 and 14.

3.3.1 The select Clause

The result of an SQL query is, of course, a relation. Let us consider a simple query using our banking example, "Find the names of all branches in the *loan* relation":

select *branch_name*
from *loan*

The result is a relation consisting of a single attribute with the heading *branch_name*.

Formal query languages are based on the mathematical notion of a relation being a set. Thus, duplicate tuples never appear in relations. In practice, duplicate elimination is time-consuming. Therefore, SQL (like most other commercial query languages) allows duplicates in relations as well as in the results of SQL expressions. Thus, the preceding query will list each *branch_name* once for every tuple in which it appears in the *loan* 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 *branch_name*
from *loan*

if we want duplicates removed.

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

select all *branch_name*
from *loan*

Since duplicate retention is the default, we will not use **all** in our examples. To ensure the elimination of duplicates in the results of our example queries, we will use **distinct** whenever it is necessary. In most queries where **distinct** is not used, the exact number of duplicate copies of each tuple present in the query result is not important. However, the number is important in certain applications; we return to this issue in Section 3.3.8.

The asterisk symbol " * " can be used to denote "all attributes." Thus, the use of *loan*.* in the preceding **select** clause would indicate that all attributes of *loan* are to be selected. A select clause of the form **select** * indicates that all attributes of all relations appearing in the **from** clause are selected.

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

select *loan_number*, *branch_name*, *amount* * 100
from *loan*

will return a relation that is the same as the *loan* relation, except that the attribute *amount* is multiplied by 100.

SQL also provides special data types, such as various forms of the *date* type, and allows several arithmetic functions to operate on these types.

3.3.2 The where Clause

Let us illustrate the use of the **where** clause in SQL. Consider the query "Find all loan numbers for loans made at the Perryridge branch with loan amounts greater that $1200." This query can be written in SQL as:

select *loan_number*
from *loan*
where *branch_name* = 'Perryridge' **and** *amount* > 1200

SQL uses the logical connectives **and**, **or**, and **not**—rather than the mathematical symbols $\wedge$, $\vee$, and $\neg$ —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.

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 loan number of those loans with loan amounts between $90,000 and $100,000, we can use the **between** comparison to write

select *loan_number*
from *loan*
where *amount* **between** 90000 **and** 100000

instead of

select *loan_number*
from *loan*
where *amount* <= 100000 **and** *amount* >= 90000

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

3.3.3 The from Clause

Finally, let us discuss the use of the **from** clause. The **from** clause by itself defines a Cartesian product of the relations in the clause. Since the natural join is defined in terms of a Cartesian product, a selection, and a projection, it is a relatively simple matter to write an SQL expression for the natural join.

We write the relational-algebra expression

$$\Pi_{customer\_name,\, loan\_number,\, amount}\ (borrower \bowtie loan)$$

for the query "For all customers who have a loan from the bank, find their names, loan numbers, and loan amount." In SQL, this query can be written as

select *customer_name, borrower.loan_number, amount*
from *borrower, loan*
where *borrower.loan_number* = *loan.loan_number*

Notice that SQL uses the notation *relation-name.attribute-name*, as does the relational algebra, to avoid ambiguity in cases where an attribute appears in the schema of more than one relation. We could have written *borrower.customer_name* instead of *customer_name* in the **select** clause. However, since the attribute *customer_name* appears in only one of the relations named in the **from** clause, there is no ambiguity when we write *customer_name*.

We can extend the preceding query and consider a more complicated case in which we require also that the loan be from the Perryridge branch: "Find the customer names, loan numbers, and loan amounts for all loans at the Perryridge branch." To write this query, we need to state two constraints in the **where** clause, connected by the logical connective **and**:

select *customer_name, borrower.loan_number, amount*
from *borrower, loan*
where *borrower.loan_number* = *loan.loan_number* **and**
branch_name = 'Perryridge'

SQL includes extensions to perform natural joins and outer joins in the **from** clause. We discuss these extensions in Section 3.11.

3.3.4 The Rename Operation

SQL provides a mechanism for renaming both relations and attributes. 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.

Consider again the query that we used earlier:

select *customer_name, borrower.loan_number, amount*
from *borrower, loan*
where *borrower.loan_number* = *loan.loan_number*

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

customer_name, loan_number, amount

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.

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

select *customer_name, borrower.loan_number* **as** *loan_id, amount*
from *borrower, loan*
where *borrower.loan_number* = *loan.loan_number*

3.3.5 Tuple Variables

The **as** clause is particularly useful in defining the notion of tuple variables. A tuple variable in SQL must be associated with a particular relation. Tuple variables are defined in the **from** clause by way of the **as** clause. To illustrate, we rewrite the query "For all customers who have a loan from the bank, find their names, loan numbers, and loan amount" as

select *customer_name, T.loan_number, S.amount*
from *borrower* **as** *T*, *loan* **as** *S*
where *T.loan_number* = *S.loan_number*

Note that we define a tuple variable in the **from** clause by placing it after the name of the relation with which it is associated, with the keyword **as** in between (the keyword **as** is optional). When we write expressions of the form *relation-name.attribute-name*, the relation name is, in effect, an implicitly defined tuple variable.

Tuple variables are most useful for comparing two tuples in the same relation. Recall that, in such cases, we could use the rename operation in the relational algebra. Suppose that we want the query "Find the names of all branches that have assets greater than at least one branch located in Brooklyn." We can write the SQL expression

select distinct *T.branch_name*
from *branch* **as** *T*, *branch* **as** *S*
where *T.assets* > *S.assets* **and** *S.branch_city* = 'Brooklyn'

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

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, $(a_1, a_2) <= (b_1, b_2)$ is true if $a_1 < b_1$, or $(a_1 = b_1) \wedge (a_2 <= b_2)$; similarly, the two tuples are equal if all their attributes are equal.

3.3.6 String Operations

SQL specifies strings by enclosing them in single quotes, for example, 'Perryridge,' as we saw earlier. 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 most commonly used operation on strings is pattern matching 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:

- 'Perry%' matches any string beginning with "Perry."
- '%idge%' matches any string containing "idge" as a substring, for example, 'Perryridge', 'Rock Ridge', 'Mianus Bridge', and 'Ridgeway.'
- '___' 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 customers whose street address includes the substring 'Main'." This query can be written as

select *customer_name*
from *customer*
where *customer_street* **like** '%Main%'

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.

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 **upper**()) and lowercase (using **lower**()), and so on. 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.

There are variations on the exact set of string functions supported by different database systems. Some database systems do not distinguish uppercase from lowercase when matching strings. Thus, 'ABC' **like** 'abc' would return true, as would 'ABC' = 'abc', on such systems. Others provide extensions to specify that a string match should ignore the case. See your database system's manual for more details on exactly what string functions it supports.

3.3.7 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 customers who have a loan at the Perryridge branch, we write

select distinct *customer_name*
from *borrower, loan*
where *borrower.loan_number* = *loan.loan_number* **and**
branch_name = 'Perryridge'
order by *customer_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 *loan* relation in descending order of *amount*. If several loans have the same amount, we order them in ascending order by loan number. We express this query in SQL as follows:

select *
from *loan*
order by *amount* **desc**, *loan_number* **asc**

To fulfill an **order by** request, SQL must perform a sort. Since sorting a large number of tuples may be costly, it should be done only when necessary.

3.3.8 Duplicates

Using relations with duplicates offers advantages in several situations. Accordingly, SQL formally defines not only what tuples are in the result of a query, but also how many copies of each of those tuples appear in the result. We can define the duplicate semantics of an SQL query using *multiset* versions of the relational operators. Here, we define the multiset versions of several of the relational-algebra operators. Given multiset relations r_1 and r_2,

1. If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selection σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
2. For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$, where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.

3. If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 * c_2$ copies of the tuple $t_1.t_2$ in $r_1 \times r_2$.

For example, suppose that relations r_1 with schema (A, B) and r_2 with schema (C) are the following multisets:

$$r_1 = \{(1, a), (2, a)\} \qquad r_2 = \{(2), (3), (3)\}$$

Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$, whereas $\Pi_B(r_1) \times r_2$ would be

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

We can now define how many copies of each tuple occur in the result of an SQL query. An SQL query of the form

select $A_1,\ A_2, \ldots, A_n$
from $r_1,\ r_2, \ldots, r_m$
where P

is equivalent to the relational-algebra expression

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

using the multiset versions of the relational operators σ, Π, and $\times$.

3.4 Set Operations

The SQL operations **union**, **intersect**, and **except** operate on relations and correspond to the relational-algebra operations $\cup$, $\cap$, and $-$. Like union, intersection, and set difference in relational algebra, the relations participating in the operations must be *compatible*; that is, they must have the same set of attributes.

Let us demonstrate how several of the example queries that we considered in Chapter 2 can be written in SQL. We shall now construct queries involving the **union**, **intersect**, and **except** operations of two sets: the set of all customers who have an account at the bank, which can be derived by

select *customer_name*
from *depositor*

and the set of customers who have a loan at the bank, which can be derived by

select *customer_name*
from *borrower*

In our discussion that follows, we shall refer to the relations obtained as the result of the preceding queries as *d* and *b*, respectively.

3.4.1 The Union Operation

To find all the bank customers having a loan, an account, or both at the bank, we write

```
(select customer_name
 from depositor)
union
(select customer_name
 from borrower)
```

The **union** operation automatically eliminates duplicates, unlike the **select** clause. Thus, in the preceding query, if a customer—say, Jones—has several accounts or loans (or both) at the bank, then Jones will appear only once in the result.

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

```
(select customer_name
 from depositor)
union all
(select customer_name
 from borrower)
```

The number of duplicate tuples in the result is equal to the total number of duplicates that appear in both d and b. Thus, if Jones has three accounts and two loans at the bank, then there will be five tuples with the name Jones in the result.

3.4.2 The Intersect Operation

To find all customers who have both a loan and an account at the bank, we write

```
(select distinct customer_name
 from depositor)
intersect
(select distinct customer_name
 from borrower)
```

The **intersect** operation automatically eliminates duplicates. Thus, in the preceding query, if a customer—say, Jones—has several accounts and loans at the bank, then Jones will appear only once in the result.

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

```
(select customer_name
 from depositor)
intersect all
(select customer_name
 from borrower)
```

The number of duplicate tuples that appear in the result is equal to the minimum number of duplicates in both d and b. Thus, if Jones has three accounts and two loans at the bank, then there will be two tuples with the name Jones in the result.

3.4.3 The Except Operation

To find all customers who have an account but no loan at the bank, we write

```
(select distinct customer_name
 from depositor)
except
(select customer_name
 from borrower)
```

The **except** operation automatically eliminates duplicates. Thus, in the preceding query, a tuple with customer name Jones will appear (exactly once) in the result only if Jones has an account at the bank, but has no loan at the bank.

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

```
(select customer_name
 from depositor)
except all
(select customer_name
 from borrower)
```

The number of duplicate copies of a tuple in the result is equal to the number of duplicate copies of the tuple in *depositor* minus the number of duplicate copies of the tuple in *borrower*, provided that the difference is positive. Thus, if Jones has three accounts and one loan at the bank, then there will be two tuples with the name Jones in the result. If, instead, this customer has two accounts and three loans at the bank, there will be no tuple with the name Jones in the result.

3.5 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.

As an illustration, consider the query "Find the average account balance at the Perryridge branch." We write this query as follows:

select avg (*balance*)
from *account*
where *branch_name* = 'Perryridge'

The result of this query is a relation with a single attribute, containing a single tuple with a numerical value corresponding to the average balance at the Perryridge branch. Optionally, we can give a name to the attribute of the result relation by using the **as** clause.

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 account balance at each branch." We write this query as follows:

select *branch_name*, **avg** (*balance*)
from *account*
group by *branch_name*

Retaining duplicates is important in computing an average. Suppose that the account balances at the (small) Brighton branch are \$1000, \$3000, \$2000, and \$1000. The average balance is $\$7000/4 = \$1750.00$. If duplicates were eliminated, we would obtain the wrong answer $(\$6000/3 = \$2000)$.

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 number of depositors for each branch." In this case, a depositor counts only once, regardless of the number of accounts that depositor may have. We write this query as follows:

select *branch_name*, **count** (**distinct** *customer_name*)
from *depositor, account*
where *depositor.account_number* = *account.account_number*
group by *branch_name*

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 branches where the average account balance is more than \$1200. 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 *branch_name*, **avg** (*balance*)
from *account*
group by *branch_name*
having avg (*balance*) > 1200

At times, we wish to treat the entire relation as a single group. In such cases, we do not use a **group by** clause. Consider the query "Find the average balance for all accounts." We write this query as follows:

select avg (*balance*)
from *account*

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 *customer* relation, we write

select count (*)
from *customer*

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.

If a **where** clause and a **having** clause appear in the same query, SQL applies the predicate in the **where** clause first. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause. SQL then applies the **having** clause, if it is present, to each group; it removes the groups that do not satisfy the **having** clause predicate. The **select** clause uses the remaining groups to generate tuples of the result of the query.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query "Find the average balance for each customer who lives in Harrison and has at least three accounts."

select *depositor.customer_name*, **avg** (*balance*)
from *depositor, account, customer*
where *depositor.account_number* = *account.account_number* **and**
depositor.customer_name = *customer.customer_name* **and**
customer_city = 'Harrison'
group by *depositor.customer_name*
having count (**distinct** *depositor.account_number*) >= 3

3.6 Null Values

SQL allows the use of *null* values to indicate absence of information about the value of an attribute.

We can use the special keyword **null** in a predicate to test for a null value. Thus, to find all loan numbers that appear in the *loan* relation with null values for *amount*, we write

select *loan_number*
from *loan*
where *amount* **is null**

The predicate **is not null** tests for the absence of a null value.

The use of a *null* value in arithmetic and comparison operations causes several complications. In Section 2.5 we saw how null values are handled in the relational algebra. We now outline how SQL handles null values.

The result of an arithmetic expression (involving, for example $+$, $-$, $*$ or $/$) is null if any of the input values is null. SQL treats as **unknown** the result of any comparison involving a *null* value (other than **is null** and **is not null**).

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**, as outlined in Section 2.5.

- **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*.

SQL defines the result of an SQL statement of the form

$$\textbf{select} \ldots \textbf{from}\ R_1, \cdots, R_n\ \textbf{where}\ P$$

to contain (projections of) tuples in $R_1 \times \cdots \times R_n$ for which predicate P evaluates to **true**. If the predicate evaluates to either **false** or **unknown** for a tuple in $R_1 \times \cdots \times R_n$ (the projection of) the tuple is not added to the result.

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

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

select sum (*amount*)
from *loan*

The values to be summed in the preceding query include null values, since some tuples have a null value for *amount*. 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** type data, which 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.

3.7 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. We shall study these uses in subsequent sections.

3.7.1 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 customers who have both a loan and an account at the bank." Earlier, we wrote such a query by intersecting two sets: the set of depositors at the bank, and the set of borrowers from the bank. We can take the alternative approach of finding all account holders at the bank who are members of the set of borrowers from the bank. 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 account holders, and we write the subquery

(**select** *customer_name*
from *depositor*)

We then need to find those customers who are borrowers from the bank and who appear in the list of account holders obtained in the subquery. We do so by nesting the subquery in an outer **select**. The resulting query is

select distinct *customer_name*
from *borrower*
where *customer_name* **in** (**select** *customer_name*
from *depositor*)

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.

In the preceding example, we tested membership in a one-attribute relation. It is also possible to test for membership in an arbitrary relation in SQL. We can thus write the query "Find all customers who have both an account and a loan at the Perryridge branch" in yet another way:

```
select distinct customer_name
from borrower, loan
where borrower.loan_number = loan.loan_number and
      branch_name = 'Perryridge' and
      (branch_name, customer_name) in
          (select branch_name, customer_name
           from depositor, account
           where depositor.account_number = account.account_number)
```

We use the **not in** construct in a similar way. For example, to find all customers who do have a loan at the bank, but do not have an account at the bank, we can write

```
select distinct customer_name
from borrower
where customer_name not in (select customer_name
                            from depositor)
```

The **in** and **not in** operators can also be used on enumerated sets. The following query selects the names of customers who have a loan at the bank, and whose names are neither Smith nor Jones.

```
select distinct customer_name
from borrower
where customer_name not in ('Smith', 'Jones')
```

3.7.2 Set Comparison

As an example of the ability of a nested subquery to compare sets, consider the query "Find the names of all branches that have assets greater than those of at least one branch located in Brooklyn." In Section 3.3.5, we wrote this query as follows:

```
select distinct T.branch_name
from branch as T, branch as S
where T.assets > S.assets and S.branch_city = 'Brooklyn'
```

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 branch_name
from branch
where assets > some (select assets
                     from branch
                     where branch_city = 'Brooklyn')
```

The subquery

(**select** *assets*
from *branch*
where *branch_city* = 'Brooklyn')

generates the set of all asset values for all branches in Brooklyn. The > **some** comparison in the **where** clause of the outer **select** is true if the *assets* value of the tuple is greater than at least one member of the set of all asset values for branches in Brooklyn.

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**. 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.

Now we modify our query slightly. Let us find the names of all branches that have an asset value greater than that of each branch in Brooklyn. The construct > **all** corresponds to the phrase "greater than all." Using this construct, we write the query as follows:

select *branch_name*
from *branch*
where *assets* > **all** (**select** *assets*
from *branch*
where *branch_city* = 'Brooklyn')

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**.

As another example of set comparisons, consider the query "Find the branch that has the highest average balance." Aggregate functions cannot be composed in SQL. Thus, we cannot use **max** (**avg** (. . .)). Instead, we can follow this strategy: We begin by writing a query to find all average balances, and then nest it as a subquery of a larger query that finds those branches for which the average balance is greater than or equal to all average balances:

select *branch_name*
from *account*
group by *branch_name*
having avg (*balance*) >= **all** (**select avg** (*balance*)
from *account*
group by *branch_name*)

3.7.3 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 customers who have both an account and a loan at the bank" in still another way:

```
select customer_name
from borrower
where exists (select *
              from depositor
              where depositor.customer_name = borrower.customer_name)
```

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 SQL-92 and SQL:1999 standards, the **contains** operator was present in some early relational systems.) To illustrate the **not exists** operator, consider again the query "Find all customers who have an account at all the branches located in Brooklyn." For each customer, we need to see whether the set of all branches at which that customer has an account contains the set of all branches in Brooklyn. Using the **except** construct, we can write the query as follows:

```
select distinct S.customer_name
from depositor as S
where not exists ((select branch_name
                   from branch
                   where branch_city = 'Brooklyn')
                  except
                   (select R.branch_name
                    from depositor as T, account as R
                    where T.account_number = R.account_number and
                          S.customer_name = T.customer_name))
```

Here, the subquery

```
(select branch_name
 from branch
 where branch_city = 'Brooklyn')
```

finds all the branches in Brooklyn. The subquery

```
(select R.branch_name
 from depositor as T, account as R
 where T.account_number = R.account_number and
       S.customer_name = T.customer_name)
```

finds all the branches at which customer *S.customer_name* has an account. Thus, the outer **select** takes each customer and tests whether the set of all branches at which that customer has an account contains the set of all branches located in Brooklyn.

In queries that contain subqueries, a scoping rule applies for tuple variables. In a subquery, according to the rule, it is legal to use only tuple variables defined in the subquery itself or in any query that contains the subquery. If a tuple variable 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.

3.7.4 Test for the Absence of Duplicate Tuples

SQL includes a feature for testing whether a subquery has any duplicate tuples in its result. The **unique** construct returns the value **true** if the argument subquery contains no duplicate tuples. Using the **unique** construct, we can write the query "Find all customers who have at most one account at the Perryridge branch" as follows:

```
select T.customer_name
from depositor as T
where unique (select R.customer_name
              from account, depositor as R
              where T.customer_name = R.customer_name and
                    R.account_number = account.account_number and
                    account.branch_name = 'Perryridge')
```

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 customers who have at least two accounts at the Perryridge branch," which we write as

```
select distinct T.customer_name
from depositor T
where not unique (select R.customer_name
                  from account, depositor as R
                  where T.customer_name = R.customer_name and
                        R.account_number = account.account_number and
                        account.branch_name = 'Perryridge')
```

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 Complex Queries

Complex queries are often hard or impossible to write as a single SQL block or a union/intersection/difference of SQL blocks. (An SQL block consists of a single **select-from-where** statement, possibly with **group by** and **having** clauses.) We study here two ways of composing multiple SQL blocks to express a complex query: derived relations and the **with** clause.

3.8.1 Derived Relations

SQL allows a subquery expression to be used in the **from** clause. If we use such an expression, then we must give the result relation a name, and we can rename the

attributes. We do this renaming by using the **as** clause. For example, consider the subquery

```
(select branch_name, avg (balance)
 from account
 group by branch_name)
as branch_avg (branch_name, avg_balance)
```

This subquery generates a relation consisting of the names of all branches and their corresponding average account balances. The subquery result is named *branch_avg*, with the attributes *branch_name* and *avg_balance*.

To illustrate the use of a subquery expression in the **from** clause, consider the query "Find the average account balance of those branches where the average account balance is greater than $1200." We wrote this query in Section 3.5 by using the **having** clause. We can now rewrite this query, without using the **having** clause, as follows:

```
select  branch_name, avg_balance
from (select branch_name, avg (balance)
        from account
        group by branch_name)
       as branch_avg (branch_name, avg_balance)
where avg_balance > 1200
```

Note that we do not need to use the **having** clause, since the subquery in the **from** clause computes the average balance, and its result is named as *branch_avg*; we can use the attributes of *branch_avg* directly in the **where** clause.

As another example, suppose we wish to find the maximum across all branches of the total balance at each branch. 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_balance)
from (select branch_name, sum(balance)
      from account
      group by branch_name) as branch_total (branch_name, tot_balance)
```

3.8.2 The with Clause

Complex queries are much easier to write and to understand if we structure them by breaking them into smaller views that we then combine, just as we structure programs by breaking their task into procedures. However, unlike a procedure definition, a **create view** clause creates a view definition in the database, and the view definition stays in the database until a command **drop view** *view-name* is executed.

The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs. Consider the following query, which selects accounts with the maximum balance; if there are many accounts with the same maximum balance, all of them are selected.

```
with max_balance (value) as
    select max(balance)
    from account
select account_number
from account, max_balance
where account.balance = max_balance.value
```

The **with** clause, introduced in SQL:1999, is currently supported only by some databases.

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 branches where the total account deposit is greater than the average of the total account deposits at all branches. We can write the query using the **with** clause as follows.

```
with branch_total (branch_name, value) as
    select branch_name, sum(balance)
    from account
    group by branch_name
with branch_total_avg(value) as
    select avg(value)
    from branch_total
select branch_name
from branch_total, branch_total_avg
where branch_total.value >= branch_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.

3.9 Views

In our examples up to this point, we have operated at the logical-model level. That is, we have assumed that the relations in the collection we are given are the actual relations stored in the database.

It is not desirable for all users to see the entire logical model. Security considerations may require that certain data be hidden from users. Consider a person who needs to know a customer's loan number and branch name, but has no need to see the loan amount. This person should see a relation described (modulo renaming of attributes), in SQL, by

```
select customer_name, borrower.loan_number, branch_name
from borrower, loan
where borrower.loan_number = loan.loan_number
```

Aside from security concerns, we may wish to create a personalized collection of relations that is better matched to a certain user's intuition than is the logical model. An employee in the advertising department, for example, might like to see a relation consisting of the customers who have either an account or a loan at the bank, and the branches with which they do business. The relation that we would create for that employee is

(**select** *branch_name, customer_name*
from *depositor, account*
where *depositor.account_number* = *account.account_number*)
union
(**select** *branch_name, customer_name*
from *borrower, loan*
where *borrower.loan_number* = *loan.loan_number*)

Any relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a **view**. It is possible to support a large number of views on top of any given set of actual relations.

3.9.1 View Definition

We define a view in SQL by using the **create view** command. To define a view, we must give the view a name and must state the query that computes the view. The form of the **create view** command is

create view v **as** <query expression>

where <query expression> is any legal query expression. The view name is represented by v.

As an example, consider the view consisting of branches and their customers. Assume that we want this view to be called *all_customer*. We define this view as follows:

create view *all_customer* **as**
(**select** *branch_name, customer_name*
from *depositor, account*
where *depositor.account_number* = *account.account_number*)
union
(**select** *branch_name, customer_name*
from *borrower, loan*
where *borrower.loan_number* = *loan.loan_number*)

Once we have defined a view, we can use the view name to refer to the virtual relation that the view generates. Using the view *all_customer*, we can find all customers of the Perryridge branch by writing

select *customer_name*
from *all_customer*
where *branch_name* = 'Perryridge'

View names may appear in any place where a relation name may appear, so long as no update operations are executed on the views. We study the issue of update operations on views in Section 3.10.4.

The attribute names of a view can be specified explicitly as follows:

create view *branch_total_loan*(*branch_name*, *total_loan*) **as**
select *branch_name*, **sum**(*amount*)
from *loan*
group by *branch_name*

The preceding view gives for each branch the sum of the amounts of all the loans at the branch. Since the expression **sum**(*amount*) does not have a name, the attribute name is specified explicitly in the view definition.

Intuitively, at any given time, the set of tuples in the view relation is the result of evaluation of the query expression that defines the view at that time. Thus, if a view relation is computed and stored, it may become out of date if the relations used to define it are modified. To avoid this, views are usually implemented as follows. When we define a view, the database system stores the definition of the view itself, rather than the result of evaluation of the relational-algebra expression that defines the view. Wherever a view relation appears in a query, it is replaced by the stored query expression. Thus, whenever we evaluate the query, the view relation gets recomputed.

Certain database systems allow view relations to be stored, but they make sure that, if the actual relations used in the view definition change, the view is kept up to date. Such views are called **materialized views**. The process of keeping the view up to date is called **view maintenance**, covered in Section 14.5. Applications that use a view frequently benefit from the use of materialized views, as do applications that demand fast response to certain view-based queries. Of course, the benefits to queries from the materialization of a view must be weighed against the storage costs and the added overhead for updates.

3.9.2 Views Defined by Using Other Views

In Section 3.9.1 we mentioned that view relations may appear in any place that a relation name may appear, except for restrictions on the use of views in update expressions. Thus, one view may be used in the expression defining another view. For example, we can define the view *perryridge_customer* as follows:

create view *perryridge_customer* **as**
select *customer_name*
from *all_customer*
where *branch_name* = 'Perryridge'

where *all_customer* is itself a view relation.

View expansion is one way to define the meaning of views defined in terms of other views. The procedure assumes that view definitions are not **recursive**; that is, no view is used in its own definition, whether directly, or indirectly through other view definitions. For example, if $v1$ is used in the definition of $v2$, $v2$ is used in the definition of $v3$, and $v3$ is used in the definition of $v1$, then each of $v1$, $v2$, and $v3$ is recursive. Recursive view definitions are useful in some situations, and we revisit them in the context of the Datalog language, in Section 5.4.

Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations. A view relation stands for the expression defining the view, and therefore a view relation can be replaced by the expression that defines it. If we modify an expression by replacing a view relation by the latter's definition, the resultant expression may still contain other view relations. Hence, view expansion of an expression repeats the replacement step as follows:

repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
until no more view relations are present in e_1

As long as the view definitions are not recursive, this loop will terminate. Thus, an expression e containing view relations can be understood as the expression resulting from view expansion of e, which does not contain any view relations.

As an illustration of view expansion, consider the following expression:

```
select *
from perryridge_customer
where customer_name = 'John'
```

The view-expansion procedure initially generates

```
select *
from  (select customer_name
       from all_customer
       where branch_name = 'Perryridge')
where customer_name = 'John'
```

It then generates

```
select *
from (select customer_name
      from ((select branch_name, customer_name
             from depositor, account
             where depositor.account_number = account.account_number)
             union
             (select branch_name, customer_name
             from borrower, loan
             where borrower.loan_number = loan.loan_number))
      where branch_name = 'Perryridge')
where customer_name = 'John'
```

At this time, there are no more uses of view relations, and view expansion terminates.

3.10 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.

3.10.1 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 *loan*

deletes all tuples from the *loan* relation. (Well-designed systems will seek confirmation from the user before executing such a devastating request.)

Here are examples of SQL delete requests:

- Delete all account tuples in the Perryridge branch.

  delete from *account*
  where *branch_name* = 'Perryridge'

- Delete all loans with loan amounts between $1300 and $1500.

  delete from *loan*
  where *amount* **between** 1300 **and** 1500

- Delete all account tuples at every branch located in Brooklyn.

  delete from *account*
  where *branch_name* **in** (**select** *branch_name*
  from *branch*
  where *branch_city* = 'Brooklyn')

This **delete** request first finds all branches in Brooklyn, and then deletes all *account* tuples pertaining to those branches.

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 accounts with balances below the average at the bank. We could write

delete from *account*
where *balance* < (**select avg** (*balance*)
from *account*)

The **delete** statement first tests each tuple in the relation *account* to check whether the account has a balance less than the average at the bank. Then, all tuples that fail the test—that is, represent an account with a lower-than-average balance—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 balance may change, and the final result of the **delete** would depend on the order in which the tuples were processed!

3.10.2 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 attribute's domain. Similarly, tuples inserted must be of the correct arity.

The simplest **insert** statement is a request to insert one tuple. Suppose that we wish to insert the fact that there is an account A-9732 at the Perryridge branch and that it has a balance of $1200. We write

insert into *account*
values ('A-9732', 'Perryridge', 1200)

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 *account* (*account_number, branch_name, balance*)
values ('A-9732', 'Perryridge', 1200)

insert into *account* (*branch_name, account_number, balance*)
values ('Perryridge', 'A-9732', 1200)

More generally, we might want to insert tuples on the basis of the result of a query. Suppose that we want to present a new $200 savings account as a gift to all loan

customers of the Perryridge branch, for each loan they have. Let the loan number serve as the account number for the savings account. We write

```
insert into account
    select loan_number, branch_name, 200
    from loan
    where branch_name = 'Perryridge'
```

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 *account* relation. Each tuple has a *loan_number* (which serves as the account number for the new account), a *branch_name* (Perryridge), and an initial balance of the new account ($200).

We also need to add tuples to the *depositor* relation; we do so by writing

```
insert into depositor
    select customer_name, loan_number
    from borrower, loan
    where borrower.loan_number = loan.loan_number and
        branch_name = 'Perryridge'
```

This query inserts a tuple (*customer_name, loan_number*) into the *depositor* relation for each *customer_name* who has a loan in the Perryridge branch with loan number *loan_number*.

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 account
    select *
    from account
```

might insert an infinite number of tuples! The request would insert the first tuple in *account* again, creating a second copy of the tuple. Since this second copy is part of *account* now, the **select** statement may find it, and a third copy would be inserted into *account*. 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.

Our discussion of the **insert** statement considered only examples in which a value is given for every attribute in inserted tuples. It is possible, as we saw in Chapter 2, 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 account
    values ('A-401', null, 1200)
```

We know that account A-401 has $1200, but the branch name is not known. Consider the query

select *account_number*
from *account*
where *branch_name* = 'Perryridge'

Since the branch at which account A-401 is maintained is not known, we cannot determine whether it is equal to "Perryridge."

We can prohibit the insertion of null values on specified attributes by using the SQL DDL, which we discuss in Section 3.2.

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.

3.10.3 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 interest payments are being made, and all balances are to be increased by 5 percent. We write

update *account*
set *balance* = *balance* * 1.05

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

If interest is to be paid only to accounts with a balance of $1000 or more, we can write

update *account*
set *balance* = *balance* * 1.05
where *balance* >= 1000

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 "Pay 5 percent interest on accounts whose balance is greater than average" as follows:

update *account*
set *balance* = *balance* * 1.05
where *balance* > (**select avg** (*balance*)
from *account*)

Let us now suppose that all accounts with balances over $10,000 receive 6 percent interest, whereas all others receive 5 percent. We could write two **update** statements:

```
update account
set balance = balance * 1.06
where balance > 10000

update account
set balance = balance * 1.05
where balance <= 10000
```

Note that, as we saw in Chapter 2, the order of the two **update** statements is important. If we changed the order of the two statements, an account with a balance just under $10,000 would receive 11.3 percent interest.

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

```
update account
set balance = case
                  when balance <= 10000 then balance * 1.05
                  else balance * 1.06
              end
```

The general form of the case statement is as follows.

```
case
    when pred_1 then result_1
    when pred_2 then result_2
    ...
    when pred_n then result_n
    else result_0
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.

3.10.4 Update of a View

Although views are a useful tool for queries, they present serious problems if we express updates, insertions, or deletions with them. The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.

To illustrate the problem, consider a clerk who needs to see all loan data in the *loan* relation, except *loan_amount*. Let *loan_branch* be the view given to the clerk. We define this view as

create view *loan_branch* **as**
select *loan_number, branch_name*
from *loan*

Since we allow a view name to appear wherever a relation name is allowed, the clerk can write:

insert into *loan_branch*
values ('L-37', 'Perryridge')

This insertion must be represented by an insertion into the relation *loan*, since *loan* is the actual relation from which the database system constructs the view *loan_branch*. However, to insert a tuple into *loan*, we must have some value for *amount*. There are two reasonable approaches to dealing with this insertion:

- Reject the insertion, and return an error message to the user.
- Insert a tuple (L-37, "Perryridge", *null*) into the *loan* relation.

Another problem with modification of the database through views occurs with a view such as

create view *loan_info* **as**
select *customer_name, amount*
from *borrower, loan*
where *borrower.loan_number* = *loan.loan_number*

This view lists the loan amount for each loan that any customer of the bank has. Consider the following insertion through this view:

insert into *loan_info*
values ('Johnson', 1900)

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500
null	*null*	1900

loan

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17
Johnson	*null*

borrower

Figure 3.3 Tuples inserted into *loan* and *borrower*.

The only possible method of inserting tuples into the *borrower* and *loan* relations is to insert ("Johnson", *null*) into *borrower* and (*null*, *null*, 1900) into *loan*. Then, we obtain the relations shown in Figure 3.3. However, this update does not have the desired effect, since the view relation *loan_info* still does *not* include the tuple ("Johnson", 1900). Thus, there is no way to update the relations *borrower* and *loan* by using nulls to get the desired update on *loan_info*.

Because of problems such as these, modifications are generally not permitted on view relations, except in limited cases. Different database systems specify different conditions under which they permit updates on view relations; see the database system manuals for details. The general problem of database modification through views has been the subject of substantial research, and the bibliographic notes provide pointers to some of this research.

In general, an SQL view is said to be **updatable** (that is, inserts, updates or deletes can be applied on the view) if the following conditions are all satisfied:

- The **from** clause has only one database relation.
- The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
- Any attribute not listed in the **select** clause can be set to null.
- The query does not have a **group** by or **having** clause.

Under these constraints, the **update**, **insert**, and **delete** operations would be forbidden on the example view *all_customer* that we defined previously.

Suppose a view *downtown_account* is defined as follows:

create view *downtown_account* **as**
select *account_number*, *branch_name*, *balance*
from *account*
where *branch_name* = 'Downtown'

The above view is updatable, since it satisfies the conditions listed earlier.

Even with the conditions on updatability, the following problem still remains. Suppose that a user tries to insert the tuple ('A-999', 'Perryridge', 1000) into the *downtown_account* view. This tuple can be inserted into the *account* relation, but it would not appear in the *downtown_account* view since it does not satisfy the selection imposed by the view.

By default, SQL would allow the above update to proceed. However, views can be defined with a **with check option** clause at the end of the view definition; then, if a tuple inserted into the view does not satisfy the view's **where** clause condition, the insertion is rejected by the database system. Updates are similarly rejected if the new value does not satisfy the **where** clause conditions.

SQL:1999 has a more complex set of rules about when inserts, updates, and deletes can be executed on a view, that allows updates through a larger class of views; however, the rules are too complex to be discussed here.

3.10.5 Transactions

A **transaction** consists of a sequence of query and/or update statements. The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed. One of the following SQL statements must end the transaction:

- **Commit work** commits the current transaction; that is, it makes the updates performed by the transaction become permanent in the database. After the transaction is committed, a new transaction is automatically started.
- **Rollback work** causes the current transaction to be rolled back; that is, it undoes all the updates performed by the SQL statements in the transaction. Thus, the database state is restored to what it was before the first statement of the transaction was executed.

The keyword **work** is optional in both the statements.

Transaction rollback is useful if some error condition is detected during execution of a transaction. Commit is similar, in a sense, to saving changes to a document that is being edited, while rollback is similar to quitting the edit session without saving changes. Once a transaction has executed **commit work**, its effects can no longer be undone by **rollback work**. The database system guarantees that in the event of some failure, such as an error in one of the SQL statements, a power outage, or a system crash, a transaction's effects will be rolled back if it has not yet executed **commit work**. In the case of power outage or other system crash, the rollback occurs when the system restarts.

For instance, to transfer money from one account to another we need to update two account balances. The two update statements would form a transaction. An error while a transaction executes one of its statements would result in undoing of the effects of the earlier statements of the transaction, so that the database is not left in a partially updated state. We study further properties of transactions in Chapter 15.

If a program terminates without executing either of these commands, the updates are either committed or rolled back. The standard does not specify which of the two happens, and the choice is implementation dependent. In many SQL implementations, by default each SQL statement is taken to be a transaction on its own, and gets committed as soon as it is executed. Automatic commit of individual SQL statements must be turned off if a transaction consisting of multiple SQL statements needs to be executed. How to turn off automatic commit depends on the specific SQL implementation.

A better alternative, which is part of the SQL:1999 standard (but supported by only some SQL implementations currently), is to allow multiple SQL statements to be enclosed between the keywords **begin atomic** ... **end**. All the statements between the keywords then form a single transaction.

3.11 Joined Relations**

SQL provides not only the basic Cartesian-product mechanism for joining tuples of relations, but also provides (in SQL-92 and later SQL versions) various other mecha-

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

Figure 3.4 The *loan* and *borrower* relations.

nisms for joining relations, including condition joins and natural joins, as well as various forms of outer joins. These additional operations are typically used as subquery expressions in the **from** clause.

3.11.1 Examples

We illustrate the various join operations by using the relations *loan* and *borrower* in Figure 3.4. We start with a simple example of inner joins. Figure 3.5 shows the result of the expression

$$loan \textbf{ inner join } borrower \textbf{ on } loan.loan\_number = borrower.loan\_number$$

The expression computes the theta join of the *loan* and the *borrower* relations, with the join condition being *loan.loan_number* = *borrower.loan_number*. The attributes of the result consist of the attributes of the left-hand-side relation followed by the attributes of the right-hand-side relation.

Note that the attribute *loan_number* appears twice in the figure—the first occurrence is from *loan*, and the second is from *borrower*. The SQL standard does not require attribute names in such results to be unique. An **as** clause should be used to assign unique names to attributes in query and subquery results.

We rename the result relation of a join and the attributes of the result relation by using an **as** clause, as illustrated here:

loan **inner join** *borrower* **on** *loan.loan_number* = *borrower.loan_number*
as *lb(loan_number, branch, amount, cust, cust_loan_num)*

We rename the second occurrence of *loan_number* to *cust_loan_num*. The ordering of the attributes in the result of the join is important for the renaming.

Next, we consider an example of the **left outer-join** operation:

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

Figure 3.5 The result of *loan* **inner join** *borrower* **on** $loan.loan\_number = borrower.loan\_number$.

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Figure 3.6 The result of *loan* **left outer join** *borrower* **on** $loan.loan\_number = borrower.loan\_number$.

loan **left outer join** *borrower* **on** *loan.loan_number* = *borrower.loan_number*

We can compute the left outer-join operation logically as follows. First, compute the result of the inner join as before. Then, for every tuple t in the left-hand-side relation *loan* that does not match any tuple in the right-hand-side relation *borrower* in the inner join, add a tuple r to the result of the join: The attributes of tuple r that are derived from the left-hand-side relation are filled in with the values from tuple t, and the remaining attributes of r are filled with null values. Figure 3.6 shows the resultant relation. The tuples (L-170, Downtown, 3000) and (L-230, Redwood, 4000) join with tuples from *borrower* and appear in the result of the inner join, and hence in the result of the left outer join. On the other hand, the tuple (L-260, Perryridge, 1700) did not match any tuple from *borrower* in the inner join, and hence a tuple (L-260, Perryridge, 1700, null, null) is present in the result of the left outer join.

Finally, we consider an example of the **natural-join** operation:

loan **natural inner join** *borrower*

This expression computes the natural join of the two relations. The only attribute name common to *loan* and *borrower* is *loan_number*. Figure 3.7 shows the result of the expression. The result is similar to the result of the inner join with the **on** condition in Figure 3.5, since they have, in effect, the same join condition. However, the attribute *loan_number* appears only once in the result of the natural join, whereas it appears twice in the result of the join with the **on** condition.

3.11.2 Join Types and Conditions

In Section 3.11.1, we saw examples of the join operations permitted in SQL. Join operations take two relations and return another relation as the result. Although outer-join expressions are typically used in the **from** clause, they can be used anywhere that a relation can be used.

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Figure 3.7 The result of *loan* **natural inner join** *borrower*.

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

Join conditions
natural
on < predicate>
using (A_1, A_1, …, A_n)

Figure 3.8 Join types and join conditions.

Each of the variants of the join operations in SQL consists of a *join type* and a *join condition*. The join condition defines which tuples in the two relations match and what attributes are present in the result of the join. The join type defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated. Figure 3.8 shows some of the allowed join types and join conditions. The first join type is the inner join, and the other three are the outer joins. Of the three join conditions, we have seen the **natural** join and the **on** condition before, and we shall discuss the **using** condition, later in this section.

The use of a join condition is mandatory for outer joins, but is optional for inner joins (if it is omitted, a Cartesian product results). Syntactically, the keyword **natural** appears before the join type, as illustrated earlier, whereas the **on** and **using** conditions appear at the end of the join expression. The keywords **inner** and **outer** are optional, since the rest of the join type enables us to deduce whether the join is an inner join or an outer join.

The meaning of the join condition **natural**, in terms of which tuples from the two relations match, is straightforward. The ordering of the attributes in the result of a natural join is as follows. The join attributes (that is, the attributes common to both relations) appear first, in the order in which they appear in the left-hand-side relation. Next come all nonjoin attributes of the left-hand-side relation, and finally all nonjoin attributes of the right-hand-side relation.

The **right outer join** is symmetric to the **left outer join**. Tuples from the right-hand-side relation that do not match any tuple in the left-hand-side relation are padded with nulls and are added to the result of the right outer join.

Here is an example of combining the natural-join condition with the right outer join type:

loan **natural right outer join** *borrower*

Figure 3.9 shows the result of this expression. The attributes of the result are defined by the join type, which is a natural join; hence, *loan_number* appears only once. The

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Figure 3.9 The result of *loan* **natural right outer join** *borrower*.

first two tuples in the result are from the inner natural join of *loan* and *borrower*. The tuple (Hayes, L-155) from the right-hand-side relation does not match any tuple from the left-hand-side relation *loan* in the natural inner join. Hence, the tuple (L-155, null, null, Hayes) appears in the join result.

The join condition **using**$(A_1, A_2, \ldots, A_n)$ is similar to the natural-join condition, except that the join attributes are the attributes $A_1, A_2, \ldots, A_n$, rather than all attributes that are common to both relations. The attributes $A_1, A_2, \ldots, A_n$ must consist of only attributes that are common to both relations, and they appear only once in the result of the join.

The **full outer join** is a combination of the left and right outer-join types. After the operation computes the result of the inner join, it extends with nulls tuples from the left-hand-side relation that did not match with any from the right-hand-side, and adds them to the result. Similarly, it extends with nulls tuples from the right-hand-side relation that did not match with any tuples from the left-hand-side relation and adds them to the result.

For example, Figure 3.10 shows the result of the expression

loan **full outer join** *borrower* **using** (loan_number)

As another example of the use of the outer-join operation, we can write the query "Find all customers who have an account but no loan at the bank" as

select *d_CN*
from (*depositor* **left outer join** *borrower*
  on *depositor.customer_name* = *borrower.customer_name*)
  as *db*1 (*d_CN*, *account_number*, *b_CN*, *loan_number*)
where *b_CN* **is** *null*

Similarly, we can write the query "Find all customers who have either an account or a loan (but not both) at the bank," with natural full outer joins as:

select *customer_name*
from (*depositor* **natural full outer join** *borrower*)
where *account_number* **is** *null* **or** *loan_number* **is** *null*

SQL-92 also provides two other join types, called **cross join** and **union join**. The first is equivalent to an inner join without a join condition; the second is equivalent to a full outer join on the "false" condition—that is, where the inner join is empty.

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Figure 3.10 The result of *loan* **full outer join** *borrower* **using** (*loan_number*).

3.12 Summary

- Commercial database systems do not use the terse, formal relational algebra covered in Chapter 2. The widely used SQL language, which we studied in this chapter, is based on the relational algebra, but includes much "syntactic sugar."
- The SQL data-definition language is used to create relations with specified schemas. The SQL DDL supports a number of types including **date** and **time** types. Further details on the SQL DDL, in particular its support for integrity constraints, appear in Section 3.2.
- SQL includes a variety of language constructs for queries on the database. All the relational-algebra operations, including the extended relational-algebra operations, can be expressed by SQL. SQL also allows ordering of query results by sorting on specified attributes.
- SQL handles queries on relations containing null values by adding the truth value "unknown" to the usual truth values of true and false.
- SQL allows nested subqueries in the where clause. The outer query can perform a variety of operations on the subquery result such as checking for emptiness or containment of a value in the subquery result. Subqueries in the from clause are called derived relations.
- View relations can be defined as relations containing the result of queries. Views are useful for hiding unneeded information, and for collecting together information from more than one relation into a single view.
- Temporary views defined by using the **with** clause are also useful for breaking up complex queries into smaller and easier-to-understand parts.
- SQL provides constructs for updating, inserting, and deleting information. Updates through views are allowed only when some fairly restrictive conditions are satisfied.
- Transactions are a sequence of queries and updates that together carry out a task. Transactions can be committed, or rolled back; when a transaction is rolled back, the effects of all updates performed by the transaction are undone.
- SQL supports several types of outer join with several types of join conditions.

Review Terms

- DDL: data-definition language
- DML: data-manipulation language
- **select** clause
- **from** clause
- **where** clause
- **as** clause
- Tuple variable
- **order** by clause
- Duplicates

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

Figure 3.11 Insurance database.

- Set operations
  - ☐ **union, intersect, except**
- Aggregate functions
  - ☐ **avg, min, max, sum, count**
  - ☐ **group by**
- Null values
  - ☐ Truth value "unknown"
- Nested subqueries
- Set operations
  - ☐ $\{<, <=, >, >=\}$ { **some, all** }
  - ☐ **exists**
  - ☐ **unique**
- Derived relations (in **from** clause)
- **with** clause
- Views
  - ☐ View definition
  - ☐ View expansion
- Database modification
  - ☐ **delete, insert, update**
  - ☐ View update
- Transaction
  - ☐ commit
  - ☐ rollback
- Join types
  - ☐ Inner and outer join
  - ☐ left, right and full outer join
  - ☐ natural, using, and on

Practice Exercises

3.1 Consider the insurance database of Figure 3.11, 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 1989.

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

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

3.2 Consider the employee database of Figure 3.12, where the primary keys are underlined. Give an expression in SQL for each of the following queries.

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

Figure 3.12 Employee database.

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

3.3 Consider the relational database of Figure 3.12. 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.

3.4 SQL-92 provides an n-ary operation called **coalesce**, which is defined as follows: **coalesce**$(A_1, A_2, \ldots, A_n)$ returns the first nonnull A_i in the list $A_1, A_2, \ldots, A_n$, and returns null if all of $A_1, A_2, \ldots, A_n$ are null.

Let a and b be relations with the schemas A(*name*, *address*, *title*) and B(*name*, *address*, *salary*), respectively. Show how to express a **natural full outer join** b using the **full outer-join** operation with an **on** condition and the **coalesce** operation. Make sure that the result relation does not contain two copies of the attributes *name* and *address*, and that the solution is correct even if some tuples in a and b have null values for attributes *name* or *address*.

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

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

3.6 Consider the SQL query

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

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

3.7 Certain systems allow *marked* nulls. A marked null $\perp_i$ is equal to itself, but if $i \neq j$, then $\perp_i \neq \perp_j$. One application of marked nulls is to allow certain updates through views. Consider the view *loan_info* (Section 3.9). Show how you can use marked nulls to allow the insertion of the tuple ("Johnson", 1900) through *loan_info*.

Exercises

3.8 Consider the insurance database of Figure 3.11, 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 license number "AABB2000" in the accident with report number "AR2197" to $3000.

3.9 Consider the employee database of Figure 3.12, 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.

3.10 Consider the relational database of Figure 3.12. 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.

3.11 Let the following relation schemas be given:

$$R = (A, B, C)$$
$$S = (D, E, F)$$

Let relations $r(R)$ and $s(S)$ be given. Give an expression in SQL that is equivalent to each of the following queries.

- **a.** $\Pi_A(r)$
- **b.** $\sigma_{B=17}(r)$
- **c.** $r \times s$
- **d.** $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

3.12 Let $R = (A, B, C)$, and let r_1 and r_2 both be relations on schema R. Give an expression in SQL that is equivalent to each of the following queries.

a. $r_1 \cup r_2$
b. $r_1 \cap r_2$
c. $r_1 - r_2$
d. $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$

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

3.14 Consider the relational database of Figure 3.12. Using SQL, define a view consisting of *manager_name* and the average salary of all employees who work for that manager. Explain why the database system should not allow updates to be expressed in terms of this view.

3.15 Write an SQL query, without using a **with** clause, to find all branches where the total account deposit is less than the average total account deposit at all branches,

a. Using a nested query in the **from** clause.
b. Using a nested query in a **having** clause.

3.16 List two reasons why null values might be introduced into the database.

3.17 Show how to express the **coalesce** operation from Exercise 3.4 using the **case** operation.

3.18 Give an SQL schema definition for the employee database of Figure 3.12. Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema.

3.19 Using the relations of our sample bank database, write SQL expressions to define the following views:

a. A view containing the account numbers and customer names (but not the balances) for all accounts at the Deer Park branch.
b. A view containing the names and addresses of all customers who have an account with the bank, but do not have a loan.
c. A view containing the name and average account balance of every customer of the Rock Ridge branch.

3.20 For each of the views that you defined in Exercise 3.19, explain how updates would be performed (if they should be allowed at all).

3.21 Consider the following relational schema

employee(*empno*, *name*, *office*, *age*)
books(*isbn*, *title*, *authors*, *publisher*)
loan(*empno*, *isbn*, *date*)

Write the following queries in SQL.

a. Print the names of employees who have borrowed any book published by McGraw-Hill.
b. Print the names of employees who have borrowed all books published by McGraw-Hill.

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

3.22 Consider the relational schema

student(*student_id*, *student_name*)
registered(*student_id*, *course_id*)

Write an SQL query to list the student-id and name of each student along with the total number of courses that the student is registered for. Students who are not registered for any course must also be listed, with the number of registered courses shown as 0.

3.23 Suppose that we have a relation *marks*(*student_id*, *score*). Write an SQL query to find the *dense rank* of each student. That is, all students with the top mark get a rank of 1, those with the next highest mark get a rank of 2, and so on. Hint: Split the task into parts, using the **with** clause.

Bibliographical Notes

The original version of SQL, called Sequel 2, is described by Chamberlin et al. [1976]. Sequel 2 was derived from the languages 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.

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 and SQL:2003 standards are published as a collection of ISO/IEC standards documents, which are described in more detail in Section 23.3. The standard documents are densely packed with information and hard to read, and of use primarily for database system implementers. The standards documents are available for purchase electronically from the Web site http://webstore.ansi.org.

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 13 and 14. Bibliographic references on these matters appear in those chapters.

The rules used by SQL to determine the updatability of a view, and how updates are reflected on the underlying database relations, are defined by the SQL:1999 standard, and are summarized in Melton and Simon [2001].