P A R T 2

Relational Databases

A relational database is a shared repository of data. To make data from a relational database available to users, we have to address several issues. One is how users specify requests for data: Which of the various query languages do they use? Chapter 4 covers the SQL language, which is the most widely used query language today. Chapter 5 covers two other query languages, QBE and Datalog, which offer alternative approaches to querying relational data.

Another issue is data integrity and security; databases need to protect data from damage by user actions, whether unintentional or intentional. The integrity maintenance component of a database ensures that updates do not violate integrity constraints that have been specified on the data. The security component of a database includes authentication of users, and access control, to restrict the permissible actions for each user. Chapter 6 covers integrity and security issues. Security and integrity issues are present regardless of the data model, but for concreteness we study them in the context of the relational model. Integrity constraints form the basis of relational database design, which we study in Chapter 7.

Relational database design—the design of the relational schema—is the first step in building a database application. Schema design was covered informally in earlier chapters. There are, however, principles that can be used to distinguish good database designs from bad ones. These are formalized by means of several "normal forms," which offer different tradeoffs between the possibility of inconsistencies and the efficiency of certain queries. Chapter 7 describes the formal design of relational schemas.

C H A P T E R 4

SQL

The formal languages described in Chapter 3 provide a concise notation for representing queries. However, commercial database systems require a query language that is more user friendly. In this chapter, we study SQL, the most influential commercially marketed query language, SQL. SQL uses a combination of relational-algebra and relational-calculus 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.

4.1 Background

IBM developed the original version of SQL at its San Jose Research Laboratory (now the Almaden Research Center). IBM implemented the language, 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. IBM published its own corporate SQL standard, the Systems Application Architecture Database Interface (SAA-SQL) in 1987. ANSI published an extended standard for SQL, SQL-89, in 1989. The next version of the standard was SQL-92 standard, and the most recent version is SQL:1999. The bibliographic notes provide references to these standards.

In this chapter, we present a survey of SQL, based mainly on the widely implemented SQL-92 standard. The SQL:1999 standard is a superset of the SQL-92 standard; we cover some features of SQL:1999 in this chapter, and provide more detailed coverage in Chapter 9. Many database systems support some of the new constructs in SQL:1999, although currently no database system supports all the new constructs. You should also be aware that some database systems do not even support all the features of SQL-92, and that many databases provide nonstandard features that we do not cover here.

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 and the tuple relational calculus. It includes also commands to insert tuples into, delete tuples from, and modify tuples in the database.
- **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.
- **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.
- **Authorization**. The SQL DDL includes commands for specifying access rights to relations and views.

In this chapter, we cover the DML and the basic DDL features of SQL. 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. SQL features supporting integrity and authorization are described in Chapter 6, while Chapter 9 outlines object-oriented extensions to SQL.

The enterprise that we use in the examples in this chapter, and later chapters, is a banking enterprise with the following relation schemas:

Branch-schema = (*branch-name*, *branch-city*, *assets*)
Customer-schema = (*customer-name*, *customer-street*, *customer-city*)
Loan-schema = (*loan-number*, *branch-name*, *amount*)
Borrower-schema = (*customer-name*, *loan-number*)
Account-schema = (*account-number*, *branch-name*, *balance*)
Depositor-schema = (*customer-name*, *account-number*)

Note that in this chapter, as elsewhere in the text, we use hyphenated names for schema, relations, and attributes for ease of reading. In actual SQL systems, however, hyphens are not valid parts of a name (they are treated as the minus operator). A simple way of translating the names we use to valid SQL names is to replace all hyphens by the underscore symbol ("_"). For example, we use *branch_name* in place of *branch-name*.

4.2 Basic Structure

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 3. 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 shall discuss in Section 4.11.

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 4.2.8.

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.

4.2.1 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 4.2.8.

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.

4.2.2 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.

4.2.3 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

$$\Pi_{customer\text{-}name,\, loan\text{-}number,\, amount}\ (borrower \bowtie loan)$$

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 4.10.

4.2.4 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*

4.2.5 Tuple Variables

The **as** clause is particularly useful in defining the notion of tuple variables, as is done in the tuple relational calculus. 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.

4.2.6 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 between uppercase and lowercase, 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.

4.2.7 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.

4.2.8 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 $\times$.

4.3 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 3 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*

We shall refer to the relations obtained as the result of the preceding queries as *d* and *b*, respectively.

4.3.1 The Union Operation

To find all 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.

4.3.2 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.

4.3.3 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 *d* minus the number of duplicate copies of the tuple in *b*, 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.

4.4 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
```

4.5 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 3.3.4 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 3.3.4.

- **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

select . . . **from** $R_1, \cdots, R_n$ **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.

4.6 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.

4.6.1 Set Membership

SQL draws on the relational calculus for operations that allow 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 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')
```

4.6.2 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 4.2.5, 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)
```

4.6.3 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.

4.6.4 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.

4.7 Views

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. Observe that the notation that we used for view definition in the relational algebra (see Chapter 3) is based on that of SQL.

As an example, consider the view consisting of branch names and the names of customers who have either an account or a loan at that branch. 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)
```

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
groupby 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.

View names may appear in any place that a relation name may appear. 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'
```

4.8 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 **groupby** and **having** clauses.) We study here two ways of composing multiple SQL blocks to express a complex query: derived relations and the **with** clause.

4.8.1 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 result (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 *result*, 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 4.4 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)
```

4.8.2 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 less 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.

4.9 Modification of the Database

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

4.9.1 Deletion

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

```
delete from r
where P
```

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

Note that a **delete** command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation.

The predicate in the **where** clause may be as complex as a **select** command's **where** clause. At the other extreme, the **where** clause may be empty. The request

delete from *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 Needham.

  ```
  delete from account
  where branch-name in (select branch-name
                        from branch
                        where branch-city = 'Needham')
  ```

  This **delete** request first finds all branches in Needham, 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!

4.9.2 Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the 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 is 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 acocunt 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 3, 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 4.11.

4.9.3 Updates

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

Suppose that annual 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 3, 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.

4.9.4 Update of a View

The view-update anomaly that we discussed in Chapter 3 exists also in SQL. As an illustration, consider the following view definition:

```
create view loan-branch as
    select branch-name, loan-number
    from loan
```

Since SQL allows a view name to appear wherever a relation name is allowed, we can write

```
insert into loan-branch
    values ('Perryridge', 'L-307')
```

SQL represents this insertion by an insertion into the relation *loan*, since *loan* is the actual relation from which the view *loan-branch* is constructed. We must, therefore, have some value for *amount*. This value is a null value. Thus, the preceding **insert** results in the insertion of the tuple

('L-307', 'Perryridge', *null*)

into the *loan* relation.

As we saw in Chapter 3, the view-update anomaly becomes more difficult to handle when a view is defined in terms of several relations. As a result, many SQL-based database systems impose the following constraint on modifications allowed through views:

- A modification is permitted through a view only if the view in question is defined in terms of one relation of the actual relational database—that is, of the logical-level database.

Under this constraint, the **update**, **insert**, and **delete** operations would be forbidden on the example view *all-customer* that we defined previously.

4.9.5 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.

4.10 Joined Relations∗∗

SQL provides not only the basic Cartesian-product mechanism for joining tuples of relations found in its earlier versions, but, SQL also provides 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 4.1 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.

4.10.1 Examples

We illustrate the various join operations by using the relations *loan* and *borrower* in Figure 4.1. We start with a simple example of inner joins. Figure 4.2 shows the result of the expression

loan **inner join** *borrower* **on** $loan.loan\text{-}number = borrower.loan\text{-}number$

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 **left outer 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

Figure 4.2 The result of *loan* **inner join** *borrower* **on** $loan.loan\text{-}number = borrower.loan\text{-}number$.

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 4.3 The result of *loan* **left outer join** *borrower* **on** $loan.loan\text{-}number = borrower.loan\text{-}number$.

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 4.3 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 4.4 shows the result of the expression. The result is similar to the result of the inner join with the **on** condition in Figure 4.2, 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.

4.10.2 Join Types and Conditions

In Section 4.10.1, 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.

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

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Figure 4.4 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 $(A_1, A_1, \ldots, A_n)$

Figure 4.5 Join types and join conditions.

relation that do not match any tuple in the other relation (based on the join condition) are treated. Figure 4.5 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 4.6 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 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

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Figure 4.6 The result of *loan* **natural right outer join** *borrower*.

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 4.7 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 4.7 The result of *loan* **full outer join** *borrower* **using**(*loan-number*).

4.11 Data-Definition Language

In most of our discussions of SQL and relational databases, we have accepted a set of relations as given. Of course, 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 schema definition and domain values; we defer discussion of the other SQL DDL features to Chapter 6.

4.11.1 Domain Types in SQL

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.
- **date**: A calendar date containing a (four-digit) year, month, and day of the month.

- **time**: The time of day, in hours, minutes, and seconds. A variant, **time**(p), can be used to specify the number of fractional digits for seconds (the default being 0). It is also possible to store time zone information along with the time.
- **timestamp**: A combination of **date** and **time**. A variant, **timestamp**(p), can be used to specify the number of fractional digits for seconds (the default here being 6).

Date and time values can be specified like this:

date '2001-04-25'
time '09:30:00'
timestamp '2001-04-25 10:29:01.45'

Dates must be specified in the format year followed by month followed by day, as shown. The seconds field of **time** or **timestamp** can have a fractional part, as in the timestamp above. We can use an expression of the form **cast** *e* **as** *t* to convert a character string (or string valued expression) *e* to the type *t*, where *t* is one of **date, time**, or **timestamp**. The string must be in the appropriate format as illustrated at the beginning of this paragraph.

To extract individual fields of a **date** or **time** value d, we can use **extract** (*field* **from** *d*), where *field* can be one of **year, month, day, hour, minute**, or **second**.

SQL allows comparison operations on all the domains listed here, and it allows both arithmetic and comparison operations on the various numeric domains. SQL also provides a data type called **interval**, and it allows computations based on dates and times and on intervals. For example, if *x* and *y* are of type **date**, then $x - y$ is an interval whose value is the number of days from date *x* to date *y*. Similarly, adding or subtracting an interval to a date or time gives back a date or time, respectively.

It is often useful to compare values from **compatible** domains. For example, since every small integer is an integer, a comparison $x < y$, where *x* is a small integer and *y* is an integer (or vice versa), makes sense. We make such a comparison by casting small integer *x* as an integer. A transformation of this sort is called a **type coercion**. Type coercion is used routinely in common programming languages, as well as in database systems.

As an illustration, suppose that the domain of *customer-name* is a character string of length 20, and the domain of *branch-name* is a character string of length 15. Although the string lengths might differ, standard SQL will consider the two domains compatible.

As we discussed in Chapter 3, the *null* value is a member of all domains. For certain attributes, however, null values may be inappropriate. Consider a tuple in the *customer* relation where *customer-name* is null. Such a tuple gives a street and city for an anonymous customer; thus, it does not contain useful information. In cases such as this, we wish to forbid null values, and we do so by restricting the domain of *customer-name* to exclude null values.

SQL allows the domain declaration of an attribute to include the specification **not null** and thus prohibits the insertion of a null value for this attribute. Any database modification that would cause a null to be inserted in a **not null** domain generates an error diagnostic. There are many situations where we want to avoid null values. In particular, it is essential to prohibit null values in the primary key of a relation schema. Thus, in our bank example, in the *customer* relation, we must prohibit a null value for the attribute *customer-name*, which is the primary key for *customer*.

4.11.2 Schema Definition in SQL

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

$$\begin{aligned}\textbf{create table } & r(A_1 D_1, A_2 D_2, \ldots, A_n D_n,\\ & \langle\text{integrity-constraint}_1\rangle,\\ & \ldots,\\ & \langle\text{integrity-constraint}_k\rangle)\end{aligned}$$

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. The allowed integrity constraints include

- **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.
- **check**(*P*): The **check** clause specifies a predicate *P* that must be satisfied by every tuple in the relation.

The **create table** command also includes other integrity constraints, which we shall discuss in Chapter 6.

Figure 4.8 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. Similarly, it flags an error and prevents the update if the **check** condition on the tuple fails.

By default **null** is a legal value for every attribute in SQL, unless the attribute is specifically stated to be **not null**. An attribute can be declared to be not null in the following way:

account-number **char**(10) **not null**

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          integer,
     primary key (branch-name),
     check (assets >= 0))

create table account
    (account-number  char(10),
     branch-name     char(15),
     balance         integer,
     primary key (account-number),
     check (balance >= 0))

create table depositor
    (customer-name   char(20),
     account-number  char(10),
     primary key (customer-name, account-number))
```

Figure 4.8 SQL data definition for part of the bank database.

SQL also supports an integrity constraint

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

The **unique** specification says that attributes $A_{j_1}, A_{j_2}, \ldots, A_{j_m}$ form a candidate key; that is, no two tuples in the relation can be equal on all the primary-key attributes. However, candidate key attributes are permitted to be null unless they have explicitly been declared to be **not null**. Recall that a null value does not equal any other value. The treatment of nulls here is the same as that of the **unique** construct defined in Section 4.6.4.

A common use of the **check** clause is to ensure that attribute values satisfy specified conditions, in effect creating a powerful type system. For instance, the **check** clause in the **create table** command for relation *branch* checks that the value of **assets** is nonnegative. As another example, consider the following:

```
create table student
    (name          char(15) not null,
     student-id    char(10),
     degree-level  char(15),
     primary key (student-id),
     check (degree-level in ('Bachelors', 'Masters', 'Doctorate')))
```

Here, we use the **check** clause to simulate an enumerated type, by specifying that *degree-level* must be one of 'Bachelors', 'Masters', or 'Doctorate'. We consider more general forms of **check** conditions, as well as a class of constraints called referential integrity constraints, in Chapter 6.

A newly created relation is empty initially. We can use the **insert** command to load data into the relation. Many relational-database products have special bulk loader utilities to load an initial set of tuples into a relation.

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.

4.12 Embedded SQL

SQL provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general-purpose programming language. However, a programmer must have access to a database from a general-purpose programming language for at least two reasons:

1. Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Cobol that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.

SQL is designed so that queries written in it can be optimized automatically and executed efficiently—and providing the full power of a programming language makes automatic optimization exceedingly difficult.

2. Nondeclarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL. Applications usually have several components, and querying or updating data is only one component; other components are written in general-purpose programming languages. For an integrated application, the programs written in the programming language must be able to access the database.

The SQL standard defines embeddings of SQL in a variety of programming languages, such as C, Cobol, Pascal, Java, PL/I, and Fortran. A language in which SQL queries are embedded is referred to as a *host* language, and the SQL structures permitted in the host language constitute *embedded* SQL.

Programs written in the host language can use the embedded SQL syntax to access and update data stored in a database. This embedded form of SQL extends the programmer's ability to manipulate the database even further. In embedded SQL, all query processing is performed by the database system, which then makes the result of the query available to the program one tuple (record) at a time.

An embedded SQL program must be processed by a special preprocessor prior to compilation. The preprocessor replaces embedded SQL requests with host-language declarations and procedure calls that allow run-time execution of the database accesses. Then, the resulting program is compiled by the host-language compiler. To identify embedded SQL requests to the preprocessor, we use the EXEC SQL statement; it has the form

EXEC SQL <embedded SQL statement > END-EXEC

The exact syntax for embedded SQL requests depends on the language in which SQL is embedded. For instance, a semicolon is used instead of END-EXEC when SQL is embedded in C. The Java embedding of SQL (called SQLJ) uses the syntax

# SQL { <embedded SQL statement > };

We place the statement SQL INCLUDE in the program to identify the place where the preprocessor should insert the special variables used for communication between the program and the database system. Variables of the host language can be used within embedded SQL statements, but they must be preceded by a colon (:) to distinguish them from SQL variables.

Embedded SQL statements are similar in form to the SQL statements that we described in this chapter. There are, however, several important differences, as we note here.

To write a relational query, we use the **declare cursor** statement. The result of the query is not yet computed. Rather, the program must use the **open** and **fetch** commands (discussed later in this section) to obtain the result tuples.

Consider the banking schema that we have used in this chapter. Assume that we have a host-language variable *amount*, and that we wish to find the names and cities of residence of customers who have more than *amount* dollars in any account. We can write this query as follows:

```
EXEC SQL
        declare c cursor for
        select customer-name, customer-city
        from depositor, customer, account
        where depositor.customer-name = customer.customer-name and
                account.account-number = depositor.account-number and
                account.balance > :amount
END-EXEC
```

The variable *c* in the preceding expression is called a *cursor* for the query. We use this variable to identify the query in the **open** statement, which causes the query to be evaluated, and in the **fetch** statement, which causes the values of one tuple to be placed in host-language variables.

The **open** statement for our sample query is as follows:

```
EXEC SQL open c  END-EXEC
```

This statement causes the database system to execute the query and to save the results within a temporary relation. The query has a host-language variable (:*amount*); the query uses the value of the variable at the time the **open** statement was executed.

If the SQL query results in an error, the database system stores an error diagnostic in the SQL communication-area (SQLCA) variables, whose declarations are inserted by the SQL INCLUDE statement.

An embedded SQL program executes a series of **fetch** statements to retrieve tuples of the result. The **fetch** statement requires one host-language variable for each attribute of the result relation. For our example query, we need one variable to hold the *customer-name* value and another to hold the *customer-city* value. Suppose that those variables are *cn* and *cc*, respectively. Then the statement:

```
EXEC SQL fetch c into :cn, :cc  END-EXEC
```

produces a tuple of the result relation. The program can then manipulate the variables *cn* and *cc* by using the features of the host programming language.

A single **fetch** request returns only one tuple. To obtain all tuples of the result, the program must contain a loop to iterate over all tuples. Embedded SQL assists the programmer in managing this iteration. Although a relation is conceptually a set, the tuples of the result of a query are in some fixed physical order. When the program executes an **open** statement on a cursor, the cursor is set to point to the first tuple of the result. Each time it executes a **fetch** statement, the cursor is updated to point to the next tuple of the result. When no further tuples remain to be processed, the variable SQLSTATE in the SQLCA is set to ’02000’ (meaning “no data”). Thus, we can use a **while** loop (or equivalent loop) to process each tuple of the result.

We must use the **close** statement to tell the database system to delete the temporary relation that held the result of the query. For our example, this statement takes the form

EXEC SQL **close** *c* END-EXEC

SQLJ, the Java embedding of SQL, provides a variation of the above scheme, where Java iterators are used in place of cursors. SQLJ associates the results of a query with an iterator, and the next() method of the Java iterator interface can be used to step through the result tuples, just as the preceding examples use **fetch** on the cursor.

Embedded SQL expressions for database modification (**update**, **insert**, and **delete**) do not return a result. Thus, they are somewhat simpler to express. A database-modification request takes the form

EXEC SQL $<$ any valid **update, insert,** or **delete**$>$ END-EXEC

Host-language variables, preceded by a colon, may appear in the SQL database-modification expression. If an error condition arises in the execution of the statement, a diagnostic is set in the SQLCA.

Database relations can also be updated through cursors. For example, if we want to add 100 to the *balance* attribute of every *account* where the branch name is "Perryridge", we could declare a cursor as follows.

```
declare c cursor for
select *
from account
where branch-name = 'Perryridge'
for update
```

We then iterate through the tuples by performing **fetch** operations on the cursor (as illustrated earlier), and after fetching each tuple we execute the following code

```
update account
set balance = balance + 100
where current of c
```

Embedded SQL allows a host-language program to access the database, but it provides no assistance in presenting results to the user or in generating reports. Most commercial database products include tools to assist application programmers in creating user interfaces and formatted reports. We discuss such tools in Chapter 5 (Section 5.3).

4.13 Dynamic SQL

The *dynamic SQL* component of SQL allows programs to construct and submit SQL queries at run time. In contrast, embedded SQL statements must be completely present at compile time; they are compiled by the embedded SQL preprocessor. Using dynamic SQL, programs can create SQL queries as strings at run time (perhaps based on input from the user) and can either have them executed immediately or have them *prepared* for subsequent use. Preparing a dynamic SQL statement compiles it, and subsequent uses of the prepared statement use the compiled version.

SQL defines standards for embedding dynamic SQL calls in a host language, such as C, as in the following example.

```
char * sqlprog = "update account set balance = balance *1.05
          where account-number = ?"
EXEC SQL prepare dynprog from :sqlprog;
char account[10] = "A-101";
EXEC SQL execute dynprog using :account;
```

The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.

However, the syntax above requires extensions to the language or a preprocessor for the extended language. An alternative that is very widely used is to use an application program interface to send SQL queries or updates to a database system, and not make any changes in the programming language itself.

In the rest of this section, we look at two standards for connecting to an SQL database and performing queries and updates. One, ODBC, is an application program interface for the C language, while the other, JDBC, is an application program interface for the Java language.

To understand these standards, we need to understand the concept of SQL sessions. The user or application *connects* to an SQL server, establishing a session; executes a series of statements; and finally *disconnects* the session. Thus, all activities of the user or application are in the context of an SQL session. In addition to the normal SQL commands, a session can also contain commands to *commit* the work carried out in the session, or to *rollback* the work carried out in the session.

4.13.1 ODBC∗∗

The **Open DataBase Connectivity** (ODBC) standard defines a way for an application program to communicate with a database server. ODBC defines an **application program interface (API)** that applications can use to open a connection with a database, send queries and updates, and get back results. Applications such as graphical user interfaces, statistics packages, and spreadsheets can make use of the same ODBC API to connect to any database server that supports ODBC.

Each database system supporting ODBC provides a library that must be linked with the client program. When the client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.

Figure 4.9 shows an example of C code using the ODBC API. The first step in using ODBC to communicate with a server is to set up a connection with the server. To do so, the program first allocates an SQL environment, then a database connection handle. ODBC defines the types HENV, HDBC, and RETCODE. The program then opens the database connection by using SQLConnect. This call takes several parameters, in-

```
int ODBCexample()
{
    RETCODE error;
    HENV env; /* environment */
    HDBC conn; /* database connection */

    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, ”aura.bell-labs.com”, SQL_NTS, ”avi”, SQL_NTS,
                ”avipasswd”, SQL_NTS);
    {
        char branchname[80];
        float balance;
        int lenOut1, lenOut2;
        HSTMT stmt;

        SQLAllocStmt(conn, &stmt);
        char * sqlquery = ”select branch_name, sum (balance)
                            from account
                            group by branch_name”;
        error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
        if (error == SQL_SUCCESS) {
            SQLBindCol(stmt, 1, SQL_C_CHAR, branchname , 80, &lenOut1);
            SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance, 0 , &lenOut2);
            while (SQLFetch(stmt) >= SQL_SUCCESS) {
                printf (” %s %g\n”, branchname, balance);
            }
        }
    }
    SQLFreeStmt(stmt, SQL_DROP);
    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
}
```

Figure 4.9 ODBC code example.

cluding the connection handle, the server to which to connect, the user identifier, and the password for the database. The constant SQL_NTS denotes that the previous argument is a null-terminated string.

Once the connection is set up, the program can send SQL commands to the database by using SQLExecDirect C language variables can be bound to attributes of the query result, so that when a result tuple is fetched using SQLFetch, its attribute values are stored in corresponding C variables. The SQLBindCol function does this task; the second argument identifies the position of the attribute in the query result, and the third argument indicates the type conversion required from SQL to C. The next argument gives the address of the variable. For variable-length types like character arrays, the last two arguments give the maximum length of the variable and a location where the actual length is to be stored when a tuple is fetched. A negative value returned for the length field indicates that the value is **null**.

The SQLFetch statement is in a **while** loop that gets executed until SQLFetch returns a value other than SQL_SUCCESS. On each fetch, the program stores the values in C variables as specified by the calls on SQLBindCol and prints out these values.

At the end of the session, the program frees the statement handle, disconnects from the database, and frees up the connection and SQL environment handles. Good programming style requires that the result of every function call must be checked to make sure there are no errors; we have omitted most of these checks for brevity.

It is possible to create an SQL statement with parameters; for example, consider the statement insert into account values(?,?,?). The question marks are placeholders for values which will be supplied later. The above statement can be "prepared," that is, compiled at the database, and repeatedly executed by providing actual values for the placeholders—in this case, by providing an account number, branch name, and balance for the relation *account*.

ODBC defines functions for a variety of tasks, such as finding all the relations in the database and finding the names and types of columns of a query result or a relation in the database.

By default, each SQL statement is treated as a separate transaction that is committed automatically. The call SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0) turns off automatic commit on connection conn, and transactions must then be committed explicitly by SQLTransact(conn, SQL_COMMIT) or rolled back by SQLTransact(conn, SQL_ROLLBACK).

The more recent versions of the ODBC standard add new functionality. Each version defines *conformance levels*, which specify subsets of the functionality defined by the standard. An ODBC implementation may provide only core level features, or it may provide more advanced (level 1 or level 2) features. Level 1 requires support for fetching information about the catalog, such as information about what relations are present and the types of their attributes. Level 2 requires further features, such as ability to send and retrieve arrays of parameter values and to retrieve more detailed catalog information.

The more recent SQL standards (SQL-92 and SQL:1999) define a **call level interface (CLI)** that is similar to the ODBC interface, but with some minor differences.

4.13.2 JDBC∗∗

The **JDBC** standard defines an API that Java programs can use to connect to database servers. (The word JDBC was originally an abbreviation for "Java Database Connectivity", but the full form is no longer used.) Figure 4.10 shows an example Java program that uses the JDBC interface. The program must first open a connection to a database, and can then execute SQL statements, but before opening a connection, it loads the appropriate drivers for the database by using Class.forName. The first parameter to the getConnection call specifies the machine name where the server

```
public static void JDBCexample(String dbid, String userid, String passwd)
{
    try
    {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection conn = DriverManager.getConnection(
                "jdbc:oracle:thin:@aura.bell-labs.com:2000:bankdb",
                userid, passwd);
        Statement stmt = conn.createStatement();
        try {
            stmt.executeUpdate(
                "insert into account values('A-9732', 'Perryridge', 1200)");
        } catch (SQLException sqle)
        {
            System.out.println("Could not insert tuple. " + sqle);
        }
        ResultSet rset = stmt.executeQuery(
                "select branch_name, avg (balance)
                from account
                group by branch_name");
        while (rset.next()) {
            System.out.println(rset.getString("branch_name") + " " +
                    rset.getFloat(2));
        }
        stmt.close();
        conn.close();
    }
    catch (SQLException sqle)
    {
        System.out.println("SQLException : " + sqle);
    }
}
```

Figure 4.10 An example of JDBC code.

runs (in our example, aura.bell-labs.com), the port number it uses for communication (in our example, 2000). The parameter also specifies which schema on the server is to be used (in our example, bankdb), since a database server may support multiple schemas. The first parameter also specifies the protocol to be used to communicate with the database (in our example, jdbc:oracle:thin:). Note that JDBC specifies only the API, not the communication protocol. A JDBC driver may support multiple protocols, and we must specify one supported by both the database and the driver. The other two arguments to getConnection are a user identifier and a password.

The program then creates a statement handle on the connection and uses it to execute an SQL statement and get back results. In our example, stmt.executeUpdate executes an update statement. The try { ... } catch { ... } construct permits us to

```
PreparedStatement pStmt = conn.prepareStatement(
                    ”insert into account values(?,?,?)”);
pStmt.setString(1, ”A-9732”);
pStmt.setString(2, ”Perryridge”);
pStmt.setInt(3, 1200);
pStmt.executeUpdate();
pStmt.setString(1, ”A-9733”);
pStmt.executeUpdate();
```

Figure 4.11 Prepared statements in JDBC code.

catch any exceptions (error conditions) that arise when JDBC calls are made, and print an appropriate message to the user.

The program can execute a query by using stmt.executeQuery. It can retrieve the set of rows in the result into a ResultSet and fetch them one tuple at a time using the next() function on the result set. Figure 4.10 shows two ways of retrieving the values of attributes in a tuple: using the name of the attribute (*branch-name*) and using the position of the attribute (2, to denote the second attribute).

We can also create a prepared statement in which some values are replaced by “?”, thereby specifying that actual values will be provided later. We can then provide the values by using setString(). The database can compile the query when it is prepared, and each time it is executed (with new values), the database can reuse the previously compiled form of the query. The code fragment in Figure 4.11 shows how prepared statements can be used.

JDBC provides a number of other features, such as **updatable result sets**. It can create an updatable result set from a query that performs a selection and/or a projection on a database relation. An update to a tuple in the result set then results in an update to the corresponding tuple of the database relation. JDBC also provides an API to examine database schemas and to find the types of attributes of a result set.

For more information about JDBC, refer to the bibliographic information at the end of the chapter.

4.14 Other SQL Features ∗∗

The SQL language has grown over the past two decades from a simple language with a few features to a rather complex language with features to satisfy many different types of users. We covered the basics of SQL earlier in this chapter. In this section we introduce the reader to some of the more complex features of SQL.

4.14.1 Schemas, Catalogs, and Environments

To understand the motivation for schemas and catalogs, consider how files are named in a file system. Early file systems were flat; that is, all files were stored in a single directory. Current generation file systems of course have a directory structure, with files stored within subdirectories. To name a file uniquely, we must specify the full path name of the file, for example, /users/avi/db-book/chapter4.tex.

Like early file systems, early database systems also had a single name space for all relations. Users had to coordinate to make sure they did not try to use the same name for different relations. Contemporary database systems provide a three-level hierarchy for naming relations. The top level of the hierarchy consists of **catalogs**, each of which can contain **schemas**. SQL objects such as relations and views are contained within a **schema**.

In order to perform any actions on a database, a user (or a program) must first *connect* to the database. The user must provide the user name and usually, a secret password for verifying the identity of the user, as we saw in the ODBC and JDBC examples in Sections 4.13.1 and 4.13.2. Each user has a default catalog and schema, and the combination is unique to the user. When a user connects to a database system, the default catalog and schema are set up for for the connection; this corresponds to the current directory being set to the user's home directory when the user logs into an operating system.

To identify a relation uniquely, a three-part name must be used, for example,

catalog5.bank-schema.account

We may omit the catalog component, in which case the catalog part of the name is considered to be the default catalog for the connection. Thus if catalog5 is the default catalog, we can use bank-schema.account to identify the same relation uniquely. Further, we may also omit the schema name, and the schema part of the name is again considered to be the default schema for the connection. Thus we can use just account if the default catalog is catalog5 and the default schema is bank-schema.

With multiple catalogs and schemas available, different applications and different users can work independently without worrying about name clashes. Moreover, multiple versions of an application—one a production version, other test versions—can run on the same database system.

The default catalog and schema are part of an **SQL environment** that is set up for each connection. The environment additionally contains the user identifier (also referred to as the *authorization identifier*). All the usual SQL statements, including the DDL and DML statements, operate in the context of a schema. We can create and drop schemas by means of **create schema** and **drop schema** statements. Creation and dropping of catalogs is implementation dependent and not part of the SQL standard.

4.14.2 Procedural Extensions and Stored Procedures

SQL provides a **module** language, which allows procedures to be defined in SQL. A module typically contains multiple SQL procedures. Each procedure has a name, optional arguments, and an SQL statement. An extension of the SQL-92 standard language also permits procedural constructs, such as **for**, **while**, and **if-then-else**, and compound SQL statements (multiple SQL statements between a **begin** and an **end**).

We can store procedures in the database and then execute them by using the **call** statement. Such procedures are also called **stored procedures**. Stored procedures are particularly useful because they permit operations on the database to be made available to external applications, without exposing any of the internal details of the database.

Chapter 9 covers procedural extensions of SQL as well as many other new features of SQL:1999.

4.15 Summary

- Commercial database systems do not use the terse, formal query languages covered in Chapter 3. The widely used SQL language, which we studied in this chapter, is based on the formal relational algebra, but includes much "syntactic sugar."
- 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.
- 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. A transaction consists of a sequence of operations, which must appear to be atomic. That is, all the operations are carried out successfully, or none is carried out. In practice, if a transaction cannot complete successfully, any partial actions it carried out are undone.
- Modifications to the database may lead to the generation of null values in tuples. We discussed how nulls can be introduced, and how the SQL query language handles queries on relations containing null values.
- 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 Chapter 6.
- SQL queries can be invoked from host languages, via embedded and dynamic SQL. The ODBC and JDBC standards define application program interfaces to access SQL databases from C and Java language programs. Increasingly, programmers use these APIs to access databases.
- We also saw a brief overview of some advanced features of SQL, such as procedural extensions, catalogs, schemas and stored procedures.

Review Terms

- DDL: data definition language
- DML: data manipulation language
- **select** clause
- **from** clause
- **where** clause
- **as** clause
- Tuple variable
- **order** by clause
- Duplicates
- 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**
- Views
- Derived relations (in **from** clause)
- **with** clause
- Database modification
  - ☐ **delete, insert, update**
  - ☐ View update
- Join types
  - ☐ Inner and outer join
  - ☐ left, right and full outer join
  - ☐ natural, using, and on
- Transaction
- Atomicity
- Index
- Schema
- Domains
- Embedded SQL
- Dynamic SQL
- ODBC
- JDBC
- Catalog
- Stored procedures

Exercises

4.1 Consider the insurance database of Figure 4.12, 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. Find the number of accidents in which the cars belonging to "John Smith" were involved.
c. Add a new accident to the database; assume any values for required attributes.
d. Delete the Mazda belonging to "John Smith".
e. Update the damage amount for the car with license number "AABB2000" in the accident with report number "AR2197" to $3000.

4.2 Consider the employee database of Figure 4.13, 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.

person (<u>*driver-id#*</u>, *name*, *address*)
car (<u>*license*</u>, *model*, *year*)
accident (<u>*report-number*</u>, <u>*date*</u>, *location*)
owns (<u>*driver-id#*</u>, <u>*license*</u>)
participated (<u>*driver-id*</u>, <u>*car*</u>, <u>*report-number*</u>, *damage-amount*)

Figure 4.12 Insurance database.

employee (<u>*employee-name*</u>, *street*, *city*)
works (<u>*employee-name*</u>, *company-name*, *salary*)
company (<u>*company-name*</u>, *city*)
manages (<u>*employee-name*</u>, *manager-name*)

Figure 4.13 Employee database.

b. Find the names and cities of residence of all employees who work for First Bank Corporation.
c. Find the names, street addresses, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000.
d. Find all employees in the database who live in the same cities as the companies for which they work.
e. Find all employees in the database who live in the same cities and on the same streets as do their managers.
f. Find all employees in the database who do not work for First Bank Corporation.
g. Find all employees in the database who earn more than each employee of Small Bank Corporation.
h. Assume that the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.
i. Find all employees who earn more than the average salary of all employees of their company.
j. Find the company that has the most employees.
k. Find the company that has the smallest payroll.
l. Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

4.3 Consider the relational database of Figure 4.13. 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 employees of First Bank Corporation a 10 percent raise.
c. Give all managers of First Bank Corporation a 10 percent raise.
d. 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.
e. Delete all tuples in the *works* relation for employees of Small Bank Corporation.

4.4 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. $\Pi_{A,F}\,(\sigma_{C\,=\,D}(r \;\times\; s))$

4.5 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. $\Pi_{AB}(r_1) \;\bowtie\; \Pi_{BC}(r_2)$

4.6 Let $R \;=\; (A, B)$ and $S \;=\; (A, C)$, and let $r(R)$ and $s(S)$ be relations. Write an expression in SQL for each of the queries below:

a. $\{< a > \mid \exists\, b\,(< a, b > \in\; r \;\wedge\; b \;=\; 17)\}$
b. $\{< a, b, c > \;\mid\; < a, b > \in\; r \;\wedge\; < a, c > \in\; s\}$
c. $\{< a > \mid \exists\, c\,(< a, c > \in\; s \;\wedge\; \exists\, b_1, b_2\;(< a, b_1 > \in\; r \wedge\; < c,\, b_2 > \in\; r \;\wedge\; b_1 \;>\; b_2))\}$

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

4.8 Consider the relational database of Figure 4.13. 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.

4.9 Consider the SQL query

> **select** *p.a*1
> **from** *p*, *r*1, *r*2
> **where** *p.a*1 = *r*1.*a*1 **or** *p.a*1 = *r*2.*a*1

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.

4.10 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** clauser.

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

4.11 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.

4.12 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. Show how to express the **coalesce** operation using the **case** operation.

4.13 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*.

4.14 Give an SQL schema definition for the employee database of Figure 4.13. Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema.

4.15 Write **check** conditions for the schema you defined in Exercise 4.14 to ensure that:

a. Every employee works for a company located in the same city as the city in which the employee lives.
b. No employee earns a salary higher than that of his manager.

4.16 Describe the circumstances in which you would choose to use embedded SQL rather than SQL alone or only a general-purpose programming language.

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]. Melton and Eisenberg [2000] provides a guide to SQLJ, JDBC, and related technologies. More information on SQLJ and SQLJ software can be obtained from http://www.sqlj.org. Date and Darwen [1997] and Date [1993a] include a critique of SQL-92.

Eisenberg and Melton [1999] provide an overview of SQL:1999. The standard is published as a sequence of five ISO/IEC standards documents, with several more parts describing various extensions under development. Part 1 (SQL/Framework), gives an overview of the other parts. Part 2 (SQL/Foundation) outlines the basics of the language. Part 3 (SQL/CLI) describes the Call-Level Interface. Part 4 (SQL/PSM) describes Persistent Stored Modules, and Part 5 (SQL/Bindings) describes host language bindings. The standard is useful to database implementers but is very hard to read. If you need them, you can purchase them electronically from the Web site http://webstore.ansi.org.

Many database products support SQL features beyond those specified in the standards, 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. http://java.sun.com/docs/books/tutorial is an excellent source for more (and up-to-date) information on JDBC, and on Java in general. References to books on Java (including JDBC) are also available at this URL. The ODBC API is described in Microsoft [1997] and Sanders [1998].

The processing of SQL queries, including algorithms and performance issues, is discussed in Chapters 13 and 14. Bibliographic references on these matters appear in that chapter.

C H A P T E R 5

Other Relational Languages

In Chapter 4, we described SQL—the most influential commercial relational-database language. In this chapter, we study two more languages: QBE and Datalog. Unlike SQL, QBE is a graphical language, where queries *look* like tables. QBE and its variants are widely used in database systems on personal computers. Datalog has a syntax modeled after the Prolog language. Although not used commercially at present, Datalog has been used in several research database systems.

Here, we present fundamental constructs and concepts rather than a complete users' guide for these languages. Keep in mind that individual implementations of a language may differ in details, or may support only a subset of the full language.

In this chapter, we also study forms interfaces and tools for generating reports and analyzing data. While these are not strictly speaking languages, they form the main interface to a database for many users. In fact, most users do not perform explicit querying with a query language at all, and access data only via forms, reports, and other data analysis tools.

5.1 Query-by-Example

Query-by-Example (QBE) is the name of both a data-manipulation language and an early database system that included this language. The QBE database system was developed at IBM's T. J. Watson Research Center in the early 1970s. The QBE data-manipulation language was later used in IBM's Query Management Facility (QMF). Today, many database systems for personal computers support variants of QBE language. In this section, we consider only the data-manipulation language. It has two distinctive features:

1. Unlike most query languages and programming languages, QBE has a **two-dimensional syntax**: Queries *look* like tables. A query in a one-dimensional

language (for example, SQL) *can* be written in one (possibly long) line. A two-dimensional language *requires* two dimensions for its expression. (There is a one-dimensional version of QBE, but we shall not consider it in our discussion).

2. QBE queries are expressed "by example." Instead of giving a procedure for obtaining the desired answer, the user gives an example of what is desired. The system generalizes this example to compute the answer to the query.

Despite these unusual features, there is a close correspondence between QBE and the domain relational calculus.

We express queries in QBE by **skeleton tables**. These tables show the relation schema, as in Figure 5.1. Rather than clutter the display with all skeletons, the user selects those skeletons needed for a given query and fills in the skeletons with **example rows**. An example row consists of constants and *example elements*, which are domain variables. To avoid confusion between the two, QBE uses an underscore character (_) before domain variables, as in _*x*, and lets constants appear without any qualification.

branch	*branch-name*	*branch-city*	*assets*

customer	*customer-name*	*customer-street*	*customer-city*

loan	*loan-number*	*branch-name*	*amount*

borrower	*customer-name*	*loan-number*

account	*account-number*	*branch-name*	*balance*

depositor	*customer-name*	*account-number*

Figure 5.1 QBE skeleton tables for the bank example.

This convention is in contrast to those in most other languages, in which constants are quoted and variables appear without any qualification.

5.1.1 Queries on One Relation

Returning to our ongoing bank example, to find all loan numbers at the Perryridge branch, we bring up the skeleton for the *loan* relation, and fill it in as follows:

loan	*loan-number*	*branch-name*	*amount*
	P._*x*	Perryridge	

This query tells the system to look for tuples in *loan* that have "Perryridge" as the value for the *branch-name* attribute. For each such tuple, the system assigns the value of the *loan-number* attribute to the variable *x*. It "prints" (actually, displays) the value of the variable *x*, because the command P. appears in the *loan-number* column next to the variable *x*. Observe that this result is similar to what would be done to answer the domain-relational-calculus query

$$\{\langle x\rangle \mid \exists\, b, a(\langle x, b, a\rangle \in loan \wedge b = \text{"Perryridge"})\}$$

QBE assumes that a blank position in a row contains a unique variable. As a result, if a variable does not appear more than once in a query, it may be omitted. Our previous query could thus be rewritten as

loan	*loan-number*	*branch-name*	*amount*
	P.	Perryridge	

QBE (unlike SQL) performs duplicate elimination automatically. To suppress duplicate elimination, we insert the command ALL. after the P. command:

loan	*loan-number*	*branch-name*	*amount*
	P.ALL.	Perryridge	

To display the entire *loan* relation, we can create a single row consisting of P. in every field. Alternatively, we can use a shorthand notation by placing a single P. in the column headed by the relation name:

loan	*loan-number*	*branch-name*	*amount*
P.			

QBE allows queries that involve arithmetic comparisons (for example, >), rather than equality comparisons, as in "Find the loan numbers of all loans with a loan amount of more than $700":

loan	*loan-number*	*branch-name*	*amount*
	P.		>700

Comparisons can involve only one arithmetic expression on the right-hand side of the comparison operation (for example, $> (\_x + \_y - 20)$). The expression can include both variables and constants. The space on the left-hand side of the comparison operation must be blank. The arithmetic operations that QBE supports are $=$, $<$, $\leq$, $>$, $\geq$, and $\neg$.

Note that requiring the left-hand side to be blank implies that we cannot compare two distinct named variables. We shall deal with this difficulty shortly.

As yet another example, consider the query "Find the names of all branches that are not located in Brooklyn." This query can be written as follows:

branch	*branch-name*	*branch-city*	*assets*
	P.	¬ Brooklyn	

The primary purpose of variables in QBE is to force values of certain tuples to have the same value on certain attributes. Consider the query "Find the loan numbers of all loans made jointly to Smith and Jones":

borrower	*customer-name*	*loan-number*
	"Smith"	P._*x*
	"Jones"	_*x*

To execute this query, the system finds all pairs of tuples in *borrower* that agree on the *loan-number* attribute, where the value for the *customer-name* attribute is "Smith" for one tuple and "Jones" for the other. The system then displays the value of the *loan-number* attribute.

In the domain relational calculus, the query would be written as

$$\{\langle l \rangle \mid \exists\, x\, (\langle x, l \rangle \in borrower \wedge x = \text{"Smith"})$$

$$\wedge \exists\, x\, (\langle x, l \rangle \in borrower \wedge x = \text{"Jones"})\}$$

As another example, consider the query "Find all customers who live in the same city as Jones":

customer	*customer-name*	*customer-street*	*customer-city*
	P._*x*		_*y*
	Jones		_*y*

5.1.2 Queries on Several Relations

QBE allows queries that span several different relations (analogous to Cartesian product or natural join in the relational algebra). The connections among the various relations are achieved through variables that force certain tuples to have the same value on certain attributes. As an illustration, suppose that we want to find the names of all customers who have a loan from the Perryridge branch. This query can be written as

loan	*loan-number*	*branch-name*	*amount*
	_*x*	Perryridge	

borrower	*customer-name*	*loan-number*
	P._*y*	_*x*

To evaluate the preceding query, the system finds tuples in *loan* with "Perryridge" as the value for the *branch-name* attribute. For each such tuple, the system finds tuples in *borrower* with the same value for the *loan-number* attribute as the *loan* tuple. It displays the values for the *customer-name* attribute.

We can use a technique similar to the preceding one to write the query "Find the names of all customers who have both an account and a loan at the bank":

depositor	*customer-name*	*account-number*
	P._*x*	

borrower	*customer-name*	*loan-number*
	_*x*	

Now consider the query "Find the names of all customers who have an account at the bank, but who do not have a loan from the bank." We express queries that involve negation in QBE by placing a **not** sign (¬) under the relation name and next to an example row:

depositor	*customer-name*	*account-number*
	P._*x*	

borrower	*customer-name*	*loan-number*
¬	_*x*	

Compare the preceding query with our earlier query "Find the names of all customers who have both an account and a loan at the bank." The only difference is the ¬ appearing next to the example row in the *borrower* skeleton. This difference, however, has a major effect on the processing of the query. QBE finds all *x* values for which

1. There is a tuple in the *depositor* relation whose *customer-name* is the domain variable *x*.
2. There is no tuple in the *borrower* relation whose *customer-name* is the same as in the domain variable *x*.

The ¬ can be read as "there does not exist."

The fact that we placed the ¬ under the relation name, rather than under an attribute name, is important. A ¬ under an attribute name is shorthand for $\neq$. Thus, to find all customers who have at least two accounts, we write

depositor	*customer-name*	*account-number*
	P._*x*	_*y*
	_*x*	¬ _*y*

In English, the preceding query reads "Display all *customer-name* values that appear in at least two tuples, with the second tuple having an *account-number* different from the first."

5.1.3 The Condition Box

At times, it is either inconvenient or impossible to express all the constraints on the domain variables within the skeleton tables. To overcome this difficulty, QBE includes a **condition box** feature that allows the expression of general constraints over any of the domain variables. QBE allows logical expressions to appear in a condition box. The logical operators are the words **and** and **or**, or the symbols "&" and "|".

For example, the query "Find the loan numbers of all loans made to Smith, to Jones (or to both jointly)" can be written as

borrower	*customer-name*	*loan-number*
	_*n*	P._*x*

conditions
_*n* = Smith **or** _*n* = Jones

It is possible to express the above query without using a condition box, by using P. in multiple rows. However, queries with P. in multiple rows are sometimes hard to understand, and are best avoided.

As yet another example, suppose that we modify the final query in Section 5.1.2 to be "Find all customers who are not named 'Jones' and who have at least two accounts." We want to include an "$x \neq$ Jones" constraint in this query. We do that by bringing up the condition box and entering the constraint "$x \neg =$ Jones":

conditions
x ¬ = Jones

Turning to another example, to find all account numbers with a balance between \$1300 and \$1500, we write

account	*account-number*	*branch-name*	*balance*
	P.		_*x*

conditions
_*x* ≥ 1300
_*x* ≤ 1500

As another example, consider the query "Find all branches that have assets greater than those of at least one branch located in Brooklyn." This query can be written as

branch	*branch-name*	*branch-city*	*assets*
	P._*x*		_*y*
		Brooklyn	_*z*

conditions
_*y* > _*z*

QBE allows complex arithmetic expressions to appear in a condition box. We can write the query "Find all branches that have assets that are at least twice as large as the assets of one of the branches located in Brooklyn" much as we did in the preceding query, by modifying the condition box to

conditions
_*y* ≥ 2 * _*z*

To find all account numbers of account with a balance between $1300 and $2000, but not exactly $1500, we write

account	*account-number*	*branch-name*	*balance*
	P.		_*x*

conditions
_*x* = (≥ 1300 **and** ≤ 2000 **and** ¬ 1500)

QBE uses the **or** construct in an unconventional way to allow comparison with a set of constant values. To find all branches that are located in either Brooklyn or Queens, we write

branch	*branch-name*	*branch-city*	*assets*
	P.	_*x*	

conditions
_*x* = (Brooklyn **or** Queens)

5.1.4 The Result Relation

The queries that we have written thus far have one characteristic in common: The results to be displayed appear in a single relation schema. If the result of a query includes attributes from several relation schemas, we need a mechanism to display the desired result in a single table. For this purpose, we can declare a temporary *result* relation that includes all the attributes of the result of the query. We print the desired result by including the command P. in only the *result* skeleton table.

As an illustration, consider the query "Find the *customer-name*, *account-number*, and *balance* for all accounts at the Perryridge branch." In relational algebra, we would construct this query as follows:

1. Join *depositor* and *account*.
2. Project *customer-name*, *account-number*, and *balance*.

To construct the same query in QBE, we proceed as follows:

1. Create a skeleton table, called *result*, with attributes *customer-name*, *account-number*, and *balance*. The name of the newly created skeleton table (that is, *result*) must be different from any of the previously existing database relation names.
2. Write the query.

The resulting query is

account	*account-number*	*branch-name*	*balance*
	_*y*	Perryridge	_*z*

depositor	*customer-name*	*account-number*
	_*x*	_*y*

result	*customer-name*	*account-number*	*balance*
P.	_*x*	_*y*	_*z*

5.1.5 Ordering of the Display of Tuples

QBE offers the user control over the order in which tuples in a relation are displayed. We gain this control by inserting either the command AO. (ascending order) or the command DO. (descending order) in the appropriate column. Thus, to list in ascending alphabetic order all customers who have an account at the bank, we write

depositor	*customer-name*	*account-number*
	P.AO.	

QBE provides a mechanism for sorting and displaying data in multiple columns. We specify the order in which the sorting should be carried out by including, with each sort operator (AO or DO), an integer surrounded by parentheses. Thus, to list all account numbers at the Perryridge branch in ascending alphabetic order with their respective account balances in descending order, we write

account	*account-number*	*branch-name*	*balance*
	P.AO(1).	Perryridge	P.DO(2).

The command P.AO(1). specifies that the account number should be sorted first; the command P.DO(2). specifies that the balances for each account should then be sorted.

5.1.6 Aggregate Operations

QBE includes the aggregate operators AVG, MAX, MIN, SUM, and CNT. We must postfix these operators with ALL. to create a multiset on which the aggregate operation is evaluated. The ALL. operator ensures that duplicates are not eliminated. Thus, to find the total balance of all the accounts maintained at the Perryridge branch, we write

account	*account-number*	*branch-name*	*balance*
		Perryridge	P.SUM.ALL.

We use the operator UNQ to specify that we want duplicates eliminated. Thus, to find the total number of customers who have an account at the bank, we write

depositor	*customer-name*	*account-number*
	P.CNT.UNQ.	

QBE also offers the ability to compute functions on groups of tuples using the G. operator, which is analogous to SQL's **group by** construct. Thus, to find the average balance at each branch, we can write

account	*account-number*	*branch-name*	*balance*
		P.G.	P.AVG.ALL._*x*

The average balance is computed on a branch-by-branch basis. The keyword ALL. in the P.AVG.ALL. entry in the *balance* column ensures that all the balances are considered. If we wish to display the branch names in ascending order, we replace P.G. by P.AO.G.

To find the average account balance at only those branches where the average account balance is more than $1200, we add the following condition box:

conditions
AVG.ALL._*x* > 1200

As another example, consider the query "Find all customers who have accounts at each of the branches located in Brooklyn":

depositor	*customer-name*	*account-number*
	P.G._*x*	_*y*

account	*account-number*	*branch-name*	*balance*
	_*y*	_*z*	

branch	*branch-name*	*branch-city*	*assets*
	_*z*	Brooklyn	
	_*w*	Brooklyn	

conditions
CNT.UNQ._*z* = CNT.UNQ._*w*

The domain variable w can hold the value of names of branches located in Brooklyn. Thus, CNT.UNQ._w is the number of distinct branches in Brooklyn. The domain variable z can hold the value of branches in such a way that both of the following hold:

- The branch is located in Brooklyn.
- The customer whose name is x has an account at the branch.

Thus, CNT.UNQ._z is the number of distinct branches in Brooklyn at which customer x has an account. If CNT.UNQ._z = CNT.UNQ._w, then customer x must have an account at all of the branches located in Brooklyn. In such a case, the displayed result includes x (because of the P.).

5.1.7 Modification of the Database

In this section, we show how to add, remove, or change information in QBE.

5.1.7.1 Deletion

Deletion of tuples from a relation is expressed in much the same way as a query. The major difference is the use of D. in place of P. QBE (unlike SQL), lets us delete whole tuples, as well as values in selected columns. When we delete information in only some of the columns, null values, specified by $-$, are inserted.

We note that a D. command operates on only one relation. If we want to delete tuples from several relations, we must use one D. operator for each relation.

Here are some examples of QBE delete requests:

- Delete customer Smith.

customer	*customer-name*	*customer-street*	*customer-city*
D.	Smith		

- Delete the *branch-city* value of the branch whose name is “Perryridge.”

branch	*branch-name*	*branch-city*	*assets*
	Perryridge	D.	

Thus, if before the delete operation the *branch* relation contains the tuple (Perryridge, Brooklyn, 50000), the delete results in the replacement of the preceding tuple with the tuple (Perryridge, −, 50000).

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

loan	*loan-number*	*branch-name*	*amount*
D.	_*y*		_*x*

borrower	*customer-name*	*loan-number*
D.		_*y*

conditions
_*x* = (≥ 1300 **and** ≤ 1500)

Note that to delete loans we must delete tuples from both the *loan* and *borrower* relations.

- Delete all accounts at all branches located in Brooklyn.

account	*account-number*	*branch-name*	*balance*
D.	_*y*	_*x*	

depositor	*customer-name*	*account-number*
D.		_*y*

branch	*branch-name*	*branch-city*	*assets*
	_*x*	Brooklyn	

Note that, in expressing a deletion, we can reference relations other than those from which we are deleting information.

5.1.7.2 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. We do the insertion by placing the I. operator in the query expression. Obviously, the attribute values for inserted tuples must be members of the attribute’s domain.

The simplest insert is a request to insert one tuple. Suppose that we wish to insert the fact that account A-9732 at the Perryridge branch has a balance of $700. We write

account	*account-number*	*branch-name*	*balance*
I.	A-9732	Perryridge	700

We can also insert a tuple that contains only partial information. To insert information into the *branch* relation about a new branch with name "Capital" and city "Queens," but with a null asset value, we write

branch	*branch-name*	*branch-city*	*assets*
I.	Capital	Queens	

More generally, we might want to insert tuples on the basis of the result of a query. Consider again the situation where we want to provide as a gift, for all loan customers of the Perryridge branch, a new $200 savings account for every loan account that they have, with the loan number serving as the account number for the savings account. We write

account	*account-number*	*branch-name*	*balance*
I.	_*x*	Perryridge	200

depositor	*customer-name*	*account-number*
I.	_*y*	_*x*

loan	*loan-number*	*branch-name*	*amount*
	_*x*	Perryridge	

borrower	*customer-name*	*loan-number*
	_*y*	_*x*

To execute the preceding insertion request, the system must get the appropriate information from the *borrower* relation, then must use that information to insert the appropriate new tuple in the *depositor* and *account* relations.

5.1.7.3 Updates

There are situations in which we wish to change one value in a tuple without changing *all* values in the tuple. For this purpose, we use the U. operator. As we could for insert and delete, we can choose the tuples to be updated by using a query. QBE, however, does not allow users to update the primary key fields.

Suppose that we want to update the asset value of the of the Perryridge branch to $10,000,000. This update is expressed as

branch	*branch-name*	*branch-city*	*assets*
	Perryridge		U.10000000

The blank field of attribute *branch-city* implies that no updating of that value is required.

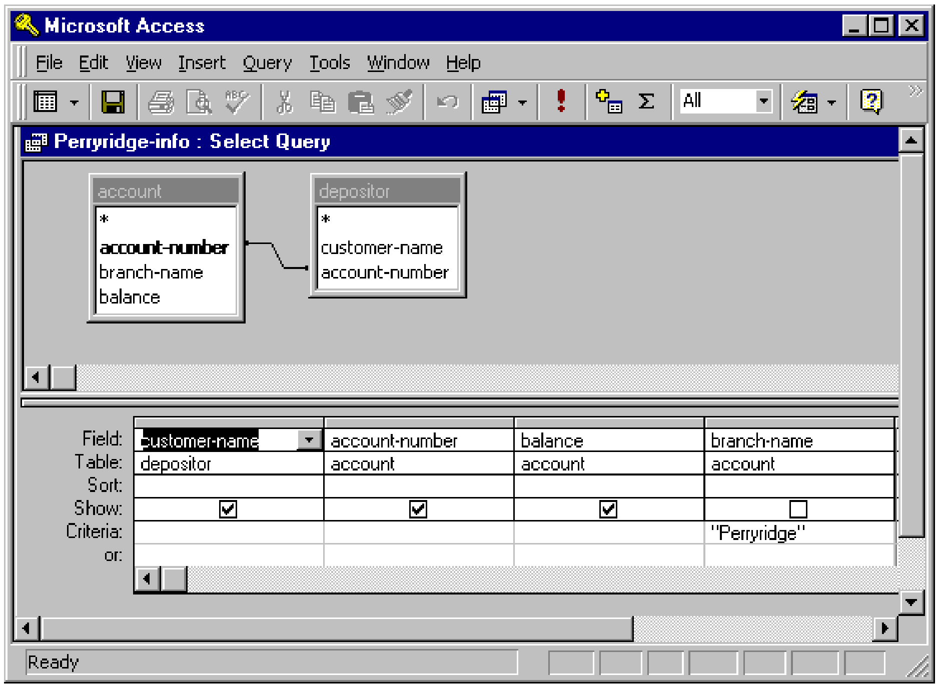
The preceding query updates the assets of the Perryridge branch to $10,000,000, regardless of the old value. There are circumstances, however, where we need to update a value by using the previous value. Suppose that interest payments are being made, and all balances are to be increased by 5 percent. We write

account	*account-number*	*branch-name*	*balance*
			U._*x* * 1.05

This query specifies that we retrieve one tuple at a time from the *account* relation, determine the balance *x*, and update that balance to *x* * 1.05.

5.1.8 QBE in Microsoft Access

In this section, we survey the QBE version supported by Microsoft Access. While the original QBE was designed for a text-based display environment, Access QBE is designed for a graphical display environment, and accordingly is called **graphical query-by-example (GQBE)**.



Field:	customer-name	account-number	balance	branch-name
Table:	depositor	account	account	account
Sort:				
Show:	☑	☑	☑	☐
Criteria:				"Perryridge"
or:				

Figure 5.2 An example query in Microsoft Access QBE.

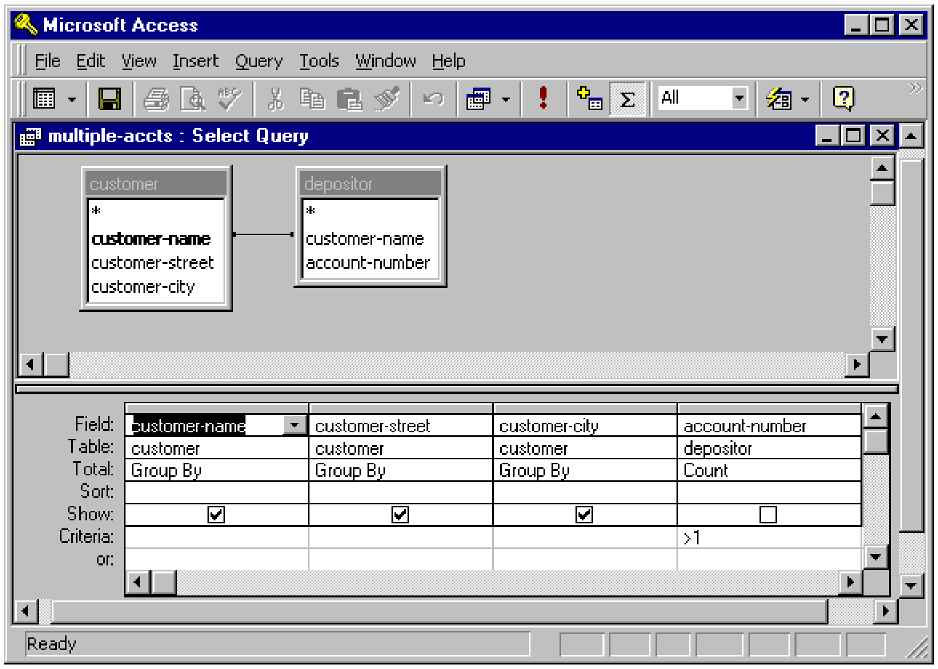
Figure 5.2 shows a sample GQBE query. The query can be described in English as "Find the *customer-name*, *account-number*, and *balance* for all accounts at the Perryridge branch." Section 5.1.4 showed how it is expressed in QBE.

A minor difference in the GQBE version is that the attributes of a table are written one below the other, instead of horizontally. A more significant difference is that the graphical version of QBE uses a line linking attributes of two tables, instead of a shared variable, to specify a join condition.

An interesting feature of QBE in Access is that links between tables are created automatically, on the basis of the attribute name. In the example in Figure 5.2, the two tables *account* and *depositor* were added to the query. The attribute *account-number* is shared between the two selected tables, and the system automatically inserts a link between the two tables. In other words, a natural join condition is imposed by default between the tables; the link can be deleted if it is not desired. The link can also be specified to denote a natural outer-join, instead of a natural join.

Another minor difference in Access QBE is that it specifies attributes to be printed in a separate box, called the **design grid**, instead of using a P. in the table. It also specifies selections on attribute values in the design grid.

Queries involving group by and aggregation can be created in Access as shown in Figure 5.3. The query in the figure finds the name, street, and city of all customers who have more than one account at the bank; we saw the QBE version of the query earlier in Section 5.1.6. The group by attributes as well as the aggregate functions



Field:	customer-name	customer-street	customer-city	account-number
Table:	customer	customer	customer	depositor
Total:	Group By	Group By	Group By	Count
Sort:				
Show:	✓	✓	✓	☐
Criteria:				>1
or:				

Figure 5.3 An aggregation query in Microsoft Access QBE.

are noted in the design grid. If an attribute is to be printed, it must appear in the design grid, and must be specified in the "Total" row to be either a group by, or have an aggregate function applied to it. SQL has a similar requirement. Attributes that participate in selection conditions but are not to be printed can alternatively be marked as "Where" in the row "Total", indicating that the attribute is neither a group by attribute, nor one to be aggregated on.

Queries are created through a graphical user interface, by first selecting tables. Attributes can then be added to the design grid by dragging and dropping them from the tables. Selection conditions, grouping and aggregation can then be specified on the attributes in the design grid. Access QBE supports a number of other features too, including queries to modify the database through insertion, deletion, or update.

5.2 Datalog

Datalog is a nonprocedural query language based on the logic-programming language Prolog. As in the relational calculus, a user describes the information desired without giving a specific procedure for obtaining that information. The syntax of Datalog resembles that of Prolog. However, the meaning of Datalog programs is defined in a purely declarative manner, unlike the more procedural semantics of Prolog, so Datalog simplifies writing simple queries and makes query optimization easier.

5.2.1 Basic Structure

A Datalog program consists of a set of **rules**. Before presenting a formal definition of Datalog rules and their formal meaning, we consider examples. Consider a Datalog rule to define a view relation *v1* containing account numbers and balances for accounts at the Perryridge branch with a balance of over $700:

$$v1(A, B) \text{ :– } account(A, \text{“Perryridge”}, B), B > 700$$

Datalog rules define views; the preceding rule **uses** the relation *account*, and **defines** the view relation *v1*. The symbol :– is read as "if," and the comma separating the "*account*(A, "Perryridge", B)" from "$B > 700$" is read as "and." Intuitively, the rule is understood as follows:

for all A, B
if $(A, \text{“Perryridge”}, B) \in account$ **and** $B > 700$
then $(A, B) \in v1$

Suppose that the relation *account* is as shown in Figure 5.4. Then, the view relation *v1* contains the tuples in Figure 5.5.

To retrieve the balance of account number A-217 in the view relation *v1*, we can write the following query:

$$?\ v1(\text{“A-217”}, B)$$

The answer to the query is

$$(\text{A-217}, 750)$$

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-215	Mianus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Perryridge	900
A-222	Redwood	700
A-217	Perryridge	750

Figure 5.4 The *account* relation.

To get the account number and balance of all accounts in relation *v1*, where the balance is greater than 800, we can write

$$? \ v1(A, B), B > 800$$

The answer to this query is

(A-201, 900)

In general, we need more than one rule to define a view relation. Each rule defines a set of tuples that the view relation must contain. The set of tuples in the view relation is then defined as the union of all these sets of tuples. The following Datalog program specifies the interest rates for accounts:

$$\begin{aligned} &\textit{interest-rate}(A, 5) \ :- \textit{account}(A, N, B), B < 10000 \\ &\textit{interest-rate}(A, 6) \ :- \textit{account}(A, N, B), B >= 10000 \end{aligned}$$

The program has two rules defining a view relation *interest-rate*, whose attributes are the account number and the interest rate. The rules say that, if the balance is less than \$10000, then the interest rate is 5 percent, and if the balance is greater than or equal to \$10000, the interest rate is 6 percent.

Datalog rules can also use negation. The following rules define a view relation *c* that contains the names of all customers who have a deposit, but have no loan, at the bank:

$$\begin{aligned} &c(N) \ :- \textit{depositor}(N, A), \textbf{not } \textit{is-borrower}(N) \\ &\textit{is-borrower}(N) \ :- \textit{borrower}(N, L), \end{aligned}$$

Prolog and most Datalog implementations recognize attributes of a relation by position and omit attribute names. Thus, Datalog rules are compact, compared to SQL

account-number	*balance*
A-201	900
A-217	750

Figure 5.5 The *v1* relation.

queries. However, when relations have a large number of attributes, or the order or number of attributes of relations may change, the positional notation can be cumbersome and error prone. It is not hard to create a variant of Datalog syntax using named attributes, rather than positional attributes. In such a system, the Datalog rule defining *v1* can be written as

v1(*account-number* A, *balance* B) :–
  account(*account-number* A, *branch-name* "Perryridge", *balance* B),
  $B > 700$

Translation between the two forms can be done without significant effort, given the relation schema.

5.2.2 Syntax of Datalog Rules

Now that we have informally explained rules and queries, we can formally define their syntax; we discuss their meaning in Section 5.2.3. We use the same conventions as in the relational algebra for denoting relation names, attribute names, and constants (such as numbers or quoted strings). We use uppercase (capital) letters and words starting with uppercase letters to denote variable names, and lowercase letters and words starting with lowercase letters to denote relation names and attribute names. Examples of constants are 4, which is a number, and "John," which is a string; X and *Name* are variables. A **positive literal** has the form

$$p(t_1, t_2, \ldots, t_n)$$

where p is the name of a relation with n attributes, and $t_1, t_2, \ldots, t_n$ are either constants or variables. A **negative literal** has the form

$$\textbf{not } p(t_1, t_2, \ldots, t_n)$$

where relation p has n attributes. Here is an example of a literal:

account(A, "Perryridge", B)

Literals involving arithmetic operations are treated specially. For example, the literal $B > 700$, although not in the syntax just described, can be conceptually understood to stand for $> (B, 700)$, which *is* in the required syntax, and where $>$ is a relation.

But what does this notation mean for arithmetic operations such as "$>$"? The relation $>$ (conceptually) contains tuples of the form (x, y) for every possible pair of values x, y such that $x > y$. Thus, $(2, 1)$ and $(5, -33)$ are both tuples in $>$. Clearly, the (conceptual) relation $>$ is infinite. Other arithmetic operations (such as $>$, $=$, $+$ or $-$) are also treated conceptually as relations. For example, $A = B + C$ stands conceptually for $+(B, C, A)$, where the relation $+$ contains every tuple (x, y, z) such that $z = x + y$.

A **fact** is written in the form

$$p(v_1, v_2, \ldots, v_n)$$

and denotes that the tuple $(v_1, v_2, \ldots, v_n)$ is in relation p. A set of facts for a relation can also be written in the usual tabular notation. A set of facts for the relations in a database schema is equivalent to an instance of the database schema. **Rules** are built out of literals and have the form

$$p(t_1, t_2, \ldots, t_n) \coloneq L_1, L_2, \ldots, L_n$$

where each L_i is a (positive or negative) literal. The literal $p(t_1, t_2, \ldots, t_n)$ is referred to as the **head** of the rule, and the rest of the literals in the rule constitute the **body** of the rule.

A **Datalog program** consists of a set of rules; the order in which the rules are written has no significance. As mentioned earlier, there may be several rules defining a relation.

Figure 5.6 shows a Datalog program that defines the interest on each account in the Perryridge branch. The first rule of the program defines a view relation *interest*, whose attributes are the account number and the interest earned on the account. It uses the relation *account* and the view relation *interest-rate*. The last two rules of the program are rules that we saw earlier.

A view relation v_1 is said to **depend directly on** a view relation v_2 if v_2 is used in the expression defining v_1. In the above program, view relation *interest* depends directly on relations *interest-rate* and *account*. Relation *interest-rate* in turn depends directly on *account*.

A view relation v_1 is said to **depend indirectly on** view relation v_2 if there is a sequence of intermediate relations $i_1, i_2, \ldots, i_n$, for some n, such that v_1 depends directly on i_1, i_1 depends directly on i_2, and so on till i_{n-1} depends on i_n.

In the example in Figure 5.6, since we have a chain of dependencies from *interest* to *interest-rate* to *account*, relation *interest* also depends indirectly on *account*.

Finally, a view relation v_1 is said to **depend on** view relation v_2 if v_1 either depends directly or indirectly on v_2.

A view relation v is said to be **recursive** if it depends on itself. A view relation that is not recursive is said to be **nonrecursive**.

Consider the program in Figure 5.7. Here, the view relation *empl* depends on itself (becasue of the second rule), and is therefore recursive. In contrast, the program in Figure 5.6 is nonrecursive.

$$\begin{aligned}
&\textit{interest}(A, I) \coloneq \textit{account}(A, \text{“Perryridge”}, B),\\
&\qquad\qquad \textit{interest-rate}(A, R), I = B * R/100.\\
&\textit{interest-rate}(A, 5) \coloneq \textit{account}(A, N, B), B < 10000.\\
&\textit{interest-rate}(A, 6) \coloneq \textit{account}(A, N, B), B >= 10000.
\end{aligned}$$

Figure 5.6 Datalog program that defines interest on Perryridge accounts.

$$\begin{aligned} empl(X, Y) &:- manager(X, Y). \\ empl(X, Y) &:- manager(X, Z), empl(Z, Y). \end{aligned}$$

Figure 5.7 Recursive Datalog program.

5.2.3 Semantics of Nonrecursive Datalog

We consider the formal semantics of Datalog programs. For now, we consider only programs that are nonrecursive. The semantics of recursive programs is somewhat more complicated; it is discussed in Section 5.2.6. We define the semantics of a program by starting with the semantics of a single rule.

5.2.3.1 Semantics of a Rule

A **ground instantiation of a rule** is the result of replacing each variable in the rule by some constant. If a variable occurs multiple times in a rule, all occurrences of the variable must be replaced by the same constant. Ground instantiations are often simply called **instantiations**.

Our example rule defining *v1*, and an instantiation of the rule, are:

$$v1(A, B) :- account(A, \text{“Perryridge”}, B), B > 700$$
$$v1(\text{“A-217”}, 750) :- account(\text{“A-217”}, \text{“Perryridge”}, 750), 750 > 700$$

Here, variable A was replaced by “A-217,” and variable B by 750.

A rule usually has many possible instantiations. These instantiations correspond to the various ways of assigning values to each variable in the rule.

Suppose that we are given a rule R,

$$p(t_1, t_2, \ldots, t_n) :- L_1, L_2, \ldots, L_n$$

and a set of facts I for the relations used in the rule (I can also be thought of as a database instance). Consider any instantiation R' of rule R:

$$p(v_1, v_2, \ldots, v_n) :- l_1, l_2, \ldots, l_n$$

where each literal l_i is either of the form $q_i(v_{i,1}, v_{1,2}, \ldots, v_{i,n_i})$ or of the form **not** $q_i(v_{i,1}, v_{1,2}, \ldots, v_{i,n_i})$, and where each v_i and each $v_{i,j}$ is a constant.

We say that the body of rule instantiation R' is **satisfied** in I if

1. For each positive literal $q_i(v_{i,1}, \ldots, v_{i,n_i})$ in the body of R', the set of facts I contains the fact $q(v_{i,1}, \ldots, v_{i,n_i})$.

2. For each negative literal **not** $q_j(v_{j,1}, \ldots, v_{j,n_j})$ in the body of R', the set of facts I does not contain the fact $q_j(v_{j,1}, \ldots, v_{j,n_j})$.

account-number	*balance*
A-201	900
A-217	750

Figure 5.8 Result of *infer*(R, I).

We define the set of facts that can be **inferred** from a given set of facts I using rule R as

$$\begin{aligned} \textit{infer}(R, I) = \{p(t_1, \ldots, t_{n_i}) \mid & \text{ there is an instantiation } R' \text{ of } R, \\ & \text{where } p(t_1, \ldots, t_{n_i}) \text{ is the head of } R', \text{ and} \\ & \text{the body of } R' \text{ is satisfied in } I\}. \end{aligned}$$

Given a set of rules $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$, we define

$$\textit{infer}(\mathcal{R}, I) = \textit{infer}(R_1, I) \cup \textit{infer}(R_2, I) \cup \ldots \cup \textit{infer}(R_n, I)$$

Suppose that we are given a set of facts I containing the tuples for relation *account* in Figure 5.4. One possible instantiation of our running-example rule R is

$$\textit{v1}(\text{“A-217”}, 750) \ :\!-\ \textit{account}(\text{“A-217”}, \text{“Perryridge”}, 750), 750 > 700.$$

The fact *account*(“A-217”, “Perryridge”, 750) is in the set of facts I. Further, 750 is greater than 700, and hence conceptually $(750, 700)$ is in the relation “$>$”. Hence, the body of the rule instantiation is satisfied in I. There are other possible instantiations of R, and using them we find that *infer*(R, I) has exactly the set of facts for *v1* that appears in Figure 5.8.

5.2.3.2 Semantics of a Program

When a view relation is defined in terms of another view relation, the set of facts in the first view depends on the set of facts in the second one. We have assumed, in this section, that the definition is nonrecursive; that is, no view relation depends (directly or indirectly) on itself. Hence, we can layer the view relations in the following way, and can use the layering to define the semantics of the program:

- A relation is in layer 1 if all relations used in the bodies of rules defining it are stored in the database.
- A relation is in layer 2 if all relations used in the bodies of rules defining it either are stored in the database or are in layer 1.
- In general, a relation p is in layer $i + 1$ if (1) it is not in layers $1, 2, \ldots, i$, and (2) all relations used in the bodies of rules defining p either are stored in the database or are in layers $1, 2, \ldots, i$.

Consider the program in Figure 5.6. The layering of view relations in the program appears in Figure 5.9. The relation *account* is in the database. Relation *interest-rate* is

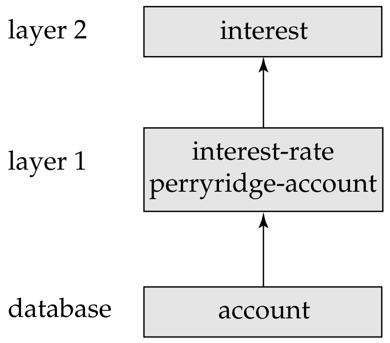


Figure 5.9 Layering of view relations.

in level 1, since all the relations used in the two rules defining it are in the database. Relation *perryridge-account* is similarly in layer 1. Finally, relation *interest* is in layer 2, since it is not in layer 1 and all the relations used in the rule defining it are in the database or in layers lower than 2.

We can now define the semantics of a Datalog program in terms of the layering of view relations. Let the layers in a given program be $1, 2, \ldots, n$. Let $\mathcal{R}_i$ denote the set of all rules defining view relations in layer i.

- We define I_0 to be the set of facts stored in the database, and define I_1 as

$$I_1 = I_0 \cup \mathit{infer}(\mathcal{R}_1, I_0)$$

- We proceed in a similar fashion, defining I_2 in terms of I_1 and $\mathcal{R}_2$, and so on, using the following definition:

$$I_{i+1} = I_i \cup \mathit{infer}(\mathcal{R}_{i+1}, I_i)$$

- Finally, the set of facts in the view relations defined by the program (also called the **semantics of the program**) is given by the set of facts I_n corresponding to the highest layer n.

For the program in Figure 5.6, I_0 is the set of facts in the database, and I_1 is the set of facts in the database along with all facts that we can infer from I_0 using the rules for relations *interest-rate* and *perryridge-account*. Finally, I_2 contains the facts in I_1 along with the facts for relation *interest* that we can infer from the facts in I_1 by the rule defining *interest*. The semantics of the program—that is, the set of those facts that are in each of the view relations—is defined as the set of facts I_2.

Recall that, in Section 3.5.3, we saw how to define the meaning of nonrecursive relational-algebra views by a technique known as view expansion. View expansion can be used with nonrecursive Datalog views as well; conversely, the layering technique described here can also be used with relational-algebra views.

5.2.4 Safety

It is possible to write rules that generate an infinite number of answers. Consider the rule

$$gt(X, Y) \text{ :– } X > Y$$

Since the relation defining $>$ is infinite, this rule would generate an infinite number of facts for the relation *gt*, which calculation would, correspondingly, take an infinite amount of time and space.

The use of negation can also cause similar problems. Consider the rule:

$$\textit{not-in-loan}(L, B, A) \text{ :– } \textbf{not } \textit{loan}(L, B, A)$$

The idea is that a tuple *(loan-number, branch-name, amount)* is in view relation *not-in-loan* if the tuple is not present in the *loan* relation. However, if the set of possible account numbers, branch-names, and balances is infinite, the relation *not-in-loan* would be infinite as well.

Finally, if we have a variable in the head that does not appear in the body, we may get an infinite number of facts where the variable is instantiated to different values.

So that these possibilities are avoided, Datalog rules are required to satisfy the following **safety** conditions:

1. Every variable that appears in the head of the rule also appears in a nonarithmetic positive literal in the body of the rule.
2. Every variable appearing in a negative literal in the body of the rule also appears in some positive literal in the body of the rule.

If all the rules in a nonrecursive Datalog program satisfy the preceding safety conditions, then all the view relations defined in the program can be shown to be finite, as long as all the database relations are finite. The conditions can be weakened somewhat to allow variables in the head to appear only in an arithmetic literal in the body in some cases. For example, in the rule

$$p(A) \text{ :– } q(B),\ A = B + 1$$

we can see that if relation *q* is finite, then so is *p*, according to the properties of addition, even though variable A appears in only an arithmetic literal.

5.2.5 Relational Operations in Datalog

Nonrecursive Datalog expressions without arithmetic operations are equivalent in expressive power to expressions using the basic operations in relational algebra ($\cup, -, \times, \sigma, \Pi$ and ρ). We shall not formally prove this assertion here. Rather, we shall show through examples how the various relational-algebra operations can be expressed in Datalog. In all cases, we define a view relation called *query* to illustrate the operations.

We have already seen how to do selection by using Datalog rules. We perform projections simply by using only the required attributes in the head of the rule. To project attribute *account-name* from account, we use

$$query(A) \text{ :- } account(A, N, B)$$

We can obtain the Cartesian product of two relations r_1 and r_2 in Datalog as follows:

$$query(X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_m) \text{ :- } r_1(X_1, X_2, \ldots, X_n), r_2(Y_1, Y_2, \ldots, Y_m)$$

where r_1 is of arity *n*, and r_2 is of arity *m*, and the $X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_m$ are all distinct variable names.

We form the union of two relations r_1 and r_2 (both of arity *n*) in this way:

$$query(X_1, X_2, \ldots, X_n) \text{ :- } r_1(X_1, X_2, \ldots, X_n)$$
$$query(X_1, X_2, \ldots, X_n) \text{ :- } r_2(X_1, X_2, \ldots, X_n)$$

We form the set difference of two relations r_1 and r_2 in this way:

$$query(X_1, X_2, \ldots, X_n) \text{ :- } r_1(X_1, X_2, \ldots, X_n), \textbf{ not } r_2(X_1, X_2, \ldots, X_n)$$

Finally, we note that with the positional notation used in Datalog, the renaming operator ρ is not needed. A relation can occur more than once in the rule body, but instead of renaming to give distinct names to the relation occurrences, we can use different variable names in the different occurrences.

It is possible to show that we can express any nonrecursive Datalog query without arithmetic by using the relational-algebra operations. We leave this demonstration as an exercise for you to carry out. You can thus establish the equivalence of the basic operations of relational algebra and nonrecursive Datalog without arithmetic operations.

Certain extensions to Datalog support the extended relational update operations of insertion, deletion, and update. The syntax for such operations varies from implementation to implementation. Some systems allow the use of $+$ or $-$ in rule heads to denote relational insertion and deletion. For example, we can move all accounts at the Perryridge branch to the Johnstown branch by executing

$$+\, account(A, \text{“Johnstown”}, B) \text{ :- } account(A, \text{“Perryridge”}, B)$$
$$-\, account(A, \text{“Perryridge”}, B) \text{ :- } account(A, \text{“Perryridge”}, B)$$

Some implementations of Datalog also support the aggregation operation of extended relational algebra. Again, there is no standard syntax for this operation.

5.2.6 Recursion in Datalog

Several database applications deal with structures that are similar to tree data structures. For example, consider employees in an organization. Some of the employees are managers. Each manager manages a set of people who report to him or her. But

```
procedure Datalog-Fixpoint
    $I$ = set of facts in the database
    repeat
        $Old\_I = I$
        $I = I \cup infer(\mathcal{R}, I)$
    until $I = Old\_I$
```

Figure 5.10 Datalog-Fixpoint procedure.

each of these people may in turn be managers, and they in turn may have other people who report to them. Thus employees may be organized in a structure similar to a tree.

Suppose that we have a relation schema

$$Manager\text{-}schema = (employee\text{-}name, manager\text{-}name)$$

Let *manager* be a relation on the preceding schema.

Suppose now that we want to find out which employees are supervised, directly or indirectly by a given manager—say, Jones. Thus, if the manager of Alon is Barinsky, and the manager of Barinsky is Estovar, and the manager of Estovar is Jones, then Alon, Barinsky, and Estovar are the employees controlled by Jones. People often write programs to manipulate tree data structures by recursion. Using the idea of recursion, we can define the set of employees controlled by Jones as follows. The people supervised by Jones are (1) people whose manager is Jones and (2) people whose manager is supervised by Jones. Note that case (2) is recursive.

We can encode the preceding recursive definition as a recursive Datalog view, called *empl-jones*:

$$\begin{aligned} &empl\text{-}jones(X) \text{ :– } manager(X, \text{"Jones"}) \\ &empl\text{-}jones(X) \text{ :– } manager(X, Y), empl\text{-}jones(Y) \end{aligned}$$

The first rule corresponds to case (1); the second rule corresponds to case (2). The view *empl-jones* depends on itself because of the second rule; hence, the preceding Datalog program is recursive. We *assume* that recursive Datalog programs contain no rules with negative literals. The reason will become clear later. The bibliographical

employee-name	*manager-name*
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Figure 5.11 The *manager* relation.

Iteration number	Tuples in *empl-jones*
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)

Figure 5.12 Employees of Jones in iterations of procedure Datalog-Fixpoint.

notes refer to papers that describe where negation can be used in recursive Datalog programs.

The view relations of a recursive program that contains a set of rules $\mathcal{R}$ are defined to contain exactly the set of facts I computed by the iterative procedure Datalog-Fixpoint in Figure 5.10. The recursion in the Datalog program has been turned into an iteration in the procedure. At the end of the procedure, *infer*$(\mathcal{R}, I) = I$, and I is called a **fixed point** of the program.

Consider the program defining *empl-jones*, with the relation *manager*, as in Figure 5.11. The set of facts computed for the view relation *empl-jones* in each iteration appears in Figure 5.12. In each iteration, the program computes one more level of employees under Jones and adds it to the set *empl-jones*. The procedure terminates when there is no change to the set *empl-jones*, which the system detects by finding $I = Old\_I$. Such a termination point must be reached, since the set of managers and employees is finite. On the given *manager* relation, the procedure Datalog-Fixpoint terminates after iteration 4, when it detects that no new facts have been inferred.

You should verify that, at the end of the iteration, the view relation *empl-jones* contains exactly those employees who work under Jones. To print out the names of the employees supervised by Jones defined by the view, you can use the query

$$?\ empl\text{-}jones(N)$$

To understand procedure Datalog-Fixpoint, we recall that a rule infers new facts from a given set of facts. Iteration starts with a set of facts I set to the facts in the database. These facts are all known to be true, but there may be other facts that are true as well.[1] Next, the set of rules $\mathcal{R}$ in the given Datalog program is used to infer what facts are true, given that facts in I are true. The inferred facts are added to I, and the rules are used again to make further inferences. This process is repeated until no new facts can be inferred.

For safe Datalog programs, we can show that there will be some point where no more new facts can be derived; that is, for some k, $I_{k+1} = I_k$. At this point, then, we have the final set of true facts. Further, given a Datalog program and a database, the fixed-point procedure infers all the facts that can be inferred to be true.

1. The word "fact" is used in a technical sense to note membership of a tuple in a relation. Thus, in the Datalog sense of "fact," a fact may be true (the tuple is indeed in the relation) or false (the tuple is not in the relation).

If a recursive program contains a rule with a negative literal, the following problem can arise. Recall that when we make an inference by using a ground instantiation of a rule, for each negative literal **not**q in the rule body we check that q is not present in the set of facts I. This test assumes that q cannot be inferred later. However, in the fixed-point iteration, the set of facts I grows in each iteration, and even if q is not present in I at one iteration, it may appear in I later. Thus, we may have made an inference in one iteration that can no longer be made at an earlier iteration, and the inference was incorrect. We require that a recursive program should not contain negative literals, in order to avoid such problems.

Instead of creating a view for the employees supervised by a specific manager Jones, we can create a more general view relation *empl* that contains every tuple (X, Y) such that X is directly or indirectly managed by Y, using the following program (also shown in Figure 5.7):

$$\begin{aligned} empl(X, Y) &\text{ :− } manager(X, Y) \\ empl(X, Y) &\text{ :− } manager(X, Z), empl(Z, Y) \end{aligned}$$

To find the direct and indirect subordinates of Jones, we simply use the query

$$?\ empl(X, \text{“Jones”})$$

which gives the same set of values for X as the view *empl-jones*. Most Datalog implementations have sophisticated query optimizers and evaluation engines that can run the preceding query at about the same speed they could evaluate the view *empl-jones*.

The view *empl* defined previously is called the **transitive closure** of the relation *manager*. If the relation *manager* were replaced by any other binary relation R, the preceding program would define the transitive closure of R.

5.2.7 The Power of Recursion

Datalog with recursion has more expressive power than Datalog without recursion. In other words, there are queries on the database that we can answer by using recursion, but cannot answer without using it. For example, we cannot express transitive closure in Datalog without using recursion (or for that matter, in SQL or QBE without recursion). Consider the transitive closure of the relation *manager*. Intuitively, a fixed number of joins can find only those employees that are some (other) fixed number of levels down from any manager (we will not attempt to prove this result here). Since any given nonrecursive query has a fixed number of joins, there is a limit on how many levels of employees the query can find. If the number of levels of employees in the *manager* relation is more than the limit of the query, the query will miss some levels of employees. Thus, a nonrecursive Datalog program cannot express transitive closure.

An alternative to recursion is to use an external mechanism, such as embedded SQL, to iterate on a nonrecursive query. The iteration in effect implements the fixed-point loop of Figure 5.10. In fact, that is how such queries are implemented on database systems that do not support recursion. However, writing such queries by iteration is more complicated than using recursion, and evaluation by recursion can be optimized to run faster than evaluation by iteration.

The expressive power provided by recursion must be used with care. It is relatively easy to write recursive programs that will generate an infinite number of facts, as this program illustrates:

$$\begin{array}{l} \textit{number}(0) \\ \textit{number}(A) \; :\!- \; \textit{number}(B),\ A = B + 1 \end{array}$$

The program generates $number(n)$ for all positive integers n, which is clearly infinite, and will not terminate. The second rule of the program does not satisfy the safety condition in Section 5.2.4. Programs that satisfy the safety condition will terminate, even if they are recursive, provided that all database relations are finite. For such programs, tuples in view relations can contain only constants from the database, and hence the view relations must be finite. The converse is not true; that is, there are programs that do not satisfy the safety conditions, but that do terminate.

5.2.8 Recursion in Other Languages

The SQL:1999 standard supports a limited form of recursion, using the **with recursive** clause. Suppose the relation *manager* has attributes *emp* and *mgr*. We can find every pair (X, Y) such that X is directly or indirectly managed by Y, using this SQL:1999 query:

```
with recursive empl(emp, mgr) as (
        select emp, mgr
        from manager
    union
        select emp, empl.mgr
        from manager, empl
        where manager.mgr = empl.emp
    )
select *
from empl
```

Recall that the **with** clause is used to define a temporary view whose definition is available only to the query where it is defined. The additional keyword **recursive** specifies that the view is recursive. The SQL definition of the view *empl* above is equivalent to the Datalog version we saw in Section 5.2.6.

The procedure Datalog-Fixpoint iteratively uses the function $infer(\mathcal{R}, I)$ to compute what facts are true, given a recursive Datalog program. Although we considered only the case of Datalog programs without negative literals, the procedure can also be used on views defined in other languages, such as SQL or relational algebra, provided that the views satisfy the conditions described next. Regardless of the language used to define a view V, the view can be thought of as being defined by an expression E_V that, given a set of facts I, returns a set of facts $E_V(I)$ for the view relation V. Given a set of view definitions $\mathcal{R}$ (in any language), we can define a function *infer*$(\mathcal{R}, I)$ that returns $I \cup \bigcup_{V \in \mathcal{R}} E_V(I)$. The preceding function has the same form as the *infer* function for Datalog.

A view V is said to be **monotonic** if, given any two sets of facts I_1 and I_2 such that $I_1 \subseteq I_2$, then $E_V(I_1) \subseteq E_V(I_2)$, where E_V is the expression used to define V. Similarly, the function *infer* is said to be monotonic if

$$I_1 \subseteq I_2 \Rightarrow \textit{infer}(\mathcal{R}, I_1) \subseteq infer(\mathcal{R}, I_2)$$

Thus, if *infer* is monotonic, given a set of facts I_0 that is a subset of the true facts, we can be sure that all facts in *infer*$(\mathcal{R}, I_0)$ are also true. Using the same reasoning as in Section 5.2.6, we can then show that procedure Datalog-Fixpoint is sound (that is, it computes only true facts), provided that the function *infer* is monotonic.

Relational-algebra expressions that use only the operators $\Pi, \sigma, \times, \bowtie, \cup, \cap$, or ρ are monotonic. Recursive views can be defined by using such expressions.

However, relational expressions that use the operator $-$ are not monotonic. For example, let $manager_1$ and $manager_2$ be relations with the same schema as the *manager* relation. Let

$$\begin{aligned} I_1 = \{\ & manager_1(\text{“Alon”, “Barinsky”}),\ manager_1(\text{“Barinsky”, “Estovar”}),\\ & manager_2(\text{“Alon”, “Barinsky”})\ \} \end{aligned}$$

and let

$$\begin{aligned} I_2 = \{\ & manager_1(\text{“Alon”, “Barinsky”}),\ manager_1(\text{“Barinsky”, “Estovar”}),\\ & manager_2(\text{“Alon”, “Barinsky”}),\ manager_2(\text{“Barinsky”, “Estovar”})\} \end{aligned}$$

Consider the expression $manager_1 - manager_2$. Now the result of the preceding expression on I_1 is (“Barinsky”, “Estovar”), whereas the result of the expression on I_2 is the empty relation. But $I_1 \subseteq I_2$; hence, the expression is not monotonic. Expressions using the grouping operation of extended relational algebra are also nonmonotonic.

The fixed-point technique does not work on recursive views defined with nonmonotonic expressions. However, there are instances where such views are useful, particularly for defining aggregates on “part−subpart” relationships. Such relationships define what subparts make up each part. Subparts themselves may have further subparts, and so on; hence, the relationships, like the manager relationship, have a natural recursive structure. An example of an aggregate query on such a structure would be to compute the total number of subparts of each part. Writing this query in Datalog or in SQL (without procedural extensions) would require the use of a recursive view on a nonmonotonic expression. The bibliographic notes provide references to research on defining such views.

It is possible to define some kinds of recursive queries without using views. For example, extended relational operations have been proposed to define transitive closure, and extensions to the SQL syntax to specify (generalized) transitive closure have been proposed. However, recursive view definitions provide more expressive power than do the other forms of recursive queries.

5.3 User Interfaces and Tools

Although many people interact with databases, few people use a query language to directly interact with a database system. Most people interact with a database system through one of the following means:

1. **Forms and graphical user interfaces** allow users to enter values that complete predefined queries. The system executes the queries and appropriately formats and displays the results to the user. Graphical user interfaces provide an easy-to-use way to interact with the database system.
2. **Report generators** permit predefined reports to be generated on the current database contents. Analysts or managers view such reports in order to make business decisions.
3. **Data analysis tools** permit users to interactively browse and analyze data.

It is worth noting that such interfaces use query languages to communicate with database systems.

In this section, we provide an overview of forms, graphical user interfaces, and report generators. Chapter 22 covers data analysis tools in more detail. Unfortunately, there are no standards for user interfaces, and each database system usually provides its own user interface. In this section, we describe the basic concepts, without going into the details of any particular user interface product.

5.3.1 Forms and Graphical User Interfaces

Forms interfaces are widely used to enter data into databases, and extract information from databases, via predefined queries. For example, World Wide Web search engines provide forms that are used to enter key words. Hitting a "submit" button causes the search engine to execute a query using the entered key words and display the result to the user.

As a more database-oriented example, you may connect to a university registration system, where you are asked to fill in your roll number and password into a form. The system uses this information to verify your identity, as well as to extract information, such as your name and the courses you have registered for, from the database and display it. There may be further links on the Web page that let you search for courses and find further information about courses such as the syllabus and the instructor.

Web browsers supporting HTML constitute the most widely used forms and graphical user interface today. Most database system vendors also provide proprietary forms interfaces that offer facilities beyond those present in HTML forms.

Programmers can create forms and graphical user interfaces by using HTML or programming languages such as C or Java. Most database system vendors also provide tools that simplify the creation of graphical user interfaces and forms. These tools allow application developers to create forms in an easy declarative fashion, using form-editor programs. Users can define the type, size, and format of each field in a form by using the form editor. System actions can be associated with user actions, such as filling in a field, hitting a function key on the keyboard, or submitting a form. For instance, the execution of a query to fill in name and address fields may be associated with filling in a roll number field, and execution of an update statement may be associated with submitting a form.

Simple error checks can be performed by defining constraints on the fields in the form.[2] For example, a constraint on the course number field may check that the course number typed in by the user corresponds to an actual course. Although such constraints can be checked when the transaction is executed, detecting errors early helps the user to correct errors quickly. Menus that indicate the valid values that can be entered in a field can help eliminate the possibility of many types of errors. System developers find that the ability to control such features declaratively with the help of a user interface development tool, instead of creating a form directly by using a scripting or programming language, makes their job much easier.

5.3.2 Report Generators

Report generators are tools to generate human-readable summary reports from a database. They integrate querying the database with the creation of formatted text and summary charts (such as bar or pie charts). For example, a report may show the total sales in each of the past two months for each sales region.

The application developer can specify report formats by using the formatting facilities of the report generator. Variables can be used to store parameters such as the month and the year and to define fields in the report. Tables, graphs, bar charts, or other graphics can be defined via queries on the database. The query definitions can make use of the parameter values stored in the variables.

Once we have defined a report structure on a report-generator facility, we can store it, and can execute it at any time to generate a report. Report-generator systems provide a variety of facilities for structuring tabular output, such as defining table and column headers, displaying subtotals for each group in a table, automatically splitting long tables into multiple pages, and displaying subtotals at the end of each page.

Figure 5.13 is an example of a formatted report. The data in the report are generated by aggregation on information about orders.

The Microsoft Office suite provides a convenient way of embedding formatted query results from a database, such as MS Access, into a document created with a text editor, such as MS Word. The query results can be formatted in a tabular fashion or graphically (as charts) by the report generator facility of MS Access. A feature called OLE (Object Linking and Embedding) links the resulting structure into a text document.

The collections of application-development tools provided by database systems, such as forms packages and report generator, used to be referred to as *fourth-generation languages* (4GLs). The name emphasizes that these tools offer a programming paradigm that is different from the imperative programming paradigm offered by third-

2. These are called “form triggers” in Oracle, but in this book we use the term “trigger” in a different sense, which we cover in Chapter 6.

Acme Supply Company Inc.
Quarterly Sales Report

Period: Jan. 1 to March 31, 2001

Region	**Category**	**Sales**	**Subtotal**
North	Computer Hardware	1,000,000	
	Computer Software	500,000	
	All categories		1,500,000
South	Computer Hardware	200,000	
	Computer Software	400,000	
	All categories		600,000
		Total Sales	2,100,000

Figure 5.13 A formatted report.

generation programming languages, such as Pascal and C. However, this term is less relevant today, since forms and report generators are typically created with graphical tools, rather than with programming languages.

5.4 Summary

- We have considered two query languages: QBE, and Datalog.
- QBE is based on a visual paradigm: The queries look much like tables.
- QBE and its variants have become popular with nonexpert database users because of the intuitive simplicity of the visual paradigm. The widely used Microsoft Access database system supports a graphical version of QBE, called GQBE.
- Datalog is derived from Prolog, but unlike Prolog, it has a declarative semantics, making simple queries easier to write and query evaluation easier to optimize.
- Defining views is particularly easy in Datalog, and the recursive views that Datalog supports makes it possible to write queries, such as transitive-closure queries, that cannot be written without recursion or iteration. However, no accepted standards exist for important features, such as grouping and aggregation, in Datalog. Datalog remains mainly a research language.
- Most users interact with databases via forms and graphical user interfaces, and there are numerous tools to simplify the construction of such interfaces. Report generators are tools that help create human-readable reports from the contents of the database.

Review Terms

- Query-by-Example (QBE)
- Two-dimensional syntax
- Skeleton tables
- Example rows
- Condition box
- Result relation
- Microsoft Access
- Graphical Query-By-Example (GQBE)
- Design grid
- Datalog
- Rules
- Uses
- Defines
- Positive literal
- Negative literal
- Fact
- Rule
  - ☐ Head
  - ☐ Body
- Datalog program
- Depend on
  - ☐ Directly
  - ☐ Indirectly
- Recursive view
- Nonrecursive view
- Instantiation
  - ☐ Ground instantiation
  - ☐ Satisfied
- Infer
- Semantics
  - ☐ Of a rule
  - ☐ Of a program
- Safety
- Fixed point
- Transitive closure
- Monotonic view definition
- Forms
- Graphical user interfaces
- Report generators

Exercises

5.1 Consider the insurance database of Figure 5.14, where the primary keys are underlined. Construct the following QBE queries for this relational-database.

a. Find the total number of people who owned cars that were involved in accidents in 1989.
b. Find the number of accidents in which the cars belonging to "John Smith" were involved.
c. Add a new accident to the database; assume any values for required attributes.
d. Delete the Mazda belonging to "John Smith."
e. Update the damage amount for the car with license number "AABB2000" in the accident with report number "AR2197" to $3000.

5.2 Consider the employee database of Figure 5.15. Give expressions in QBE, and Datalog for each of the following queries:

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

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 5.14 Insurance database.

c. Find the names, street addresses, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum.
d. Find all employees who live in the same city as the company for which they work is located.
e. Find all employees who live in the same city and on the same street as their managers.
f. Find all employees in the database who do not work for First Bank Corporation.
g. Find all employees who earn more than every employee of Small Bank Corporation.
h. Assume that the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

5.3 Consider the relational database of Figure 5.15. where the primary keys are underlined. Give expressions in QBE for each of the following queries:

a. Find all employees who earn more than the average salary of all employees of their company.
b. Find the company that has the most employees.
c. Find the company that has the smallest payroll.
d. Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

5.4 Consider the relational database of Figure 5.15. Give expressions in QBE for each of the following queries:

a. Modify the database so that Jones now lives in Newtown.
b. Give all employees of First Bank Corporation a 10 percent raise.
c. Give all managers in the database a 10 percent raise.
d. Give all managers in the database a 10 percent raise, unless the salary would be greater than $100,000. In such cases, give only a 3 percent raise.

employee (*person-name*, *street*, *city*)
works (*person-name*, *company-name*, *salary*)
company (*company-name*, *city*)
manages (*person-name*, *manager-name*)

Figure 5.15 Employee database.

e. Delete all tuples in the *works* relation for employees of Small Bank Corporation.

5.5 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 expressions in QBE, and Datalog equivalent to each of the following queries:

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

5.6 Let $R = (A, B, C)$, and let r_1 and r_2 both be relations on schema R. Give expressions in QBE, and Datalog 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. $\Pi_{AB}(r_1)\ \bowtie\ \Pi_{BC}(r_2)$

5.7 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Write expressions in QBE and Datalog for each of the following queries:

a. $\{< a > \mid \exists\, b\, (< a, b > \in\ r\ \wedge\ b\ =\ 17)\}$
b. $\{< a, b, c > \ \mid\ < a, b > \in\ r \wedge\ < a, c > \in\ s\}$
c. $\{< a > \mid \exists\, c\, (< a, c > \in\ s\ \wedge\ \exists\, b_1, b_2\, (< a, b_1 > \in\ r \wedge\ < c,\, b_2 > \in\ r\ \wedge\ b_1\ >\ b_2))\}$

5.8 Consider the relational database of Figure 5.15. Write a Datalog program for each of the following queries:

a. Find all employees who work (directly or indirectly) under the manager "Jones".
b. Find all cities of residence of all employees who work (directly or indirectly) under the manager "Jones".
c. Find all pairs of employees who have a (direct or indirect) manager in common.
d. Find all pairs of employees who have a (direct or indirect) manager in common, and are at the same number of levels of supervision below the common manager.

5.9 Write an extended relational-algebra view equivalent to the Datalog rule

$$p(A, C, D) :\!-\ q1(A, B),\ q2(B, C),\ q3(4, B),\ D = B + 1\ .$$

5.10 Describe how an arbitrary Datalog rule can be expressed as an extended relational algebra view.

Bibliographical Notes

The experimental version of Query-by-Example is described in Zloof [1977]; the commercial version is described in IBM [1978]. Numerous database systems—in particular, database systems that run on personal computers—implement QBE or variants. Examples are Microsoft Access and Borland Paradox.

Implementations of Datalog include LDL system (described in Tsur and Zaniolo [1986] and Naqvi and Tsur [1988]), Nail! (described in Derr et al. [1993]), and Coral (described in Ramakrishnan et al. [1992b] and Ramakrishnan et al. [1993]). Early discussions concerning logic databases were presented in Gallaire and Minker [1978] and Gallaire et al. [1984]. Ullman [1988] and Ullman [1989] provide extensive textbook discussions of logic query languages and implementation techniques. Ramakrishnan and Ullman [1995] provides a more recent survey on deductive databases.

Datalog programs that have both recursion and negation can be assigned a simple semantics if the negation is "stratified"—that is, if there is no recursion through negation. Chandra and Harel [1982] and Apt and Pugin [1987] discuss stratified negation. An important extension, called the *modular-stratification semantics*, which handles a class of recursive programs with negative literals, is discussed in Ross [1990]; an evaluation technique for such programs is described by Ramakrishnan et al. [1992a].

Tools

The Microsoft Access QBE is probably the most widely used implementation of QBE. IBM DB2 QMF and Borland Paradox also support QBE.

The Coral system from the University of Wisconsin−Madison is a widely used implementation of Datalog (see (http://www.cs.wisc.edu/coral). The XSB system from the State University of New York (SUNY) Stony Brook (http://xsb.sourceforge.net) is a widely used Prolog implementation that supports database querying; recall that Datalog is a nonprocedural subset of Prolog.

C H A P T E R 6

Integrity and Security

Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.

We have already seen two forms of integrity constraints for the E-R model in Chapter 2:

- **Key declarations**—the stipulation that certain attributes form a candidate key for a given entity set.
- **Form of a relationship**—many to many, one to many, one to one.

In general, an integrity constraint can be an arbitrary predicate pertaining to the database. However, arbitrary predicates may be costly to test. Thus, we concentrate on integrity constraints that can be tested with minimal overhead. We study some such forms of integrity constraints in Sections 6.1 and 6.2, and cover a more complex form in Section 6.3. In Chapter 7 we study another form of integrity constraint, called "functional dependency," which is primarily used in the process of schema design.

In Section 6.4 we study *triggers*, which are statements that are executed automatically by the system as a side effect of a modification to the database. Triggers are used to ensure some types of integrity.

In addition to protecting against accidental introduction of inconsistency, the data stored in the database need to be protected from unauthorized access and malicious destruction or alteration. In Sections 6.5 through 6.7, we examine ways in which data may be misused or intentionally made inconsistent, and present security mechanisms to guard against such occurrences.

6.1 Domain Constraints

We have seen that a domain of possible values must be associated with every attribute. In Chapter 4, we saw a number of standard domain types, such as integer types, character types, and date/time types defined in SQL. Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database.

It is possible for several attributes to have the same domain. For example, the attributes *customer-name* and *employee-name* might have the same domain: the set of all person names. However, the domains of *balance* and *branch-name* certainly ought to be distinct. It is perhaps less clear whether *customer-name* and *branch-name* should have the same domain. At the implementation level, both customer names and branch names are character strings. However, we would normally not consider the query "Find all customers who have the same name as a branch" to be a meaningful query. Thus, if we view the database at the conceptual, rather than the physical, level, *customer-name* and *branch-name* should have distinct domains.

From the above discussion, we can see that a proper definition of domain constraints not only allows us to test values inserted in the database, but also permits us to test queries to ensure that the comparisons made make sense. The principle behind attribute domains is similar to that behind typing of variables in programming languages. Strongly typed programming languages allow the compiler to check the program in greater detail.

The **create domain** clause can be used to define new domains. For example, the statements:

create domain *Dollars* **numeric**(12,2)
create domain *Pounds* **numeric**(12,2)

define the domains *Dollars* and *Pounds* to be decimal numbers with a total of 12 digits, two of which are placed after the decimal point. An attempt to assign a value of type *Dollars* to a variable of type *Pounds* would result in a syntax error, although both are of the same numeric type. Such an assignment is likely to be due to a programmer error, where the programmer forgot about the differences in currency. Declaring different domains for different currencies helps catch such errors.

Values of one domain can be *cast* (that is, converted) to another domain. If the attribute *A* or relation *r* is of type *Dollars*, we can convert it to *Pounds* by writing

cast *r.A* **as** *Pounds*

In a real application we would of course multiply *r.A* by a currency conversion factor before casting it to pounds. SQL also provides **drop domain** and **alter domain** clauses to drop or modify domains that have been created earlier.

The **check** clause in SQL permits domains to be restricted in powerful ways that most programming language type systems do not permit. Specifically, the **check** clause permits the schema designer to specify a predicate that must be satisfied by any value assigned to a variable whose type is the domain. For instance, a **check** clause can ensure that an hourly wage domain allows only values greater than a specified value (such as the minimum wage):

```
create domain HourlyWage numeric(5,2)
        constraint wage-value-test check(value >= 4.00)
```

The domain *HourlyWage* has a constraint that ensures that the hourly wage is greater than 4.00. The clause **constraint** *wage-value-test* is optional, and is used to give the name *wage-value-test* to the constraint. The name is used to indicate which constraint an update violated.

The **check** clause can also be used to restrict a domain to not contain any null values:

```
create domain AccountNumber char(10)
        constraint account-number-null-test check(value not null )
```

As another example, the domain can be restricted to contain only a specified set of values by using the **in** clause:

```
create domain AccountType char(10)
        constraint account-type-test
                check(value in ('Checking', 'Saving'))
```

The preceding **check** conditions can be tested quite easily, when a tuple is inserted or modified. However, in general, the **check** conditions can be more complex (and harder to check), since subqueries that refer to other relations are permitted in the **check** condition. For example, this constraint could be specified on the relation *deposit*:

```
check (branch-name in (select branch-name from branch))
```

The **check** condition verifies that the *branch-name* in each tuple in the *deposit* relation is actually the name of a branch in the *branch* relation. Thus, the condition has to be checked not only when a tuple is inserted or modified in *deposit*, but also when the relation *branch* changes (in this case, when a tuple is deleted or modified in relation *branch*).

The preceding constraint is actually an example of a class of constraints called *referential-integrity* constraints. We discuss such constraints, along with a simpler way of specifying them in SQL, in Section 6.2.

Complex **check** conditions can be useful when we want to ensure integrity of data, but we should use them with care, since they may be costly to test.

6.2 Referential Integrity

Often, we wish to ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called **referential integrity**.

6.2.1 Basic Concepts

Consider a pair of relations $r(R)$ and s(S), and the natural join $r \bowtie s$. There may be a tuple t_r in *r* that does not join with any tuple in *s*. That is, there is no t_s in *s* such that $t_r[R \cap S] = t_s[R \cap S]$. Such tuples are called *dangling* tuples. Depending on the entity set or relationship set being modeled, dangling tuples may or may not be acceptable. In Section 3.3.3, we considered a modified form of join—the outer join—to operate on relations containing dangling tuples. Here, our concern is not with queries, but rather with when we should permit dangling tuples to exist in the database.

Suppose there is a tuple t_1 in the *account* relation with $t_1[branch\text{-}name]$ = "Lunartown," but there is no tuple in the *branch* relation for the Lunartown branch. This situation would be undesirable. We expect the *branch* relation to list all bank branches. Therefore, tuple t_1 would refer to an account at a branch that does not exist. Clearly, we would like to have an integrity constraint that prohibits dangling tuples of this sort.

Not all instances of dangling tuples are undesirable, however. Assume that there is a tuple t_2 in the *branch* relation with $t_2[branch\text{-}name]$ = "Mokan," but there is no tuple in the *account* relation for the Mokan branch. In this case, a branch exists that has no accounts. Although this situation is not common, it may arise when a branch is opened or is about to close. Thus, we do not want to prohibit this situation.

The distinction between these two examples arises from two facts:

- The attribute *branch-name* in *Account-schema* is a foreign key referencing the primary key of *Branch-schema*.
- The attribute *branch-name* in *Branch-schema* is not a foreign key.

(Recall from Section 3.1.3 that a foreign key is a set of attributes in a relation schema that forms a primary key for another schema.)

In the Lunartown example, tuple t_1 in *account* has a value on the foreign key *branch-name* that does not appear in *branch*. In the Mokan-branch example, tuple t_2 in *branch* has a value on *branch-name* that does not appear in *account*, but *branch-name* is not a foreign key. Thus, the distinction between our two examples of dangling tuples is the presence of a foreign key.

Let $r_1(R_1)$ and $r_2(R_2)$ be relations with primary keys K_1 and K_2, respectively. We say that a subset α of R_2 is a **foreign key** referencing K_1 in relation r_1 if it is required that, for every t_2 in r_2, there must be a tuple t_1 in r_1 such that $t_1[K_1] = t_2[\alpha]$. Requirements of this form are called **referential integrity constraints**, or **subset dependencies**. The latter term arises because the preceding referential-integrity constraint can be written as $\Pi_\alpha (r_2) \subseteq \Pi_{K_1} (r_1)$. Note that, for a referential-integrity constraint to make sense, either α must be equal to K_1, or α and K_1 must be compatible sets of attributes.

6.2.2 Referential Integrity and the E-R Model

Referential-integrity constraints arise frequently. If we derive our relational-database schema by constructing tables from E-R diagrams, as we did in Chapter 2, then every

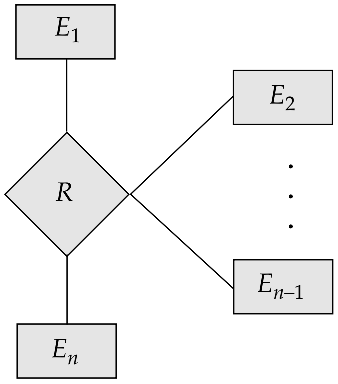


Figure 6.1 An n-ary relationship set.

relation arising from a relationship set has referential-integrity constraints. Figure 6.1 shows an n-ary relationship set R, relating entity sets $E_1,\ E_2, \ldots, E_n$. Let K_i denote the primary key of E_i. The attributes of the relation schema for relationship set R include $K_1 \ \cup \ K_2 \ \cup \ \cdots \ \cup \ K_n$. The following referential integrity constraints are then present: For each i, K_i in the schema for R is a foreign key referencing K_i in the relation schema generated from entity set E_i

Another source of referential-integrity constraints is weak entity sets. Recall from Chapter 2 that the relation schema for a weak entity set must include the primary key of the entity set on which the weak entity set depends. Thus, the relation schema for each weak entity set includes a foreign key that leads to a referential-integrity constraint.

6.2.3 Database Modification

Database modifications can cause violations of referential integrity. We list here the test that we must make for each type of database modification to preserve the following referential-integrity constraint:

$$\Pi_\alpha \ (r_2) \ \subseteq \ \Pi_K \ (r_1)$$

- **Insert**. If a tuple t_2 is inserted into r_2, the system must ensure that there is a tuple t_1 in r_1 such that $t_1[K] \ = \ t_2[\alpha]$. That is,

$$t_2[\alpha] \ \in \ \Pi_K \ (r_1)$$

- **Delete**. If a tuple t_1 is deleted from r_1, the system must compute the set of tuples in r_2 that reference t_1:

$$\sigma_{\alpha \ = \ t_1[K]} \ (r_2)$$

  If this set is not empty, either the delete command is rejected as an error, or the tuples that reference t_1 must themselves be deleted. The latter solution may lead to cascading deletions, since tuples may reference tuples that reference t_1, and so on.

- **Update**. We must consider two cases for update: updates to the referencing relation (r_2), and updates to the referenced relation (r_1).
  - ☐ If a tuple t_2 is updated in relation r_2, and the update modifies values for the foreign key α, then a test similar to the insert case is made. Let $t_2{}'$ denote the new value of tuple t_2. The system must ensure that

$$t_2{}'[\alpha] \in \Pi_K \ (r_1)$$

  - ☐ If a tuple t_1 is updated in r_1, and the update modifies values for the primary key (K), then a test similar to the delete case is made. The system must compute

$$\sigma_{\alpha \, = \, t_1[K]} \ (r_2)$$

  using the old value of t_1 (the value before the update is applied). If this set is not empty, the update is rejected as an error, or the update is cascaded in a manner similar to delete.

6.2.4 Referential Integrity in SQL

Foreign keys can be specified as part of the SQL **create table** statement by using the **foreign key** clause. We illustrate foreign-key declarations by using the SQL DDL definition of part of our bank database, shown in Figure 6.2.

By default, a foreign key references the primary key attributes of the referenced table. SQL also supports a version of the **references** clause where a list of attributes of the referenced relation can be specified explicitly. The specified list of attributes must be declared as a candidate key of the referenced relation.

We can use the following short form as part of an attribute definition to declare that the attribute forms a foreign key:

branch-name **char**(15) **references** *branch*

When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation. However, a **foreign key** clause can specify that if a delete or update action on the referenced relation violates the constraint, then, instead of rejecting the action, the system must take steps to change the tuple in the referencing relation to restore the constraint. Consider this definition of an integrity constraint on the relation *account*:

```
create table account
    ( . . .
    foreign key (branch-name) references branch
                              on delete cascade
                              on update cascade,
    . . . )
```

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *branch* results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "cascades" to the

```
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           integer,
    primary key (branch-name),
    check (assets >= 0))

create table account
   (account-number   char(10),
    branch-name      char(15),
    balance          integer,
    primary key (account-number),
    foreign key (branch-name) references branch,
    check (balance >= 0))

create table depositor
   (customer-name    char(20),
    account-number   char(10),
    primary key (customer-name, account-number),
    foreign key (customer-name) references customer,
    foreign key (account-number) references account)
```

Figure 6.2 SQL data definition for part of the bank database.

account relation, deleting the tuple that refers to the branch that was deleted. Similarly, the system does not reject an update to a field referenced by the constraint if it violates the constraint; instead, the system updates the field *branch-name* in the referencing tuples in *account* to the new value as well. SQL also allows the **foreign key** clause to specify actions other than **cascade**, if the constraint is violated: The referencing field (here, *branch-name*) can be set to null (by using **set null** in place of **cascade**), or to the default value for the domain (by using **set default**).

If there is a chain of foreign-key dependencies across multiple relations, a deletion or update at one end of the chain can propagate across the entire chain. An interesting case where the **foreign key** constraint on a relation references the same relation appears in Exercise 6.4. If a cascading update or delete causes a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction. As a result, all the changes caused by the transaction and its cascading actions are undone.

Null values complicate the semantics of referential integrity constraints in SQL. Attributes of foreign keys are allowed to be null, provided that they have not otherwise been declared to be non-null. If all the columns of a foreign key are non-null in a given tuple, the usual definition of foreign-key constraints is used for that tuple. If any of the foreign-key columns is null, the tuple is defined automatically to satisfy the constraint.

This definition may not always be the right choice, so SQL also provides constructs that allow you to change the behavior with null values; we do not discuss the constructs here. To avoid such complexity, it is best to ensure that all columns of a **foreign key** specification are declared to be non-null.

Transactions may consist of several steps, and integrity constraints may be violated temporarily after one step, but a later step may remove the violation. For instance, suppose we have a relation *marriedperson* with primary key *name*, and an attribute *spouse*, and suppose that *spouse* is a foreign key on *marriedperson*. That is, the constraint says that the *spouse* attribute must contain a name that is present in the *person* table. Suppose we wish to note the fact that John and Mary are married to each other by inserting two tuples, one for John and one for Mary, in the above relation. The insertion of the first tuple would violate the foreign key constraint, regardless of which of the two tuples is inserted first. After the second tuple is inserted the foreign key constraint would hold again.

To handle such situations, integrity constraints are checked at the end of a transaction, and not at intermediate steps.[1]

6.3 Assertions

An **assertion** is a predicate expressing a condition that we wish the database always to satisfy. Domain constraints and referential-integrity constraints are special forms of assertions. We have paid substantial attention to these forms of assertion because they are easily tested and apply to a wide range of database applications. However, there are many constraints that we cannot express by using only these special forms. Two examples of such constraints are:

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.
- Every loan has at least one customer who maintains an account with a minimum balance of $1000.00.

An assertion in SQL takes the form

create assertion <assertion-name> **check** <predicate>

Here is how the two examples of constraints can be written. Since SQL does not provide a "for all X, $P(X)$" construct (where P is a predicate), we are forced to im-

1. We can work around the problem in the above example in another way, if the *spouse* attribute can be set to null: We set the spouse attributes to null when inserting the tuples for John and Mary, and we update them later. However, this technique is rather messy, and does not work if the attributes cannot be set to null.

plement the construct by the equivalent "not exists X such that not $P(X)$" construct, which can be written in SQL. We write

```
create assertion sum-constraint check
    (not exists (select * from branch
        where (select sum(amount) from loan
                where loan.branch-name = branch.branch-name)
              >= (select sum(balance) from account
                where account.branch-name = branch.branch-name)))

create assertion balance-constraint check
    (not exists (select * from loan
        where not exists ( select *
            from borrower, depositor, account
            where loan.loan-number = borrower.loan-number
                and borrower.customer-name = depositor.customer-name
                and depositor.account-number = account.account-number
                and account.balance >= 1000)))
```

When an assertion is created, the system tests it for validity. If the assertion is valid, then any future modification to the database is allowed only if it does not cause that assertion to be violated. This testing may introduce a significant amount of overhead if complex assertions have been made. Hence, assertions should be used with great care. The high overhead of testing and maintaining assertions has led some system developers to omit support for general assertions, or to provide specialized forms of assertions that are easier to test.

6.4 Triggers

A **trigger** is a statement that the system executes automatically as a side effect of a modification to the database. To design a trigger mechanism, we must meet two requirements:

1. Specify when a trigger is to be executed. This is broken up into an *event* that causes the trigger to be checked and a *condition* that must be satisfied for trigger execution to proceed.
2. Specify the *actions* to be taken when the trigger executes.

The above model of triggers is referred to as the **event-condition-action model** for triggers.

The database stores triggers just as if they were regular data, so that they are persistent and are accessible to all database operations. Once we enter a trigger into the database, the database system takes on the responsibility of executing it whenever the specified event occurs and the corresponding condition is satisfied.

6.4.1 Need for Triggers

Triggers are useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met. As an illustration, suppose that, instead of allowing negative account balances, the bank deals with overdrafts by setting the account balance to zero, and creating a loan in the amount of the overdraft. The bank gives this loan a loan number identical to the account number of the overdrawn account. For this example, the condition for executing the trigger is an update to the *account* relation that results in a negative *balance* value. Suppose that Jones' withdrawal of some money from an account made the account balance negative. Let *t* denote the account tuple with a negative *balance* value. The actions to be taken are:

- Insert a new tuple *s* in the *loan* relation with

  $$\begin{aligned} &s[\textit{loan-number}] = t[\textit{account-number}] \\ &s[\textit{branch-name}] = t[\textit{branch-name}] \\ &s[\textit{amount}] = -t[\textit{balance}] \end{aligned}$$

  (Note that, since *t*[*balance*] is negative, we negate *t*[*balance*] to get the loan amount—a positive number.)

- Insert a new tuple *u* in the *borrower* relation with

  $$\begin{aligned} &u[\textit{customer-name}] = \text{“Jones”} \\ &u[\textit{loan-number}] = t[\textit{account-number}] \end{aligned}$$

- Set *t*[*balance*] to 0.

As another example of the use of triggers, suppose a warehouse wishes to maintain a minimum inventory of each item; when the inventory level of an item falls below the minimum level, an order should be placed automatically. This is how the business rule can be implemented by triggers: On an update of the inventory level of an item, the trigger should compare the level with the minimum inventory level for the item, and if the level is at or below the minimum, a new order is added to an *orders* relation.

Note that trigger systems cannot usually perform updates outside the database, and hence in the inventory replenishment example, we cannot use a trigger to directly place an order in the external world. Instead, we add an order to the *orders* relation as in the inventory example. We must create a separate permanently running system process that periodically scans the *orders* relation and places orders. This system process would also note which tuples in the *orders* relation have been processed and when each order was placed. The process would also track deliveries of orders, and alert managers in case of exceptional conditions such as delays in deliveries.

6.4.2 Triggers in SQL

SQL-based database systems use triggers widely, although before SQL:1999 they were not part of the SQL standard. Unfortunately, each database system implemented its

```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
    insert into borrower
              (select customer-name, account-number
               from depositor
               where nrow.account-number = depositor.account-number);
     insert into loan values
              (nrow.account-number, nrow.branch-name, − nrow.balance);
    update account set balance = 0
              where account.account-number = nrow.account-number
end
```

Figure 6.3 Example of SQL:1999 syntax for triggers.

own syntax for triggers, leading to incompatibilities. We outline in Figure 6.3 the SQL:1999 syntax for triggers (which is similar to the syntax in the IBM DB2 and Oracle database systems).

This trigger definition specifies that the trigger is initiated *after* any update of the relation *account* is executed. An SQL update statement could update multiple tuples of the relation, and the **for each row** clause in the trigger code would then explicitly iterate over each updated row. The **referencing new row as** clause creates a variable *nrow* (called a **transition variable**), which stores the value of an updated row after the update.

The **when** statement specifies a condition, namely *nrow.balance* < 0. The system executes the rest of the trigger body only for tuples that satisfy the condition. The **begin atomic** . . . **end** clause serves to collect multiple SQL statements into a single compound statement. The two **insert** statements with the **begin** . . . **end** structure carry out the specific tasks of creating new tuples in the *borrower* and *loan* relations to represent the new loan. The **update** statement serves to set the account balance back to 0 from its earlier negative value.

The triggering event and actions can take many forms:

- The triggering *event* can be **insert** or **delete**, instead of **update**.

  For example, the action on **delete** of an account could be to check if the holders of the account have any remaining accounts, and if they do not, to delete them from the *depositor* relation. You can define this trigger as an exercise (Exercise 6.7).

  As another example, if a new *depositor* is inserted, the triggered action could be to send a welcome letter to the depositor. Obviously a trigger cannot directly cause such an action outside the database, but could instead add a tuple to a relation storing addresses to which welcome letters need to be sent. A separate process would go over this table, and print out letters to be sent.

- For updates, the trigger can specify columns whose update causes the trigger to execute. For instance if the first line of the overdraft trigger were replaced by

  create trigger *overdraft-trigger* **after update of** *balance* **on** *account*

  then the trigger would be executed only on updates to *balance*; updates to other attributes would not cause it to be executed.

- The **referencing old row as** clause can be used to create a variable storing the old value of an updated or deleted row. The **referencing new row as** clause can be used with inserts in addition to updates.

- Triggers can be activated **before** the event (insert/delete/update) instead of **after** the event.

  Such triggers can serve as extra constraints that can prevent invalid updates. For instance, if we wish not to permit overdrafts, we can create a **before** trigger that rolls back the transaction if the new balance is negative.

  As another example, suppose the value in a phone number field of an inserted tuple is blank, which indicates absence of a phone number. We can define a trigger that replaces the value by the **null** value. The **set** statement can be used to carry out such modifications.

  create trigger *setnull-trigger* **before update on** *r*
  referencing new row as *nrow*
  for each row
  when *nrow.phone-number* = ' '
  set *nrow.phone-number* = **null**

- Instead of carrying out an action for each affected row, we can carry out a single action for the entire SQL statement that caused the insert/delete/update. To do so, we use the **for each statement** clause instead of the **for each row** clause.

  The clauses **referencing old table as** or **referencing new table as** can then be used to refer to temporary tables (called *transition tables*) containing all the affected rows. Transition tables cannot be used with **before** triggers, but can be used with **after** triggers, regardless of whether they are statement triggers or row triggers.

  A single SQL statement can then be used to carry out multiple actions on the basis of the transition tables.

Returning to our warehouse inventory example, suppose we have the following relations:

- *inventory(item, level)*, which notes the current amount (number/weight/volume) of the item in the warehouse

```
create trigger reorder-trigger after update of amount on inventory
referencing old row as orow, new row as nrow
for each row
when nrow.level <= (select level
                    from minlevel
                    where minlevel.item = orow.item)
and orow.level > (select level
                    from minlevel
                    where minlevel.item = orow.item)
begin
    insert into orders
        (select item, amount
         from reorder
         where reorder.item = orow.item)
end
```

Figure 6.4 Example of trigger for reordering an item.

- *minlevel(item, level)*, which notes the minimum amount of the item to be maintained
- *reorder(item, amount)*, which notes the amount of the item to be ordered when its level falls below the minimum
- *orders(item, amount)*, which notes the amount of the item to be ordered.

We can then use the trigger shown in Figure 6.4 for reordering the item.

Note that we have been careful to place an order only when the amount falls from above the minimum level to below the minimum level. If we only check that the new value after an update is below the minimum level, we may place an order erroneously when the item has already been reordered.

Many database systems provide nonstandard trigger implementations, or implement only some of the trigger features. For instance, many database systems do not implement the **before** clause, and the keyword **on** is used instead of **after**. They may not implement the **referencing** clause. Instead, they may specify transition tables by using the keywords **inserted** or **deleted**. Figure 6.5 illustrates how the overdraft trigger would be written in MS-SQLServer. Read the user manual for the database system you use for more information about the trigger features it supports.

6.4.3 When Not to Use Triggers

There are many good uses for triggers, such as those we have just seen in Section 6.4.2, but some uses are best handled by alternative techniques. For example, in the past, system designers used triggers to maintain summary data. For instance, they used triggers on insert/delete/update of a *employee* relation containing *salary* and *dept* attributes to maintain the total salary of each department. However, many database systems today support materialized views (see Section 3.5.1), which provide a much

```
create trigger overdraft-trigger on account
for update
as
if nrow.balance < 0
begin
   insert into borrower
          (select customer-name, account-number
           from depositor, inserted
           where inserted.account-number = depositor.account-number)
    insert into loan values
          (inserted.account-number, inserted.branch-name, − inserted.balance)
   update account set balance = 0
          from account, inserted
          where account.account-number = inserted.account-number
end
```

Figure 6.5 Example of trigger in MS-SQL server syntax

easier way to maintain summary data. Designers also used triggers extensively for replicating databases; they used triggers on insert/delete/update of each relation to record the changes in relations called **change** or **delta** relations. A separate process copied over the changes to the replica (copy) of the database, and the system executed the changes on the replica. Modern database systems, however, provide built-in facilities for database replication, making triggers unnecessary for replication in most cases.

In fact, many trigger applications, including our example overdraft trigger, can be substituted by "encapsulation" features being introduced in SQL:1999. Encapsulation can be used to ensure that updates to the *balance* attribute of *account* are done only through a special procedure. That procedure would in turn check for negative balance, and carry out the actions of the overdraft trigger. Encapsulations can replace the reorder trigger in a similar manner.

Triggers should be written with great care, since a trigger error detected at run time causes the failure of the insert/delete/update statement that set off the trigger. Furthermore, the action of one trigger can set off another trigger. In the worst case, this could even lead to an infinite chain of triggering. For example, suppose an insert trigger on a relation has an action that causes another (new) insert on the same relation. The insert action then triggers yet another insert action, and so on ad infinitum. Database systems typically limit the length of such chains of triggers (for example to 16 or 32), and consider longer chains of triggering an error.

Triggers are occasionally called *rules*, or *active rules*, but should not be confused with Datalog rules (see Section 5.2), which are really view definitions.

6.5 Security and Authorization

The data stored in the database need protection from unauthorized access and malicious destruction or alteration, in addition to the protection against accidental introduction of inconsistency that integrity constraints provide. In this section, we examine the ways in which data may be misused or intentionally made inconsistent. We then present mechanisms to guard against such occurrences.

6.5.1 Security Violations

Among the forms of malicious access are:

- Unauthorized reading of data (theft of information)
- Unauthorized modification of data
- Unauthorized destruction of data

Database security refers to protection from malicious access. Absolute protection of the database from malicious abuse is not possible, but the cost to the perpetrator can be made high enough to deter most if not all attempts to access the database without proper authority.

To protect the database, we must take security measures at several levels:

- **Database system**. Some database-system users may be authorized to access only a limited portion of the database. Other users may be allowed to issue queries, but may be forbidden to modify the data. It is the responsibility of the database system to ensure that these authorization restrictions are not violated.
- **Operating system**. No matter how secure the database system is, weakness in operating-system security may serve as a means of unauthorized access to the database.
- **Network**. Since almost all database systems allow remote access through terminals or networks, software-level security within the network software is as important as physical security, both on the Internet and in private networks.
- **Physical**. Sites with computer systems must be physically secured against armed or surreptitious entry by intruders.
- **Human**. Users must be authorized carefully to reduce the chance of any user giving access to an intruder in exchange for a bribe or other favors.

Security at all these levels must be maintained if database security is to be ensured. A weakness at a low level of security (physical or human) allows circumvention of strict high-level (database) security measures.

In the remainder of this section, we shall address security at the database-system level. Security at the physical and human levels, although important, is beyond the scope of this text.

Security within the operating system is implemented at several levels, ranging from passwords for access to the system to the isolation of concurrent processes running within the system. The file system also provides some degree of protection. The bibliographical notes reference coverage of these topics in operating-system texts. Finally, network-level security has gained widespread recognition as the Internet has evolved from an academic research platform to the basis of international electronic commerce. The bibliographic notes list textbook coverage of the basic principles of network security. We shall present our discussion of security in terms of the relational-data model, although the concepts of this chapter are equally applicable to all data models.

6.5.2 Authorization

We may assign a user several forms of authorization on parts of the database. For example,

- **Read authorization** allows reading, but not modification, of data.
- **Insert authorization** allows insertion of new data, but not modification of existing data.
- **Update authorization** allows modification, but not deletion, of data.
- **Delete authorization** allows deletion of data.

We may assign the user all, none, or a combination of these types of authorization.

In addition to these forms of authorization for access to data, we may grant a user authorization to modify the database schema:

- **Index authorization** allows the creation and deletion of indices.
- **Resource authorization** allows the creation of new relations.
- **Alteration authorization** allows the addition or deletion of attributes in a relation.
- **Drop authorization** allows the deletion of relations.

The **drop** and **delete** authorization differ in that **delete** authorization allows deletion of tuples only. If a user deletes all tuples of a relation, the relation still exists, but it is empty. If a relation is dropped, it no longer exists.

We regulate the ability to create new relations through **resource** authorization. A user with **resource** authorization who creates a new relation is given all privileges on that relation automatically.

Index authorization may appear unnecessary, since the creation or deletion of an index does not alter data in relations. Rather, indices are a structure for performance enhancements. However, indices also consume space, and all database modifications are required to update indices. If **index** authorization were granted to all users, those who performed updates would be tempted to delete indices, whereas those who issued queries would be tempted to create numerous indices. To allow the *database administrator* to regulate the use of system resources, it is necessary to treat index creation as a privilege.

The ultimate form of authority is that given to the database administrator. The database administrator may authorize new users, restructure the database, and so on. This form of authorization is analogous to that of a **superuser** or operator for an operating system.

6.5.3 Authorization and Views

In Chapter 3, we introduced the concept of *views* as a means of providing a user with a personalized model of the database. A view can hide data that a user does not need to see. The ability of views to hide data serves both to simplify usage of the system and to enhance security. Views simplify system usage because they restrict the user's attention to the data of interest. Although a user may be denied direct access to a relation, that user may be allowed to access part of that relation through a view. Thus, a combination of relational-level security and view-level security limits a user's access to precisely the data that the user needs.

In our banking example, consider a clerk who needs to know the names of all customers who have a loan at each branch. This clerk is not authorized to see information regarding specific loans that the customer may have. Thus, the clerk must be denied direct access to the *loan* relation. But, if she is to have access to the information needed, the clerk must be granted access to the view *cust-loan*, which consists of only the names of customers and the branches at which they have a loan. This view can be defined in SQL as follows:

```
create view cust-loan as
   (select branch-name, customer-name
    from borrower, loan
    where borrower.loan-number = loan.loan-number)
```

Suppose that the clerk issues the following SQL query:

```
select *
from cust-loan
```

Clearly, the clerk is authorized to see the result of this query. However, when the query processor translates it into a query on the actual relations in the database, it produces a query on *borrower* and *loan*. Thus, the system must check authorization on the clerk's query before it begins query processing.

Creation of a view does not require **resource** authorization. A user who creates a view does not necessarily receive all privileges on that view. She receives only those privileges that provide no additional authorization beyond those that she already had. For example, a user cannot be given **update** authorization on a view without having **update** authorization on the relations used to define the view. If a user creates a view on which no authorization can be granted, the system will deny the view creation request. In our *cust-loan* view example, the creator of the view must have **read** authorization on both the *borrower* and *loan* relations.

6.5.4 Granting of Privileges

A user who has been granted some form of authorization may be allowed to pass on this authorization to other users. However, we must be careful how authorization may be passed among users, to ensure that such authorization can be revoked at some future time.

Consider, as an example, the granting of update authorization on the *loan* relation of the bank database. Assume that, initially, the database administrator grants update authorization on *loan* to users U_1, U_2, and U_3, who may in turn pass on this authorization to other users. The passing of authorization from one user to another can be represented by an **authorization graph**. The nodes of this graph are the users. The graph includes an edge $U_i \rightarrow U_j$ if user U_i grants update authorization on *loan* to U_j. The root of the graph is the database administrator. In the sample graph in Figure 6.6, observe that user U_5 is granted authorization by both U_1 and U_2; U_4 is granted authorization by only U_1.

A user has an authorization *if and only if* there is a path from the root of the authorization graph (namely, the node representing the database administrator) down to the node representing the user.

Suppose that the database administrator decides to revoke the authorization of user U_1. Since U_4 has authorization from U_1, that authorization should be revoked as well. However, U_5 was granted authorization by both U_1 and U_2. Since the database administrator did not revoke update authorization on *loan* from U_2, U_5 retains update authorization on *loan*. If U_2 eventually revokes authorization from U_5, then U_5 loses the authorization.

A pair of devious users might attempt to defeat the rules for revocation of authorization by granting authorization to each other, as shown in Figure 6.7a. If the database administrator revokes authorization from U_2, U_2 retains authorization through U_3, as in Figure 6.7b. If authorization is revoked subsequently from U_3, U_3 appears to retain authorization through U_2, as in Figure 6.7c. However, when the database administrator revokes authorization from U_3, the edges from U_3 to U_2 and from U_2 to U_3 are no longer part of a path starting with the database administrator.

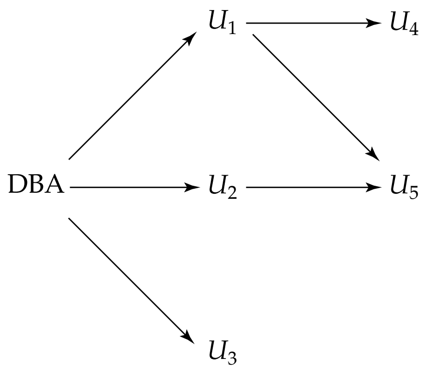

$$\begin{tikzcd}
 & U_1 \arrow[r] \arrow[dr] & U_4 \\
\text{DBA} \arrow[ur] \arrow[r] \arrow[dr] & U_2 \arrow[r] & U_5 \\
 & U_3 &
\end{tikzcd}$$

Figure 6.6 Authorization-grant graph.

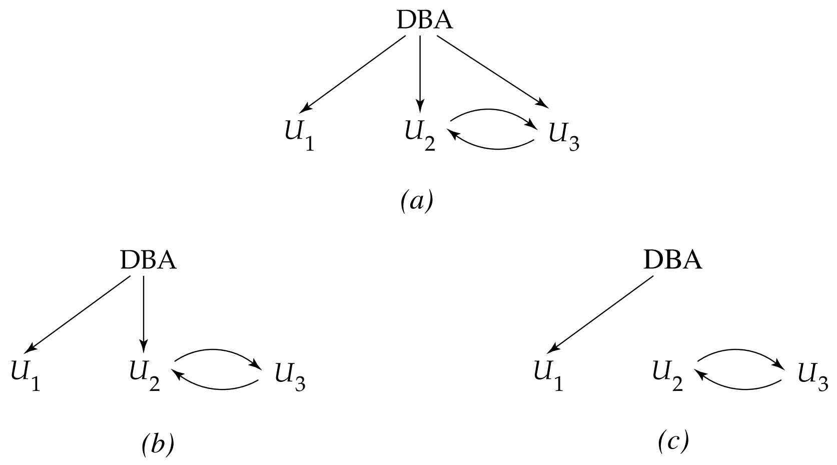


Figure 6.7 Attempt to defeat authorization revocation.

We require that all edges in an authorization graph be part of some path originating with the database administrator. The edges between U_2 and U_3 are deleted, and the resulting authorization graph is as in Figure 6.8.

6.5.5 Notion of Roles

Consider a bank where there are many tellers. Each teller must have the same types of authorizations to the same set of relations. Whenever a new teller is appointed, she will have to be given all these authorizations individually.

A better scheme would be to specify the authorizations that every teller is to be given, and to separately identify which database users are tellers. The system can use these two pieces of information to determine the authorizations of each person who is a teller. When a new person is hired as a teller, a user identifier must be allocated to him, and he must be identified as a teller. Individual permissions given to tellers need not be specified again.

The notion of **roles** captures this scheme. A set of roles is created in the database. Authorizations can be granted to roles, in exactly the same fashion as they are granted to individual users. Each database user is granted a set of roles (which may be empty) that he or she is authorized to perform.

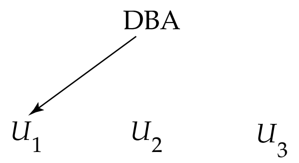


Figure 6.8 Authorization graph.

In our bank database, examples of roles could include *teller*, *branch-manager*, *auditor*, and *system-administrator*.

A less preferable alternative would be to create a *teller* userid, and permit each teller to connect to the database using the *teller* userid. The problem with this scheme is that it would not be possible to identify exactly which teller carried out a transaction, leading to security risks. The use of roles has the benefit of requiring users to connect to the database with their own userid.

Any authorization that can be granted to a user can be granted to a role. Roles are granted to users just as authorizations are. And like other authorizations, a user may also be granted authorization to grant a particular role to others. Thus, branch managers may be granted authorization to grant the *teller* role.

6.5.6 Audit Trails

Many secure database applications require an **audit trail** be maintained. An audit trail is a log of all changes (inserts/deletes/updates) to the database, along with information such as which user performed the change and when the change was performed.

The audit trail aids security in several ways. For instance, if the balance on an account is found to be incorrect, the bank may wish to trace all the updates performed on the account, to find out incorrect (or fraudulent) updates, as well as the persons who carried out the updates. The bank could then also use the audit trail to trace all the updates performed by these persons, in order to find other incorrect or fraudulent updates.

It is possible to create an audit trail by defining appropriate triggers on relation updates (using system-defined variables that identify the user name and time). However, many database systems provide built-in mechanisms to create audit trails, which are much more convenient to use. Details of how to create audit trails vary across database systems, and you should refer the database system manuals for details.

6.6 Authorization in SQL

The SQL language offers a fairly powerful mechanism for defining authorizations. We describe these mechanisms, as well as their limitations, in this section.

6.6.1 Privileges in SQL

The SQL standard includes the privileges **delete**, **insert**, **select**, and **update**. The **select** privilege corresponds to the **read** privilege. SQL also includes a **references** privilege that permits a user/role to declare foreign keys when creating relations. If the relation to be created includes a foreign key that references attributes of another relation, the user/role must have been granted **references** privilege on those attributes. The reason that the **references** privilege is a useful feature is somewhat subtle; we explain the reason later in this section.

The SQL data-definition language includes commands to grant and revoke privileges. The **grant** statement is used to confer authorization. The basic form of this statement is:

grant <privilege list> **on** <relation name or view name> **to** <user/role list>

The *privilege list* allows the granting of several privileges in one command.

The following **grant** statement grants users U_1, U_2, and U_3 **select** authorization on the *account* relation:

grant select on *account* **to** U_1, U_2, U_3

The **update** authorization may be given either on all attributes of the relation or on only some. If **update** authorization is included in a **grant** statement, the list of attributes on which update authorization is to be granted optionally appears in parentheses immediately after the **update** keyword. If the list of attributes is omitted, the update privilege will be granted on all attributes of the relation.

This **grant** statement gives users U_1, U_2, and U_3 update authorization on the *amount* attribute of the *loan* relation:

grant update (*amount*) **on** *loan* **to** U_1, U_2, U_3

The **insert** privilege may also specify a list of attributes; any inserts to the relation must specify only these attributes, and the system either gives each of the remaining attributes default values (if a default is defined for the attribute) or sets them to null.

The SQL **references** privilege is granted on specific attributes in a manner like that for the **update** privilege. The following **grant** statement allows user U_1 to create relations that reference the key *branch-name* of the *branch* relation as a foreign key:

grant references (*branch-name*) **on** *branch* **to** U_1

Initially, it may appear that there is no reason ever to prevent users from creating foreign keys referencing another relation. However, recall from Section 6.2 that foreign-key constraints restrict deletion and update operations on the referenced relation. In the preceding example, if U_1 creates a foreign key in a relation r referencing the *branch-name* attribute of the *branch* relation, and then inserts a tuple into r pertaining to the Perryridge branch, it is no longer possible to delete the Perryridge branch from the *branch* relation without also modifying relation r. Thus, the definition of a foreign key by U_1 restricts future activity by other users; therefore, there is a need for the **references** privilege.

The privilege **all privileges** can be used as a short form for all the allowable privileges. Similarly, the user name **public** refers to all current and future users of the system. SQL also includes a **usage** privilege that authorizes a user to use a specified domain (recall that a domain corresponds to the programming-language notion of a type, and may be user defined).

6.6.2 Roles

Roles can be created in SQL:1999 as follows

create role *teller*

Roles can then be granted privileges just as the users can, as illustrated in this statement:

grant select on *account*
to *teller*

Roles can be asigned to the users, as well as to some other roles, as these statements show.

grant *teller* **to** john
create role *manager*
grant *teller* **to** *manager*
grant *manager* **to** mary

Thus the privileges of a user or a role consist of

- All privileges directly granted to the user/role
- All privileges granted to roles that have been granted to the user/role

Note that there can be a chain of roles; for example, the role *employee* may be granted to all *tellers*. In turn the role *teller* is granted to all *manager*s. Thus, the *manager* role inherits all privileges granted to the roles *employee* and to *teller* in addition to privileges granted directly to *manager*.

6.6.3 The Privilege to Grant Privileges

By default, a user/role that is granted a privilege is not authorized to grant that privilege to another user/role. If we wish to grant a privilege and to allow the recipient to pass the privilege on to other users, we append the **with grant option** clause to the appropriate **grant** command. For example, if we wish to allow U_1 the **select** privilege on *branch* and allow U_1 to grant this privilege to others, we write

grant select on *branch* **to** U_1 **with grant option**

To revoke an authorization, we use the **revoke** statement. It takes a form almost identical to that of **grant**:

revoke <privilege list> **on** <relation name or view name>
from <user/role list> **[restrict** | **cascade]**

Thus, to revoke the privileges that we granted previously, we write

revoke select on *branch* **from** U_1, U_2, U_3
revoke update (*amount*) **on** *loan* **from** U_1, U_2, U_3
revoke references (*branch-name*) **on** *branch* **from** U_1

As we saw in Section 6.5.4, the revocation of a privilege from a user/role may cause other users/roles also to lose that privilege. This behavior is called *cascading of the revoke*. In most database systems, cascading is the default behavior; the keyword **cascade** can thus be omitted, as we have done in the preceding examples. The **revoke** statement may alternatively specify **restrict**:

revoke select on *branch* **from** U_1, U_2, U_3 **restrict**

In this case, the system returns an error if there are any cascading revokes, and does not carry out the revoke action. The following **revoke** statement revokes only the grant option, rather than the actual **select** privilege:

revoke grant option for select on *branch* **from** U_1

6.6.4 Other Features

The creator of an object (relation/view/role) gets all privileges on the object, including the privilege to grant privileges to others.

The SQL standard specifies a primitive authorization mechanism for the database schema: Only the owner of the schema can carry out any modification to the schema. Thus, schema modifications—such as creating or deleting relations, adding or dropping attributes of relations, and adding or dropping indices—may be executed by only the owner of the schema. Several database implementations have more powerful authorization mechanisms for database schemas, similar to those discussed earlier, but these mechanisms are nonstandard.

6.6.5 Limitations of SQL Authorization

The current SQL standards for authorization have some shortcomings. For instance, suppose you want all students to be able to see their own grades, but not the grades of anyone else. Authorization must then be at the level of individual tuples, which is not possible in the SQL standards for authorization.

Furthermore, with the growth in the Web, database accesses come primarily from Web application servers. The end users may not have individual user identifiers on the database, and indeed there may only be a single user identifier in the database corresponding to all users of an application server.

The task of authorization then falls on the application server; the entire authorization scheme of SQL is bypassed. The benefit is that fine-grained authorizations, such as those to individual tuples, can be implemented by the application. The problems are these:

- The code for checking authorization becomes intermixed with the rest of the application code.

- Implementing authorization through application code, rather than specifying it declaratively in SQL, makes it hard to ensure the absence of loopholes. Because of an oversight, one of the application programs may not check for authorization, allowing unauthorized users access to confidential data. Verifying that all application programs make all required authorization checks involves reading through all the application server code, a formidable task in a large system.

6.7 Encryption and Authentication

The various provisions that a database system may make for authorization may still not provide sufficient protection for highly sensitive data. In such cases, data may be stored in **encrypt**ed form. It is not possible for encrypted data to be read unless the reader knows how to decipher (**decrypt**) them. Encryption also forms the basis of good schemes for authenticating users to a database.

6.7.1 Encryption Techniques

There are a vast number of techniques for the encryption of data. Simple encryption techniques may not provide adequate security, since it may be easy for an unauthorized user to break the code. As an example of a weak encryption technique, consider the substitution of each character with the next character in the alphabet. Thus,

Perryridge

becomes

Qfsszsjehf

If an unauthorized user sees only "Qfsszsjehf," she probably has insufficient information to break the code. However, if the intruder sees a large number of encrypted branch names, she could use statistical data regarding the relative frequency of characters to guess what substitution is being made (for example, *E* is the most common letter in English text, followed by *T, A, O, N, I* and so on).

A good encryption technique has the following properties:

- It is relatively simple for authorized users to encrypt and decrypt data.
- It depends not on the secrecy of the algorithm, but rather on a parameter of the algorithm called the *encryption key*.
- Its encryption key is extremely difficult for an intruder to determine.

One approach, the *Data Encryption Standard* (DES), issued in 1977, does both a substitution of characters and a rearrangement of their order on the basis of an encryption key. For this scheme to work, the authorized users must be provided with the encryption key via a secure mechanism. This requirement is a major weakness, since the scheme is no more secure than the security of the mechanism by which the encryption key is transmitted. The DES standard was reaffirmed in 1983, 1987, and again in 1993. However, weakness in DES was recongnized in 1993 as reaching a point where a new standard to be called the **Advanced Encryption Standard** (AES), needed to be selected. In 2000, the **Rijndael algorithm** (named for the inventors V. Rijmen and J. Daemen), was selected to be the AES. The Rijndael algorithm was chosen for its significantly stronger level of security and its relative ease of implementation on current computer systems as well as such devices as smart cards. Like the DES standard, the Rijndael algorithm is a shared-key (or, symmetric key) algorithm in which the authorized users share a key.

Public-key encryption is an alternative scheme that avoids some of the problems that we face with the DES. It is based on two keys; a *public key* and a *private key*. Each user U_i has a public key E_i and a private key D_i. All public keys are published: They can be seen by anyone. Each private key is known to only the one user to whom the key belongs. If user U_1 wants to store encrypted data, U_1 encrypts them using public key E_1. Decryption requires the private key D_1.

Because the encryption key for each user is public, it is possible to exchange information securely by this scheme. If user U_1 wants to share data with U_2, U_1 encrypts the data using E_2, the public key of U_2. Since only user U_2 knows how to decrypt the data, information is transferred securely.

For public-key encryption to work, there must be a scheme for encryption that can be made public without making it easy for people to figure out the scheme for decryption. In other words, it must be hard to deduce the private key, given the public key. Such a scheme does exist and is based on these conditions:

- There is an efficient algorithm for testing whether or not a number is prime.
- No efficient algorithm is known for finding the prime factors of a number.

For purposes of this scheme, data are treated as a collection of integers. We create a public key by computing the product of two large prime numbers: P_1 and P_2. The private key consists of the pair (P_1, P_2). The decryption algorithm cannot be used successfully if only the product P_1P_2 is known; it needs the individual values P_1 and P_2. Since all that is published is the product P_1P_2, an unauthorized user would need to be able to factor P_1P_2 to steal data. By choosing P_1 and P_2 to be sufficiently large (over 100 digits), we can make the cost of factoring P_1P_2 prohibitively high (on the order of years of computation time, on even the fastest computers).

The details of public-key encryption and the mathematical justification of this technique's properties are referenced in the bibliographic notes.

Although public-key encryption by this scheme is secure, it is also computationally expensive. A hybrid scheme used for secure communication is as follows: DES keys are exchanged via a public-key−encryption scheme, and DES encryption is used on the data transmitted subsequently.

6.7.2 Authentication

Authentication refers to the task of verifying the identity of a person/software connecting to a database. The simplest form of authentication consists of a secret password which must be presented when a connection is opened to a database.

Password-based authentication is used widely by operating systems as well as databases. However, the use of passwords has some drawbacks, especially over a network. If an eavesdropper is able to "sniff" the data being sent over the network, she may be able to find the password as it is being sent across the network. Once the eavesdropper has a user name and password, she can connect to the database, pretending to be the legitimate user.

A more secure scheme involves a **challenge-response** system. The database system sends a challenge string to the user. The user encrypts the challenge string using a secret password as encryption key, and then returns the result. The database system can verify the authenticity of the user by decrypting the string with the same secret password, and checking the result with the original challenge string. This scheme ensures that no passwords travel across the network.

Public-key systems can be used for encryption in challenge−response systems. The database system encrypts a challenge string using the user's public key and sends it to the user. The user decrypts the string using her private key, and returns the result to the database system. The database system then checks the response. This scheme has the added benefit of not storing the secret password in the database, where it could potentially be seen by system administrators.

Another interesting application of public-key encryption is in **digital signatures** to verify authenticity of data; digital signatures play the electronic role of physical signatures on documents. The private key is used to sign data, and the signed data can be made public. Anyone can verify them by the public key, but no one could have generated the signed data without having the private key. Thus, we can **authenticate** the data; that is, we can verify that the data were indeed created by the person who claims to have created them.

Furthermore, digital signatures also serve to ensure **nonrepudiation**. That is, in case the person who created the data later claims she did not create it (the electronic equivalent of claiming not to have signed the check), we can prove that that person must have created the data (unless her private key was leaked to others).

6.8 Summary

- Integrity constraints ensure that changes made to the database by authorized users do not result in a loss of data consistency.

- In earlier chapters, we considered several forms of constraints, including key declarations and the declaration of the form of a relationship (many to many, many to one, one to one). In this chapter, we considered several additional forms of constraints, and discussed mechanisms for ensuring the maintenance of these constraints.

- Domain constraints specify the set of possible values that may be associated with an attribute. Such constraints may also prohibit the use of null values for particular attributes.

- Referential-integrity constraints ensure that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
- Domain constraints, and referential-integrity constraints are relatively easy to test. Use of more complex constraints may lead to substantial overhead. We saw two ways to express more general constraints. Assertions are declarative expressions that state predicates that we require always to be true.
- Triggers define actions to be executed automatically when certain events occur and corresponding conditions are satisfied. Triggers have many uses, such as implementing business rules, audit logging, and even carrying out actions outside the database system. Although triggers were added only lately to the SQL standard as part of SQL:1999, most database systems have long implemented triggers.
- The data stored in the database need to be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency.
- It is easier to protect against accidental loss of data consistency than to protect against malicious access to the database. Absolute protection of the database from malicious abuse is not possible, but the cost to the perpetrator can be made sufficiently high to deter most, if not all, attempts to access the database without proper authority.
- A user may have several forms of authorization on parts of the database. Authorization is a means by which the database system can be protected against malicious or unauthorized access.
- A user who has been granted some form of authority may be allowed to pass on this authority to other users. However, we must be careful about how authorization can be passed among users if we are to ensure that such authorization can be revoked at some future time.
- Roles help to assign a set of privileges to a user according to on the role that the user plays in the organization.
- The various authorization provisions in a database system may not provide sufficient protection for highly sensitive data. In such cases, data can be *encrypted*. Only a user who knows how to decipher (*decrypt*) the encrypted data can read them. Encryption also forms the basis for secure authentication of users.

Review Terms

- Domain constraints
- Check clause
- Referential integrity
- Primary key constraint
- Unique constraint
- Foreign key constraint

- Cascade
- Assertion
- Trigger
- Event-condition-action model
- Before and after triggers
- Transition variables and tables
- Database security
- Levels of security
- Authorization
- Privileges
  - ☐ Read
  - ☐ Insert
  - ☐ Update
  - ☐ Delete
  - ☐ Index
  - ☐ Resource
  - ☐ Alteration
  - ☐ Drop
  - ☐ Grant
  - ☐ All privileges
- Authorization graph
- Granting of privileges
- Roles
- Encryption
- Secret-key encryption
- Public-key encryption
- Authentication
- Challenge−response system
- Digital signature
- Nonrepudiation

Exercises

6.1 Complete the SQL DDL definition of the bank database of Figure 6.2 to include the relations *loan* and *borrower*.

6.2 Consider the following relational database:

employee (*employee-name*, *street*, *city*)
works (*employee-name*, *company-name*, *salary*)
company (*company-name*, *city*)
manages (*employee-name*, *manager-name*)

Give an SQL DDL definition of this database. Identify referential-integrity constraints that should hold, and include them in the DDL definition.

6.3 Referential-integrity constraints as defined in this chapter involve exactly two relations. Consider a database that includes the following relations:

salaried-worker (*name, office, phone, salary*)
hourly-worker (*name, hourly-wage*)
address (*name, street, city*)

Suppose that we wish to require that every name that appears in *address* appear in either *salaried-worker* or *hourly-worker*, but not necessarily in both.

a. Propose a syntax for expressing such constraints.
b. Discuss the actions that the system must take to enforce a constraint of this form.

6.4 SQL allows a foreign-key dependency to refer to the same relation, as in the following example:

```
create table manager
    (employee-name   char(20)  not null
     manager-name    char(20)  not null,
     primary key employee-name,
     foreign key (manager-name) references manager
                                on delete cascade )
```

Here, *employee-name* is a key to the table *manager*, meaning that each employee has at most one manager. The foreign-key clause requires that every manager also be an employee. Explain exactly what happens when a tuple in the relation *manager* is deleted.

6.5 Suppose there are two relations r and s, such that the foreign key B of r references the primary key A of s. Describe how the trigger mechanism can be used to implement the **on delete cascade** option, when a tuple is deleted from s.

6.6 Write an assertion for the bank database to ensure that the assets value for the Perryridge branch is equal to the sum of all the amounts lent by the Perryridge branch.

6.7 Write an SQL trigger to carry out the following action: On **delete** of an account, for each owner of the account, check if the owner has any remaining accounts, and if she does not, delete her from the *depositor* relation.

6.8 Consider a view *branch-cust* defined as follows:

```
create view branch-cust as
    select branch-name, customer-name
    from depositor, account
    where depositor.account-number = account.account-number
```

Suppose that the view is *materialized*, that is, the view is computed and stored. Write active rules to *maintain* the view, that is, to keep it up to date on insertions to and deletions from *depositor* or *account*. Do not bother about updates.

6.9 Make a list of security concerns for a bank. For each item on your list, state whether this concern relates to physical security, human security, operating-system security, or database security.

6.10 Using the relations of our sample bank database, write an SQL expression 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.

6.11 For each of the views that you defined in Exercise 6.10, explain how updates would be performed (if they should be allowed at all). *Hint*: See the discussion of views in Chapter 3.

6.12 In Chapter 3, we described the use of views to simplify access to the database by users who need to see only part of the database. In this chapter, we described the use of views as a security mechanism. Do these two purposes for views ever conflict? Explain your answer.

6.13 What is the purpose of having separate categories for index authorization and resource authorization?

6.14 Database systems that store each relation in a separate operating-system file may use the operating system's security and authorization scheme, instead of defining a special scheme themselves. Discuss an advantage and a disadvantage of such an approach.

6.15 What are two advantages of encrypting data stored in the database?

6.16 Perhaps the most important data items in any database system are the passwords that control access to the database. Suggest a scheme for the secure storage of passwords. Be sure that your scheme allows the system to test passwords supplied by users who are attempting to log into the system.

Bibliographical Notes

Discussions of integrity constraints in the relational model are offered by Hammer and McLeod [1975], Stonebraker [1975], Eswaran and Chamberlin [1975], Schmid and Swenson [1975] and Codd [1979]. The original SQL proposals for assertions and triggers are discussed in Astrahan et al. [1976], Chamberlin et al. [1976], and Chamberlin et al. [1981]. See the bibliographic notes of Chapter 4 for references to SQL standards and books on SQL.

Discussions of efficient maintenance and checking of semantic-integrity assertions are offered by Hammer and Sarin [1978], Badal and Popek [1979], Bernstein et al. [1980a], Hsu and Imielinski [1985], McCune and Henschen [1989], and Chomicki [1992]. An alternative to using run-time integrity checking is certifying the correctness of programs that access the database. Sheard and Stemple [1989] discusses this approach.

Active databases are databases that support triggers and other mechanisms that permit the database to take actions on occurrence of events. McCarthy and Dayal [1989] discuss the architecture of an active database system based on the event–condition–action formalism. Widom and Finkelstein [1990] describe the architecture of a rule system based on set-oriented rules; the implementation of the rule system on the Starburst extensible database system is presented in Widom et al. [1991]. Consider an execution mechanism that allows a nondeterministic choice of which rule to execute next. A rule system is said to be **confluent** if, regardless of the rule chosen, the final state is the same. Issues of termination, nondeterminism, and confluence of rule systems are discussed in Aiken et al. [1995].

Security aspects of computer systems in general are discussed in Bell and LaPadula [1976] and by US Dept. of Defense [1985]. Security aspects of SQL can be found in the SQL standards and textbooks on SQL referenced in the bibliographic notes of Chapter 4.

Stonebraker and Wong [1974] discusses the Ingres approach to security, which involves modification of users' queries to ensure that users do not access data for which authorization has not been granted. Denning and Denning [1979] survey database security.

Database systems that can produce incorrect answers when necessary for security maintenance are discussed in Winslett et al. [1994] and Tendick and Matloff [1994]. Work on security in relational databases includes that of Stachour and Thuraisingham [1990], Jajodia and Sandhu [1990], and Qian and Lunt [1996]. Operating-system security issues are discussed in most operating-system texts, including Silberschatz and Galvin [1998].

Stallings [1998] provides a textbook description of cryptography. Daemen and Rijmen [2000] present the Rijndael algorithm. The Data Encryption Standard is presented by US Dept. of Commerce [1977]. Public-key encryption is discussed by Rivest et al. [1978]. Other discussions on cryptography include Diffie and Hellman [1979], Simmons [1979], Fernandez et al. [1981], and Akl [1983].

C H A P T E R 7

Relational-Database Design

This chapter continues our discussion of design issues in relational databases. In general, the goal of a relational-database design is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. One approach is to design schemas that are in an appropriate *normal form*. To determine whether a relation schema is in one of the desirable normal forms, we need additional information about the real-world enterprise that we are modeling with the database. In this chapter, we introduce the notion of functional dependencies. We then define normal forms in terms of functional dependencies and other types of data dependencies.

7.1 First Normal Form

The first of the normal forms that we study, **first normal form**, imposes a very basic requirement on relations; unlike the other normal forms, it does not require additional information such as functional dependencies.

A domain is **atomic** if elements of the domain are considered to be indivisible units. We say that a relation schema R is in **first normal form** (1NF) if the domains of all attributes of R are atomic.

A set of names is an example of a nonatomic value. For example, if the schema of a relation *employee* included an attribute *children* whose domain elements are sets of names, the schema would not be in first normal form.

Composite attributes, such as an attribute *address* with component attributes *street* and *city*, also have nonatomic domains.

Integers are assumed to be atomic, so the set of integers is an atomic domain; the set of all sets of integers is a nonatomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts—namely, the integers making up the set. But the important issue is not what the domain itself is, but rather how we use domain elements in our database.

The domain of all integers would be nonatomic if we considered each integer to be an ordered list of digits.

As a practical illustration of the above point, consider an organization that assigns employees identification numbers of the following form: The first two letters specify the department and the remaining four digits are a unique number within the department for the employee. Examples of such numbers would be $CS0012$ and $EE1127$. Such identification numbers can be divided into smaller units, and are therefore nonatomic. If a relation schema had an attribute whose domain consists of identification numbers encoded as above, the schema would not be in first normal form.

When such identification numbers are used, the department of an employee can be found by writing code that breaks up the structure of an identification number. Doing so requires extra programming, and information gets encoded in the application program rather than in the database. Further problems arise if such identification numbers are used as primary keys: When an employee changes department, the employee's identification number must be changed everywhere it occurs, which can be a difficult task, or the code that interprets the number would give a wrong result.

The use of set valued attributes can lead to designs with redundant storage of data, which in turn can result in inconsistencies. For instance, instead of the relationship between accounts and customers being represented as a separate relation *depositor*, a database designer may be tempted to store a set of *owners* with each account, and a set of *accounts* with each customer. Whenever an account is created, or the set of owners of an account is updated, the update has to be performed at two places; failure to perform both updates can leave the database in an inconsistent state. Keeping only one of these sets would avoid repeated information, but would complicate some queries. Set valued attributes are also more complicated to write queries with, and more complicated to reason about.

In this chapter we consider only atomic domains, and assume that relations are in first normal form. Although we have not mentioned first normal form earlier, when we introduced the relational model in Chapter 3 we stated that attribute values must be atomic.

Some types of nonatomic values can be useful, although they should be used with care. For example, composite valued attributes are often useful, and set valued attributes are also useful in many cases, which is why both are supported in the E-R model. In many domains where entities have a complex structure, forcing a first normal form representation represents an unnecessary burden on the application programmer, who has to write code to convert data into atomic form. There is also a runtime overhead of converting data back and forth from the atomic form. Support for nonatomic values can thus be very useful in such domains. In fact, modern database systems do support many types of nonatomic values, as we will see in Chapters 8 and 9. However, in this chapter we restrict ourselves to relations in first normal form.

7.2 Pitfalls in Relational-Database Design

Before we continue our discussion of normal forms, let us look at what can go wrong in a bad database design. Among the undesirable properties that a bad design may have are:

- Repetition of information
- Inability to represent certain information

We shall discuss these problems with the help of a modified database design for our banking example: In contrast to the relation schema used in Chapters 3 to 6, suppose the information concerning loans is kept in one single relation, *lending*, which is defined over the relation schema

Lending-schema = (*branch-name, branch-city, assets, customer-name, loan-number, amount*)

Figure 7.1 shows an instance of the relation *lending* (*Lending-schema*). A tuple *t* in the *lending* relation has the following intuitive meaning:

- *t*[*assets*] is the asset figure for the branch named *t*[*branch-name*].
- *t*[*branch-city*] is the city in which the branch named *t*[*branch-name*] is located.
- *t*[*loan-number*] is the number assigned to a loan made by the branch named *t*[*branch-name*] to the customer named *t*[*customer-name*].
- *t*[*amount*] is the amount of the loan whose number is *t*[*loan-number*].

Suppose that we wish to add a new loan to our database. Say that the loan is made by the Perryridge branch to Adams in the amount of $1500. Let the *loan-number* be L-31. In our design, we need a tuple with values on all the attributes of *Lending-schema*. Thus, we must repeat the asset and city data for the Perryridge branch, and must add the tuple

(Perryridge, Horseneck, 1700000, Adams, L-31, 1500)

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Figure 7.1 Sample *lending* relation.

to the *lending* relation. In general, the asset and city data for a branch must appear once for each loan made by that branch.

The repetition of information in our alternative design is undesirable. Repeating information wastes space. Furthermore, it complicates updating the database. Suppose, for example, that the assets of the Perryridge branch change from 1700000 to 1900000. Under our original design, one tuple of the *branch* relation needs to be changed. Under our alternative design, many tuples of the *lending* relation need to be changed. Thus, updates are more costly under the alternative design than under the original design. When we perform the update in the alternative database, we must ensure that *every* tuple pertaining to the Perryridge branch is updated, or else our database will show two different asset values for the Perryridge branch.

That observation is central to understanding why the alternative design is bad. We know that a bank branch has a unique value of assets, so given a branch name we can uniquely identify the assets value. On the other hand, we know that a branch may make many loans, so given a branch name, we cannot uniquely determine a loan number. In other words, we say that the *functional dependency*

$$\textit{branch-name} \rightarrow \textit{assets}$$

holds on *Lending-schema*, but we do not expect the functional dependency *branch-name* → *loan-number* to hold. The fact that a branch has a particular value of assets, and the fact that a branch makes a loan are independent, and, as we have seen, these facts are best represented in separate relations. We shall see that we can use functional dependencies to specify formally when a database design is good.

Another problem with the *Lending-schema* design is that we cannot represent directly the information concerning a branch (*branch-name*, *branch-city*, *assets*) unless there exists at least one loan at the branch. This is because tuples in the *lending* relation require values for *loan-number*, *amount*, and *customer-name*.

One solution to this problem is to introduce *null values*, as we did to handle updates through views. Recall, however, that null values are difficult to handle, as we saw in Section 3.3.4. If we are not willing to deal with null values, then we can create the branch information only when the first loan application at that branch is made. Worse, we would have to delete this information when all the loans have been paid. Clearly, this situation is undesirable, since, under our original database design, the branch information would be available regardless of whether or not loans are currently maintained in the branch, and without resorting to null values.

7.3 Functional Dependencies

Functional dependencies play a key role in differentiating good database designs from bad database designs. A **functional dependency** is a type of constraint that is a generalization of the notion of *key*, as discussed in Chapters 2 and 3.

7.3.1 Basic Concepts

Functional dependencies are constraints on the set of legal relations. They allow us to express facts about the enterprise that we are modeling with our database.

In Chapter 2, we defined the notion of a *superkey* as follows. Let *R* be a relation schema. A subset *K* of *R* is a **superkey** of *R* if, in any legal relation $r(R)$, for all pairs t_1 and t_2 of tuples in *r* such that $t_1 \neq t_2$, then $t_1[K] \neq t_2[K]$. That is, no two tuples in any legal relation $r(R)$ may have the same value on attribute set *K*.

The notion of functional dependency generalizes the notion of superkey. Consider a relation schema *R*, and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **functional dependency**

$$\alpha \rightarrow \beta$$

holds on schema R if, in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in *r* such that $t_1[\alpha] = t_2[\alpha]$, it is also the case that $t_1[\beta] = t_2[\beta]$.

Using the functional-dependency notation, we say that *K* is a superkey of *R* if *K* $\rightarrow$ *R*. That is, *K* is a superkey if, whenever $t_1[K] = t_2[K]$, it is also the case that $t_1[R] = t_2[R]$ (that is, $t_1 = t_2$).

Functional dependencies allow us to express constraints that we cannot express with superkeys. Consider the schema

Loan-info-schema = (*loan-number, branch-name, customer-name, amount*)

which is simplification of the *Lending-schema* that we saw earlier. The set of functional dependencies that we expect to hold on this relation schema is

loan-number $\rightarrow$ *amount*
loan-number $\rightarrow$ *branch-name*

We would not, however, expect the functional dependency

loan-number $\rightarrow$ *customer-name*

to hold, since, in general, a given loan can be made to more than one customer (for example, to both members of a husband–wife pair).

We shall use functional dependencies in two ways:

1. To test relations to see whether they are legal under a given set of functional dependencies. If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* **satisfies** *F*.

2. To specify constraints on the set of legal relations. We shall thus concern ourselves with *only* those relations that satisfy a given set of functional dependencies. If we wish to constrain ourselves to relations on schema *R* that satisfy a set *F* of functional dependencies, we say that *F* **holds** on R.

Let us consider the relation *r* of Figure 7.2, to see which functional dependencies are satisfied. Observe that $A \rightarrow C$ is satisfied. There are two tuples that have an *A* value of a_1. These tuples have the same *C* value—namely, c_1. Similarly, the two tuples with an *A* value of a_2 have the same *C* value, c_2. There are no other pairs of distinct tuples that have the same *A* value. The functional dependency $C \rightarrow A$ is not satisfied, however. To see that it is not, consider the tuples $t_1 = (a_2, b_3, c_2, d_3)$ and

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_2	c_2	d_3
a_3	b_3	c_2	d_4

Figure 7.2 Sample relation *r*.

$t_2 = (a_3, b_3, c_2, d_4)$. These two tuples have the same *C* values, c_2, but they have different *A* values, a_2 and a_3, respectively. Thus, we have found a pair of tuples t_1 and t_2 such that $t_1[C] = t_2[C]$, but $t_1[A] \neq t_2[A]$.

Many other functional dependencies are satisfied by *r*, including, for example, the functional dependency $AB \rightarrow D$. Note that we use *AB* as a shorthand for {*A*,*B*}, to conform with standard practice. Observe that there is no pair of distinct tuples t_1 and t_2 such that $t_1[AB] = t_2[AB]$. Therefore, if $t_1[AB] = t_2[AB]$, it must be that $t_1 = t_2$ and, thus, $t_1[D] = t_2[D]$. So, *r* satisfies $AB \rightarrow D$.

Some functional dependencies are said to be **trivial** because they are satisfied by all relations. For example, $A \rightarrow A$ is satisfied by all relations involving attribute *A*. Reading the definition of functional dependency literally, we see that, for all tuples t_1 and t_2 such that $t_1[A] = t_2[A]$, it is the case that $t_1[A] = t_2[A]$. Similarly, $AB \rightarrow A$ is satisfied by all relations involving attribute *A*. In general, a functional dependency of the form $\alpha \rightarrow \beta$ is **trivial** if $\beta \subseteq \alpha$.

To distinguish between the concepts of a relation satisfying a dependency and a dependency holding on a schema, we return to the banking example. If we consider the *customer* relation (on *Customer-schema*) in Figure 7.3, we see that *customer-street* → *customer-city* is satisfied. However, we believe that, in the real world, two cities

customer-name	*customer-street*	*customer-city*
Jones	Main	Harrison
Smith	North	Rye
Hayes	Main	Harrison
Curry	North	Rye
Lindsay	Park	Pittsfield
Turner	Putnam	Stamford
Williams	Nassau	Princeton
Adams	Spring	Pittsfield
Johnson	Alma	Palo Alto
Glenn	Sand Hill	Woodside
Brooks	Senator	Brooklyn
Green	Walnut	Stamford

Figure 7.3 The *customer* relation.

loan-number	*branch-name*	*amount*
L-17	Downtown	1000
L-23	Redwood	2000
L-15	Perryridge	1500
L-14	Downtown	1500
L-93	Mianus	500
L-11	Round Hill	900
L-29	Pownal	1200
L-16	North Town	1300
L-18	Downtown	2000
L-25	Perryridge	2500
L-10	Brighton	2200

Figure 7.4 The *loan* relation.

can have streets with the same name. Thus, it is possible, at some time, to have an instance of the *customer* relation in which *customer-street* → *customer-city* is not satisfied. So, we would not include *customer-street* → *customer-city* in the set of functional dependencies that hold on *Customer-schema*.

In the *loan* relation (on *Loan-schema*) of Figure 7.4, we see that the dependency *loan-number* → *amount* is satisfied. In contrast to the case of *customer-city* and *customer-street* in *Customer-schema*, we do believe that the real-world enterprise that we are modeling requires each loan to have only one amount. Therefore, we want to require that *loan-number* → *amount* be satisfied by the *loan* relation at all times. In other words, we require that the constraint *loan-number* → *amount* hold on *Loan-schema*.

In the *branch* relation of Figure 7.5, we see that *branch-name* → *assets* is satisfied, as is *assets* → *branch-name*. We want to require that *branch-name* → *assets* hold on *Branch-schema*. However, we do not wish to require that *assets* → *branch-name* hold, since it is possible to have several branches that have the same asset value.

In what follows, we assume that, when we design a relational database, we first list those functional dependencies that must always hold. In the banking example, our list of dependencies includes the following:

branch-name	*branch-city*	*assets*
Downtown	Brooklyn	9000000
Redwood	Palo Alto	2100000
Perryridge	Horseneck	1700000
Mianus	Horseneck	400000
Round Hill	Horseneck	8000000
Pownal	Bennington	300000
North Town	Rye	3700000
Brighton	Brooklyn	7100000

Figure 7.5 The *branch* relation.

- On *Branch-schema*:

$$\begin{array}{l} \textit{branch-name} \rightarrow \textit{branch-city} \\ \textit{branch-name} \rightarrow \textit{assets} \end{array}$$

- On *Customer-schema*:

$$\begin{array}{l} \textit{customer-name} \rightarrow \textit{customer-city} \\ \textit{customer-name} \rightarrow \textit{customer-street} \end{array}$$

- On *Loan-schema*:

$$\begin{array}{l} \textit{loan-number} \rightarrow \textit{amount} \\ \textit{loan-number} \rightarrow \textit{branch-name} \end{array}$$

- On *Borrower-schema*:

No functional dependencies

- On *Account-schema*:

$$\begin{array}{l} \textit{account-number} \rightarrow \textit{branch-name} \\ \textit{account-number} \rightarrow \textit{balance} \end{array}$$

- On *Depositor-schema*:

No functional dependencies

7.3.2 Closure of a Set of Functional Dependencies

It is not sufficient to consider the given set of functional dependencies. Rather, we need to consider *all* functional dependencies that hold. We shall see that, given a set F of functional dependencies, we can prove that certain other functional dependencies hold. We say that such functional dependencies are “logically implied” by F.

More formally, given a relational schema R, a functional dependency f on R is **logically implied** by a set of functional dependencies F on R if every relation instance $r(R)$ that satisfies F also satisfies f.

Suppose we are given a relation schema $R = (A, B, C, G, H, I)$ and the set of functional dependencies

$$\begin{array}{l} A \rightarrow B \\ A \rightarrow C \\ CG \rightarrow H \\ CG \rightarrow I \\ B \rightarrow H \end{array}$$

The functional dependency

$$A \rightarrow H$$

is logically implied. That is, we can show that, whenever our given set of functional dependencies holds on a relation, $A \rightarrow H$ must also hold on the relation. Suppose that t_1 and t_2 are tuples such that

$$t_1[A] = t_2[A]$$

Since we are given that $A \rightarrow B$, it follows from the definition of functional dependency that

$$t_1[B] = t_2[B]$$

Then, since we are given that $B \rightarrow H$, it follows from the definition of functional dependency that

$$t_1[H] = t_2[H]$$

Therefore, we have shown that, whenever t_1 and t_2 are tuples such that $t_1[A] = t_2[A]$, it must be that $t_1[H] = t_2[H]$. But that is exactly the definition of $A \rightarrow H$.

Let *F* be a set of functional dependencies. The **closure** of *F*, denoted by F^+, is the set of all functional dependencies logically implied by *F*. Given *F*, we can compute F^+ directly from the formal definition of functional dependency. If *F* were large, this process would be lengthy and difficult. Such a computation of F^+ requires arguments of the type just used to show that $A \rightarrow H$ is in the closure of our example set of dependencies.

Axioms, or rules of inference, provide a simpler technique for reasoning about functional dependencies. In the rules that follow, we use Greek letters (α, β, γ, . . .) for sets of attributes, and uppercase Roman letters from the beginning of the alphabet for individual attributes. We use $\alpha\beta$ to denote $\alpha \cup \beta$.

We can use the following three rules to find logically implied functional dependencies. By applying these rules *repeatedly*, we can find all of F^+, given *F*. This collection of rules is called **Armstrong's axioms** in honor of the person who first proposed it.

- **Reflexivity rule**. If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- **Augmentation rule**. If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity rule**. If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Armstrong's axioms are **sound**, because they do not generate any incorrect functional dependencies. They are **complete**, because, for a given set *F* of functional dependencies, they allow us to generate all F^+. The bibliographical notes provide references for proofs of soundness and completeness.

Although Armstrong's axioms are complete, it is tiresome to use them directly for the computation of F^+. To simplify matters further, we list additional rules. It is possible to use Armstrong's axioms to prove that these rules are correct (see Exercises 7.8, 7.9, and 7.10).

- **Union rule**. If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition rule**. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds.
- **Pseudotransitivity rule**. If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds.

Let us apply our rules to the example of schema $R = (A, B, C, G, H, I)$ and the set F of functional dependencies $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. We list several members of F^+ here:

- $A \rightarrow H$. Since $A \rightarrow B$ and $B \rightarrow H$ hold, we apply the transitivity rule. Observe that it was much easier to use Armstrong's axioms to show that $A \rightarrow H$ holds than it was to argue directly from the definitions, as we did earlier in this section.
- $CG \rightarrow HI$. Since $CG \rightarrow H$ and $CG \rightarrow I$, the union rule implies that $CG \rightarrow HI$.
- $AG \rightarrow I$. Since $A \rightarrow C$ and $CG \rightarrow I$, the pseudotransitivity rule implies that $AG \rightarrow I$ holds.

  Another way of finding that $AG \rightarrow I$ holds is as follows. We use the augmentation rule on $A \rightarrow C$ to infer $AG \rightarrow CG$. Applying the transitivity rule to this dependency and $CG \rightarrow I$, we infer $AG \rightarrow I$.

Figure 7.6 shows a procedure that demonstrates formally how to use Armstrong's axioms to compute F^+. In this procedure, when a functional dependency is added to F^+, it may be already present, and in that case there is no change to F^+. We will also see an alternative way of computing F^+ in Section 7.3.3.

The left-hand and right-hand sides of a functional dependency are both subsets of R. Since a set of size n has 2^n subsets, there are a total of $2 \times 2^n = 2^{n+1}$ possible functional dependencies, where n is the number of attributes in R. Each iteration of the repeat loop of the procedure, except the last iteration, adds at least one functional dependency to F^+. Thus, the procedure is guaranteed to terminate.

7.3.3 Closure of Attribute Sets

To test whether a set α is a superkey, we must devise an algorithm for computing the set of attributes functionally determined by α. One way of doing this is to compute F^+, take all functional dependencies with α as the left-hand side, and take the union of the right-hand sides of all such dependencies. However, doing so can be expensive, since F^+ can be large.

```
$F^+$ = $F$
repeat
    for each functional dependency $f$ in $F^+$
        apply reflexivity and augmentation rules on $f$
        add the resulting functional dependencies to $F^+$
    for each pair of functional dependencies $f_1$ and $f_2$ in $F^+$
        if $f_1$ and $f_2$ can be combined using transitivity
            add the resulting functional dependency to $F^+$
until $F^+$ does not change any further
```

Figure 7.6 A procedure to compute F^+.

An efficient algorithm for computing the set of attributes functionally determined by α is useful not only for testing whether α is a superkey, but also for several other tasks, as we will see later in this section.

Let α be a set of attributes. We call the set of all attributes functionally determined by α under a set *F* of functional dependencies the **closure** of α under *F*; we denote it by α^+. Figure 7.7 shows an algorithm, written in pseudocode, to compute α^+. The input is a set *F* of functional dependencies and the set α of attributes. The output is stored in the variable *result*.

To illustrate how the algorithm works, we shall use it to compute $(AG)^+$ with the functional dependencies defined in Section 7.3.2. We start with $result = AG$. The first time that we execute the **while** loop to test each functional dependency, we find that

- $A \rightarrow B$ causes us to include *B* in *result*. To see this fact, we observe that $A \rightarrow B$ is in *F*, $A \subseteq result$ (which is *AG*), so $result := result \cup B$.
- $A \rightarrow C$ causes *result* to become *ABCG*.
- $CG \rightarrow H$ causes *result* to become *ABCGH*.
- $CG \rightarrow I$ causes *result* to become *ABCGHI*.

The second time that we execute the **while** loop, no new attributes are added to *result*, and the algorithm terminates.

Let us see why the algorithm of Figure 7.7 is correct. The first step is correct, since $\alpha \rightarrow \alpha$ always holds (by the reflexivity rule). We claim that, for any subset β of *result*, $\alpha \rightarrow \beta$. Since we start the **while** loop with $\alpha \rightarrow result$ being true, we can add γ to *result* only if $\beta \subseteq result$ and $\beta \rightarrow \gamma$. But then $result \rightarrow \beta$ by the reflexivity rule, so $\alpha \rightarrow \beta$ by transitivity. Another application of transitivity shows that $\alpha \rightarrow \gamma$ (using $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$). The union rule implies that $\alpha \rightarrow result \cup \gamma$, so α functionally determines any new result generated in the **while** loop. Thus, any attribute returned by the algorithm is in α^+.

It is easy to see that the algorithm finds all α^+. If there is an attribute in α^+ that is not yet in *result*, then there must be a functional dependency $\beta \rightarrow \gamma$ for which $\beta \subseteq result$, and at least one attribute in γ is not in *result*.

It turns out that, in the worst case, this algorithm may take an amount of time quadratic in the size of *F*. There is a faster (although slightly more complex) algorithm that runs in time linear in the size of *F*; that algorithm is presented as part of Exercise 7.14.

```
result := $\alpha$;
while (changes to result) do
        for each functional dependency $\beta \rightarrow \gamma$ in F do
            begin
                if $\beta \subseteq result$ then result := $result \cup \gamma$;
            end
```

Figure 7.7 An algorithm to compute α^+, the closure of α under *F*.

There are several uses of the attribute closure algorithm:

- To test if α is a superkey, we compute α^+, and check if α^+ contains all attributes of R.
- We can check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), by checking if $\beta \subseteq \alpha^+$. That is, we compute α^+ by using attribute closure, and then check if it contains β. This test is particularly useful, as we will see later in this chapter.
- It gives us an alternative way to compute F^+: For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

7.3.4 Canonical Cover

Suppose that we have a set of functional dependencies F on a relation schema. Whenever a user performs an update on the relation, the database system must ensure that the update does not violate any functional dependencies, that is, all the functional dependencies in F are satisfied in the new database state.

The system must roll back the update if it violates any functional dependencies in the set F.

We can reduce the effort spent in checking for violations by testing a simplified set of functional dependencies that has the same closure as the given set. Any database that satisfies the simplified set of functional dependencies will also satisfy the original set, and vice versa, since the two sets have the same closure. However, the simplified set is easier to test. We shall see how the simplified set can be constructed in a moment. First, we need some definitions.

An attribute of a functional dependency is said to be **extraneous** if we can remove it without changing the closure of the set of functional dependencies. The formal definition of **extraneous attributes** is as follows. Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.

- Attribute A is extraneous in α if $A \in \alpha$, and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
- Attribute A is extraneous in β if $A \in \beta$, and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.

For example, suppose we have the functional dependencies $AB \rightarrow C$ and $A \rightarrow C$ in F. Then, B is extraneous in $AB \rightarrow C$. As another example, suppose we have the functional dependencies $AB \rightarrow CD$ and $A \rightarrow C$ in F. Then C would be extraneous in the right-hand side of $AB \rightarrow CD$.

Beware of the direction of the implications when using the definition of extraneous attributes: If you exchange the left-hand side with right-hand side, the implication will *always* hold. That is, $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ always logically implies F, and also F always logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$

Here is how we can test efficiently if an attribute is extraneous. Let R be the relation schema, and let F be the given set of functional dependencies that hold on R. Consider an attribute A in a dependency $\alpha \rightarrow \beta$.

- If $A \in \beta$, to check if A is extraneous consider the set
  $$F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$$
  and check if $\alpha \rightarrow A$ can be inferred from F'. To do so, compute α^+ (the closure of α) under F'; if α^+ includes A, then A is extraneous in β.

- If $A \in \alpha$, to check if A is extraneous, let $\gamma = \alpha - \{A\}$, and check if $\gamma \rightarrow \beta$ can be inferred from F. To do so, compute γ^+ (the closure of γ) under F; if γ^+ includes all attributes in β, then A is extraneous in α.

For example, suppose F contains $AB \rightarrow CD$, $A \rightarrow E$, and $E \rightarrow C$. To check if C is extraneous in $AB \rightarrow CD$, we compute the attribute closure of AB under $F' = \{AB \rightarrow D, A \rightarrow E, \text{and } E \rightarrow C\}$. The closure is $ABCDE$, which includes CD, so we infer that C is extraneous.

A **canonical cover** F_c for F is a set of dependencies such that F logically implies all dependencies in F_c, and F_c logically implies all dependencies in F. Furthermore, F_c must have the following properties:

- No functional dependency in F_c contains an extraneous attribute.

- Each left side of a functional dependency in F_c is unique. That is, there are no two dependencies $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ in F_c such that $\alpha_1 = \alpha_2$.

A canonical cover for a set of functional dependencies F can be computed as depicted in Figure 7.8. It is important to note that when checking if an attribute is extraneous, the check uses the dependencies in the current value of F_c, and **not** the dependencies in F. If a functional dependency contains only one attribute in its right-hand side, for example $A \rightarrow C$, and that attribute is found to be extraneous, we would get a functional dependency with an empty right-hand side. Such functional dependencies should be deleted.

The canonical cover of F, F_c, can be shown to have the same closure as F; hence, testing whether F_c is satisfied is equivalent to testing whether F is satisfied. However, F_c is minimal in a certain sense—it does not contain extraneous attributes, and it

```
$F_c = F$
**repeat**
    Use the union rule to replace any dependencies in $F_c$ of the form
        $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$.
    Find a functional dependency $\alpha \rightarrow \beta$ in $F_c$ with an extraneous
        attribute either in $\alpha$ or in $\beta$.
        /* Note: the test for extraneous attributes is done using $F_c$, not $F$ */
    If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$.
**until** $F_c$ does not change.
```

Figure 7.8 Computing canonical cover

combines functional dependencies with the same left side. It is cheaper to test F_c than it is to test F itself.

Consider the following set F of functional dependencies on schema (A, B, C):

$$\begin{aligned}A &\rightarrow BC\\ B &\rightarrow C\\ A &\rightarrow B\\ AB &\rightarrow C\end{aligned}$$

Let us compute the canonical cover for F.

- There are two functional dependencies with the same set of attributes on the left side of the arrow:

  $$\begin{aligned}A &\rightarrow BC\\ A &\rightarrow B\end{aligned}$$

  We combine these functional dependencies into $A \rightarrow BC$.

- A is extraneous in $AB \rightarrow C$ because F logically implies $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$. This assertion is true because $B \rightarrow C$ is already in our set of functional dependencies.

- C is extraneous in $A \rightarrow BC$, since $A \rightarrow BC$ is logically implied by $A \rightarrow B$ and $B \rightarrow C$.

Thus, our canonical cover is

$$\begin{aligned}A &\rightarrow B\\ B &\rightarrow C\end{aligned}$$

Given a set F of functional dependencies, it may be that an entire functional dependency in the set is extraneous, in the sense that dropping it does not change the closure of F. We can show that a canonical cover F_c of F contains no such extraneous functional dependency. Suppose that, to the contrary, there were such an extraneous functional dependency in F_c. The right-side attributes of the dependency would then be extraneous, which is not possible by the definition of canonical covers.

A canonical cover might not be unique. For instance, consider the set of functional dependencies $F = \{A \rightarrow BC, B \rightarrow AC$, and $C \rightarrow AB\}$. If we apply the extraneity test to $A \rightarrow BC$, we find that both B and C are extraneous under F. However, it is incorrect to delete both! The algorithm for finding the canonical cover picks one of the two, and deletes it. Then,

1. If C is deleted, we get the set $F' = \{A \rightarrow B, B \rightarrow AC$, and $C \rightarrow AB\}$. Now, B is not extraneous in the righthand side of $A \rightarrow B$ under F'. Continuing the algorithm, we find A and B are extraneous in the right-hand side of $C \rightarrow AB$, leading to two canonical covers

   $$\begin{aligned}F_c &= \{A \rightarrow B, B \rightarrow C, \text{and } C \rightarrow A\}, \text{and}\\ F_c &= \{A \rightarrow B, B \rightarrow AC, \text{and } C \rightarrow B\}.\end{aligned}$$

2. If B is deleted, we get the set $\{A \rightarrow C, B \rightarrow AC$, and $C \rightarrow AB\}$. This case is symmetrical to the previous case, leading to the canonical covers

$$F_c = \{A \rightarrow C, C \rightarrow B, \text{and } B \rightarrow A\}, \text{and}$$
$$F_c = \{A \rightarrow C, B \rightarrow C, \text{and } C \rightarrow AB\}.$$

As an exercise, can you find one more canonical cover for F?

7.4 Decomposition

The bad design of Section 7.2 suggests that we should *decompose* a relation schema that has many attributes into several schemas with fewer attributes. Careless decomposition, however, may lead to another form of bad design.

Consider an alternative design in which we decompose *Lending-schema* into the following two schemas:

Branch-customer-schema = (*branch-name, branch-city, assets, customer-name*)
Customer-loan-schema = (*customer-name, loan-number, amount*)

Using the *lending* relation of Figure 7.1, we construct our new relations *branch-customer* (*Branch-customer*) and *customer-loan* (*Customer-loan-schema*):

$$branch\text{-}customer = \Pi_{branch\text{-}name,\ branch\text{-}city,\ assets,\ customer\text{-}name}\ (lending)$$
$$customer\text{-}loan = \Pi_{customer\text{-}name,\ loan\text{-}number,\ amount}\ (lending)$$

Figures 7.9 and 7.10, respectively, show the resulting *branch-customer* and *customer-name* relations.

Of course, there are cases in which we need to reconstruct the *loan* relation. For example, suppose that we wish to find all branches that have loans with amounts less than \$1000. No relation in our alternative database contains these data. We need to reconstruct the *lending* relation. It appears that we can do so by writing

$$branch\text{-}customer \bowtie customer\text{-}loan$$

branch-name	*branch-city*	*assets*	*customer-name*
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
Pownal	Bennington	300000	Williams
North Town	Rye	3700000	Hayes
Downtown	Brooklyn	9000000	Johnson
Perryridge	Horseneck	1700000	Glenn
Brighton	Brooklyn	7100000	Brooks

Figure 7.9 The relation *branch-customer*.

customer-name	*loan-number*	*amount*
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-93	500
Turner	L-11	900
Williams	L-29	1200
Hayes	L-16	1300
Johnson	L-18	2000
Glenn	L-25	2500
Brooks	L-10	2200

Figure 7.10 The relation *customer-loan*.

Figure 7.11 shows the result of computing $branch\text{-}customer \bowtie customer\text{-}loan$. When we compare this relation and the *lending* relation with which we started (Figure 7.1), we notice a difference: Although every tuple that appears in the *lending* relation appears in $branch\text{-}customer \bowtie customer\text{-}loan$, there are tuples in $branch\text{-}customer \bowtie customer\text{-}loan$ that are not in *lending*. In our example, *branch-customer* ⋈ *customer-loan* has the following additional tuples:

(Downtown, Brooklyn, 9000000, Jones, L-93, 500)
(Perryridge, Horseneck, 1700000, Hayes, L-16, 1300)
(Mianus, Horseneck, 400000, Jones, L-17, 1000)
(North Town, Rye, 3700000, Hayes, L-15, 1500)

Consider the query, "Find all bank branches that have made a loan in an amount less than $1000." If we look back at Figure 7.1, we see that the only branches with loan amounts less than $1000 are Mianus and Round Hill. However, when we apply the expression

$$\Pi_{branch\text{-}name}\ (\sigma_{amount\,<\,1000}\ (branch\text{-}customer \bowtie customer\text{-}loan))$$

we obtain *three* branch names: Mianus, Round Hill, and Downtown.

A closer examination of this example shows why. If a customer happens to have several loans from different branches, we cannot tell which loan belongs to which branch. Thus, when we join *branch-customer* and *customer-loan*, we obtain not only the tuples we had originally in *lending*, but also several additional tuples. Although we have *more* tuples in $branch\text{-}customer \bowtie customer\text{-}loan$, we actually have *less* information. We are no longer able, in general, to represent in the database information about which customers are borrowers from which branch. Because of this loss of information, we call the decomposition of *Lending-schema* into *Branch-customer-schema* and *customer-loan-schema* a **lossy decomposition**, or a **lossy-join decomposition**. A decomposition that is not a lossy-join decomposition is a **lossless-join decomposi-**

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Figure 7.11 The relation $branch\text{-}customer \bowtie customer\text{-}loan$.

tion. It should be clear from our example that a lossy-join decomposition is, in general, a bad database design.

Why is the decomposition lossy? There is one attribute in common between *Branch-customer-schema* and *Customer-loan-schema*:

$$\textit{Branch-customer-schema} \cap \textit{Customer-loan-schema} = \{\textit{customer-name}\}$$

The only way that we can represent a relationship between, for example, *loan-number* and *branch-name* is through *customer-name*. This representation is not adequate because a customer may have several loans, yet these loans are not necessarily obtained from the same branch.

Let us consider another alternative design, in which we decompose *Lending-schema* into the following two schemas:

$$\textit{Branch-schema} = (\textit{branch-name, branch-city, assets})$$
$$\textit{Loan-info-schema} = (\textit{branch-name, customer-name, loan-number, amount})$$

There is one attribute in common between these two schemas:

$$\textit{Branch-loan-schema} \cap \textit{Customer-loan-schema} = \{\textit{branch-name}\}$$

Thus, the only way that we can represent a relationship between, for example, *customer-name* and *assets* is through *branch-name*. The difference between this example and the preceding one is that the assets of a branch are the same, regardless of the customer to which we are referring, whereas the lending branch associated with a certain loan amount *does* depend on the customer to which we are referring. For a given *branch-name*, there is exactly one *assets* value and exactly one *branch-city*;

whereas a similar statement cannot be made for *customer-name*. That is, the functional dependency

$$\textit{branch-name} \rightarrow \textit{assets branch-city}$$

holds, but *customer-name* does not functionally determine *loan-number*.

The notion of lossless joins is central to much of relational-database design. Therefore, we restate the preceding examples more concisely and more formally. Let *R* be a relation schema. A set of relation schemas $\{R_1, R_2, \ldots, R_n\}$ is a **decomposition** of *R* if

$$R = R_1 \cup R_2 \cup \cdots \cup R_n$$

That is, $\{R_1, R_2, \ldots, R_n\}$ is a decomposition of *R* if, for $i = 1, 2, \ldots, n$, each R_i is a subset of *R*, and every attribute in *R* appears in at least one R_i.

Let *r* be a relation on schema *R*, and let $r_i = \Pi_{R_i}(r)$ for $i = 1, 2, \ldots, n$. That is, $\{r_1, r_2, \ldots, r_n\}$ is the database that results from decomposing *R* into $\{R_1, R_2, \ldots, R_n\}$. It is always the case that

$$r \subseteq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

To see that this assertion is true, consider a tuple *t* in relation *r*. When we compute the relations $r_1, r_2, \ldots, r_n$, the tuple *t* gives rise to one tuple t_i in each r_i, $i = 1, 2, \ldots, n$. These *n* tuples combine to regenerate *t* when we compute $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$. The details are left for you to complete as an exercise. Therefore, every tuple in *r* appears in $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$.

In general, $r \neq r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$. As an illustration, consider our earlier example, in which

- $n = 2$.
- R = *Lending-schema*.
- R_1 = *Branch-customer-schema*.
- R_2 = *Customer-loan-schema*.
- r = the relation shown in Figure 7.1.
- r_1 = the relation shown in Figure 7.9.
- r_2 = the relation shown in Figure 7.10.
- $r_1 \bowtie r_2$ = the relation shown in Figure 7.11.

Note that the relations in Figures 7.1 and 7.11 are not the same.

To have a lossless-join decomposition, we need to impose constraints on the set of possible relations. We found that the decomposition of *Lending-schema* into *Branch-schema* and *Loan-info-schema* is lossless because the functional dependency

$$\textit{branch-name} \rightarrow \textit{branch-city assets}$$

holds on *Branch-schema*.

Later in this chapter, we shall introduce constraints other than functional dependencies. We say that a relation is **legal** if it satisfies all rules, or constraints, that we impose on our database.

Let *C* represent a set of constraints on the database, and let R be a relation schema. A decomposition $\{R_1, R_2, \ldots, R_n\}$ of R is a **lossless-join decomposition** if, for all relations *r* on schema R that are legal under *C*,

$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie \cdots \bowtie \Pi_{R_n}(r)$$

We shall show how to test whether a decomposition is a lossless-join decomposition in the next few sections. A major part of this chapter deals with the questions of how to specify constraints on the database, and how to obtain lossless-join decompositions that avoid the pitfalls represented by the examples of bad database designs that we have seen in this section.

7.5 Desirable Properties of Decomposition

We can use a given set of functional dependencies in designing a relational database in which most of the undesirable properties discussed in Section 7.2 do not occur. When we design such systems, it may become necessary to decompose a relation into several smaller relations.

In this section, we outline the desirable properties of a decomposition of a relational schema. In later sections, we outline specific ways of decomposing a relational schema to get the properties we desire. We illustrate our concepts with the *Lending-schema* schema of Section 7.2:

Lending-schema = (*branch-name, branch-city, assets, customer-name, loan-number, amount*)

The set *F* of functional dependencies that we require to hold on *Lending-schema* are

branch-name → *branch-city assets*
loan-number → *amount branch-name*

As we discussed in Section 7.2, *Lending-schema* is an example of a bad database design. Assume that we decompose it to the following three relations:

Branch-schema = (*branch-name, branch-city, assets*)
Loan-schema = (*loan-number, branch-name, amount*)
Borrower-schema = (*customer-name, loan-number*)

We claim that this decomposition has several desirable properties, which we discuss next. Note that these three relation schemas are precisely the ones that we used previously, in Chapters 3 through 5.

7.5.1 Lossless-Join Decomposition

In Section 7.2, we argued that, when we decompose a relation into a number of smaller relations, it is crucial that the decomposition be lossless. We claim that the decomposition in Section 7.5 is indeed lossless. To demonstrate our claim, we must first present a criterion for determining whether a decomposition is lossy.

Let *R* be a relation schema, and let *F* be a set of functional dependencies on *R*. Let R_1 and R_2 form a decomposition of *R*. This decomposition is a lossless-join decomposition of *R* if at least one of the following functional dependencies is in F^+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

In other words, if $R_1 \cap R_2$ forms a superkey of either R_1 or R_2, the decomposition of R is a lossless-join decomposition. We can use attribute closure to efficiently test for superkeys, as we have seen earlier.

We now demonstrate that our decomposition of *Lending-schema* is a lossless-join decomposition by showing a sequence of steps that generate the decomposition. We begin by decomposing *Lending-schema* into two schemas:

Branch-schema = (*branch-name, branch-city, assets*)
Loan-info-schema = (*branch-name, customer-name, loan-number, amount*)

Since *branch-name* → *branch-city assets*, the augmentation rule for functional dependencies (Section 7.3.2) implies that

branch-name → *branch-name branch-city assets*

Since *Branch-schema* ∩ *Loan-info-schema* = {*branch-name*}, it follows that our initial decomposition is a lossless-join decomposition.

Next, we decompose *Loan-info-schema* into

Loan-schema = (*loan-number, branch-name, amount*)
Borrower-schema = (*customer-name, loan-number*)

This step results in a lossless-join decomposition, since *loan-number* is a common attribute and *loan-number* → *amount branch-name*.

For the general case of decomposition of a relation into multiple parts at once, the test for lossless join decomposition is more complicated. See the bibliographical notes for references on the topic.

While the test for binary decomposition is clearly a sufficient condition for lossless join, it is a necessary condition only if all constraints are functional dependencies. We shall see other types of constraints later (in particular, a type of constraint called multivalued dependencies), that can ensure that a decomposition is lossless join even if no functional dependencies are present.

7.5.2 Dependency Preservation

There is another goal in relational-database design: *dependency preservation*. When an update is made to the database, the system should be able to check that the update will not create an illegal relation—that is, one that does not satisfy all the given functional dependencies. If we are to check updates efficiently, we should design relational-database schemas that allow update validation without the computation of joins.

To decide whether joins must be computed to check an update, we need to determine what functional dependencies can be tested by checking each relation individually. Let F be a set of functional dependencies on a schema R, and let $R_1, R_2, \ldots, R_n$ be a decomposition of R. The **restriction** of F to R_i is the set F_i of all functional dependencies in F^+ that include *only* attributes of R_i. Since all functional dependencies in a restriction involve attributes of only one relation schema, it is possible to test such a dependency for satisfaction by checking only one relation.

Note that the definition of restriction uses all dependencies in F^+, not just those in F. For instance, suppose $F = \{A \rightarrow B, B \rightarrow C\}$, and we have a decomposition into AC and AB. The restriction of F to AC is then $A \rightarrow C$, since $A \rightarrow C$ is in F^+, even though it is not in F.

The set of restrictions $F_1, F_2, \ldots, F_n$ is the set of dependencies that can be checked efficiently. We now must ask whether testing only the restrictions is sufficient. Let $F' = F_1 \cup F_2 \cup \cdots \cup F_n$. F' is a set of functional dependencies on schema R, but, in general, $F' \neq F$. However, even if $F' \neq F$, it may be that $F'^+ = F^+$. If the latter is true, then every dependency in F is logically implied by F', and, if we verify that F' is satisfied, we have verified that F is satisfied. We say that a decomposition having the property $F'^+ = F^+$ is a **dependency-preserving decomposition**.

Figure 7.12 shows an algorithm for testing dependency preservation. The input is a set $D = \{R_1, R_2, \ldots, R_n\}$ of decomposed relation schemas, and a set F of functional dependencies. This algorithm is expensive since it requires computation of F^+; we will describe another algorithm that is more efficient after giving an example of testing for dependency preservation.

We can now show that our decomposition of *Lending-schema* is dependency preserving. Instead of applying the algorithm of Figure 7.12, we consider an easier alternative: We consider each member of the set F of functional dependencies that we

```
compute $F^+$;
**for each** schema $R_i$ in $D$ **do**
    **begin**
        $F_i :=$ the restriction of $F^+$ to $R_i$;
    **end**
$F' := \emptyset$
**for each** restriction $F_i$ **do**
    **begin**
        $F' = F' \cup F_i$
    **end**
compute $F'^+$;
**if** ($F'^+ = F^+$) **then** return (true)
                **else** return (false);
```

Figure 7.12 Testing for dependency preservation.

require to hold on *Lending-schema*, and show that each one can be tested in at least one relation in the decomposition.

- We can test the functional dependency: *branch-name* $\rightarrow$ *branch-city assets* using *Branch-schema* = (*branch-name*, *branch-city*, *assets*).
- We can test the functional dependency: *loan-number* $\rightarrow$ *amount branch-name* using *Loan-schema* = (*branch-name*, *loan-number*, *amount*).

If each member of *F* can be tested on one of the relations of the decomposition, then the decomposition is dependency preserving. However, there are cases where, even though the decomposition is dependency preserving, there is a dependency in *F* that cannot be tested in any one relation in the decomposition. The alternative test can therefore be used as a sufficient condition that is easy to check; if it fails we cannot conclude that the decomposition is not dependency preserving, instead we will have to apply the general test.

We now give a more efficient test for dependency preservation, which avoids computing F^+. The idea is to test each functional dependency $\alpha \rightarrow \beta$ in F by using a modified form of attribute closure to see if it is preserved by the decomposition. We apply the following procedure to each $\alpha \rightarrow \beta$ in F.

```
result = $\alpha$
while (changes to result) do
    for each $R_i$ in the decomposition
        t = $(result \cap R_i)^+ \cap R_i$
        result = $result \cup t$
```

The attribute closure is with respect to the functional dependencies in F. If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved. The decomposition is dependency preserving if and only if all the dependencies in F are preserved.

Note that instead of precomputing the restriction of F on R_i and using it for computing the attribute closure of *result*, we use attribute closure on (*result* $\cap R_i$) with respect to F, and then intersect it with R_i, to get an equivalent result. This procedure takes polynomial time, instead of the exponential time required to compute F^+.

7.5.3 Repetition of Information

The decomposition of *Lending-schema* does not suffer from the problem of repetition of information that we discussed in Section 7.2. In *Lending-schema*, it was necessary to repeat the city and assets of a branch for each loan. The decomposition separates branch and loan data into distinct relations, thereby eliminating this redundancy. Similarly, observe that, if a single loan is made to several customers, we must repeat the amount of the loan once for each customer (as well as the city and assets of the branch) in *lending-schema*. In the decomposition, the relation on schema *Borrower-schema* contains the *loan-number*, *customer-name* relationship, and no other schema does. Therefore, we have one tuple for each customer for a loan in only the relation on *Borrower-schema*. In the other relations involving *loan-number* (those on schemas *Loan-schema* and *Borrower-schema*), only one tuple per loan needs to appear.

Clearly, the lack of redundancy in our decomposition is desirable. The degree to which we can achieve this lack of redundancy is represented by several *normal forms*, which we shall discuss in the remainder of this chapter.

7.6 Boyce−Codd Normal Form

Using functional dependencies, we can define several *normal forms* that represent “good” database designs. In this section we cover BCNF (defined below), and later, in Section 7.7, we cover 3NF.

7.6.1 Definition

One of the more desirable normal forms that we can obtain is **Boyce−Codd normal form** (**BCNF**). A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (that is, $\beta \subseteq \alpha$).
- α is a superkey for schema *R*.

A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF.

As an illustration, consider the following relation schemas and their respective functional dependencies:

- *Customer-schema* = (*customer-name*, *customer-street*, *customer-city*)
  customer-name → *customer-street customer-city*
- *Branch-schema* = (*branch-name*, *assets*, *branch-city*)
  branch-name → *assets branch-city*
- *Loan-info-schema* = (*branch-name*, *customer-name*, *loan-number*, *amount*)
  loan-number → *amount branch-name*

We claim that *Customer-schema* is in BCNF. We note that a candidate key for the schema is *customer-name*. The only nontrivial functional dependencies that hold on *Customer-schema* have *customer-name* on the left side of the arrow. Since *customer-name* is a candidate key, functional dependencies with *customer-name* on the left side do not violate the definition of BCNF. Similarly, it can be shown easily that the relation schema *Branch-schema* is in BCNF.

The schema *Loan-info-schema*, however, is *not* in BCNF. First, note that *loan-number* is not a superkey for *Loan-info-schema*, since we *could* have a pair of tuples representing a single loan made to two people—for example,

(Downtown, John Bell, L-44, 1000)
(Downtown, Jane Bell, L-44, 1000)

Because we did not list functional dependencies that rule out the preceding case, *loan-number* is not a candidate key. However, the functional dependency *loan-number* $\rightarrow$ *amount* is nontrivial. Therefore, *Loan-info-schema* does not satisfy the definition of BCNF.

We claim that *Loan-info-schema* is not in a desirable form, since it suffers from the problem of *repetition of information* that we described in Section 7.2. We observe that, if there are several customer names associated with a loan, in a relation on *Loan-info-schema*, then we are forced to repeat the branch name and the amount once for each customer. We can eliminate this redundancy by redesigning our database such that all schemas are in BCNF. One approach to this problem is to take the existing non-BCNF design as a starting point, and to decompose those schemas that are not in BCNF. Consider the decomposition of *Loan-info-schema* into two schemas:

Loan-schema = (*loan-number, branch-name, amount*)
Borrower-schema = (*customer-name, loan-number*)

This decomposition is a lossless-join decomposition.

To determine whether these schemas are in BCNF, we need to determine what functional dependencies apply to them. In this example, it is easy to see that

loan-number $\rightarrow$ *amount branch-name*

applies to the *Loan-schema*, and that only trivial functional dependencies apply to *Borrower-schema*. Although *loan-number* is not a superkey for *Loan-info-schema*, it is a candidate key for *Loan-schema*. Thus, both schemas of our decomposition are in BCNF.

It is now possible to avoid redundancy in the case where there are several customers associated with a loan. There is exactly one tuple for each loan in the relation on *Loan-schema*, and one tuple for each customer of each loan in the relation on *Borrower-schema*. Thus, we do not have to repeat the branch name and the amount once for each customer associated with a loan.

Often testing of a relation to see if it satisfies BCNF can be simplified:

- To check if a nontrivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF, compute α^+ (the attribute closure of α), and verify that it includes all attributes of R; that is, it is a superkey of R.

- To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than check all dependencies in F^+.

  We can show that if none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.

Unfortunately, the latter procedure does not work when a relation is decomposed. That is, it *does not* suffice to use F when we test a relation R_i, in a decomposition of R, for violation of BCNF. For example, consider relation schema $R\ (A, B, C, D, E)$, with functional dependencies F containing $A \rightarrow B$ and $BC \rightarrow D$. Suppose this were decomposed into $R1(A, B)$ and $R2(A, C, D, E)$. Now, neither of the dependencies in F contains only attributes from (A, C, D, E) so we might be misled into thinking $R2$ satisfies BCNF. In fact, there is a dependency $AC \rightarrow D$ in F^+ (which can be inferred using the pseudotransitivity rule from the two dependencies in F), which shows that $R2$ is not in BCNF. Thus, we may need a dependency that is in F^+, but is not in F, to show that a decomposed relation is not in BCNF.

An alternative BCNF test is sometimes easier than computing every dependency in F^+. To check if a relation R_i in a decomposition of R is in BCNF, we apply this test:

- For every subset α of attributes in R_i, check that α^+ (the attribute closure of α under F) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i.

If the condition is violated by some set of attributes α in R_i, consider the following functional dependency, which can be shown to be present in F^+:

$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i.$$

The above dependency shows that R_i violates BCNF, and is a "witness" for the violation. The BCNF decomposition algorithm, which we shall see in Section 7.6.2, makes use of the witness.

7.6.2 Decomposition Algorithm

We are now able to state a general method to decompose a relation schema so as to satisfy BCNF. Figure 7.13 shows an algorithm for this task. If R is not in BCNF, we can decompose R into a collection of BCNF schemas $R_1, R_2, \ldots, R_n$ by the algorithm. The algorithm uses dependencies ("witnesses") that demonstrate violation of BCNF to perform the decomposition.

The decomposition that the algorithm generates is not only in BCNF, but is also a lossless-join decomposition. To see why our algorithm generates only lossless-join decompositions, we note that, when we replace a schema R_i with $(R_i - \beta)$ and (α, β), the dependency $\alpha \rightarrow \beta$ holds, and $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.

```
result := $\{R\}$;
done := false;
compute $F^+$;
while (not done) do
    if (there is a schema $R_i$ in result that is not in BCNF)
        then begin
                let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds
                on $R_i$ such that $\alpha \rightarrow R_i$ is not in $F^+$, and $\alpha \cap \beta = \emptyset$;
                result := (result $-$ $R_i$) $\cup$ $(R_i - \beta)$ $\cup$ $(\alpha, \beta)$;
            end
        else done := true;
```

Figure 7.13 BCNF decomposition algorithm.

We apply the BCNF decomposition algorithm to the *Lending-schema* schema that we used in Section 7.2 as an example of a poor database design:

Lending-schema = (*branch-name, branch-city, assets, customer-name, loan-number, amount*)

The set of functional dependencies that we require to hold on *Lending-schema* are

branch-name → *assets branch-city*
loan-number → *amount branch-name*

A candidate key for this schema is {*loan-number, customer-name*}.

We can apply the algorithm of Figure 7.13 to the *Lending-schema* example as follows:

- The functional dependency

  branch-name → *assets branch-city*

  holds on *Lending-schema*, but *branch-name* is not a superkey. Thus, *Lending-schema* is not in BCNF. We replace *Lending-schema* by

  Branch-schema = (*branch-name, branch-city, assets*)
  Loan-info-schema = (*branch-name, customer-name, loan-number, amount*)

- The only nontrivial functional dependencies that hold on *Branch-schema* include *branch-name* on the left side of the arrow. Since *branch-name* is a key for *Branch-schema*, the relation *Branch-schema* is in BCNF.

- The functional dependency

  loan-number → *amount branch-name*

  holds on *Loan-info-schema*, but *loan-number* is not a key for *Loan-info-schema*. We replace *Loan-info-schema* by

  Loan-schema = (*loan-number, branch-name, amount*)
  Borrower-schema = (*customer-name, loan-number*)

- *Loan-schema* and *Borrower-schema* are in BCNF.

Thus, the decomposition of *Lending-schema* results in the three relation schemas *Branch-schema*, *Loan-schema*, and *Borrower-schema*, each of which is in BCNF. These relation schemas are the same as those in Section 7.5, where we demonstrated that the resulting decomposition is both a lossless-join decomposition and a dependency-preserving decomposition.

The BCNF decomposition algorithm takes time exponential in the size of the initial schema, since the algorithm for checking if a relation in the decomposition satisfies BCNF can take exponential time. The bibliographical notes provide references to an algorithm that can compute a BCNF decomposition in polynomial time. However, the algorithm may "overnormalize," that is, decompose a relation unnecessarily.

7.6.3 Dependency Preservation

Not every BCNF decomposition is dependency preserving. As an illustration, consider the relation schema

$$\textit{Banker-schema} = (\textit{branch-name, customer-name, banker-name})$$

which indicates that a customer has a "personal banker" in a particular branch. The set *F* of functional dependencies that we require to hold on the *Banker-schema* is

$$\begin{array}{l} \textit{banker-name} \rightarrow \textit{branch-name} \\ \textit{branch-name customer-name} \rightarrow \textit{banker-name} \end{array}$$

Clearly, *Banker-schema* is not in BCNF since *banker-name* is not a superkey.

If we apply the algorithm of Figure 7.13, we obtain the following BCNF decomposition:

$$\begin{array}{l} \textit{Banker-branch-schema} = (\textit{banker-name}, \textit{branch-name}) \\ \textit{Customer-banker-schema} = (\textit{customer-name}, \textit{banker-name}) \end{array}$$

The decomposed schemas preserve only *banker-name* → *branch-name* (and trivial dependencies), but the closure of {*banker-name* → *branch-name*} does not include *customer-name branch-name* → *banker-name*. The violation of this dependency cannot be detected unless a join is computed.

To see why the decomposition of *Banker-schema* into the schemas *Banker-branch-schema* and *Customer-banker-schema* is not dependency preserving, we apply the algorithm of Figure 7.12. We find that the restrictions F_1 and F_2 of *F* to each schema are:

$$\begin{array}{l} F_1 = \{\textit{banker-name} \rightarrow \textit{branch-name}\} \\ F_2 = \emptyset \text{ (only trivial dependencies hold on } \textit{Customer-banker-schema}) \end{array}$$

(For brevity, we do not show trivial functional dependencies.) It is easy to see that the dependency *customer-name branch-name* → *banker-name* is not in $(F_1 \cup F_2)^+$ even though it *is* in F^+. Therefore, $(F_1 \cup F_2)^+ \neq F^+$, and the decomposition is not dependency preserving.

This example demonstrates that not every BCNF decomposition is dependency preserving. Moreover, it is easy to see that *any* BCNF decomposition of *Banker-schema* must fail to preserve *customer-name branch-name* → *banker-name*. Thus, the example shows that we cannot always satisfy all three design goals:

1. Lossless join
2. BCNF
3. Dependency preservation

Recall that lossless join is an essential condition for a decomposition, to avoid loss of information. We are therefore forced to give up either BCNF or dependency preservation. In Section 7.7 we present an alternative normal form, called **third normal form**, which is a small relaxation of BCNF; the motivation for using third normal form is that there is always a dependency preserving decomposition into third normal form.

There are situations where there is more than one way to decompose a schema into BCNF. Some of these decompositions may be dependency preserving, while others may not. For instance, suppose we have a relation schema $R(A, B, C)$ with the functional dependencies $A \rightarrow B$ and $B \rightarrow C$. From this set we can derive the further dependency $A \rightarrow C$. If we used the dependency $A \rightarrow B$ (or equivalently, $A \rightarrow C$) to decompose R, we would end up with two relations $R1(A, B)$ and $R2(A, C)$; the dependency $B \rightarrow C$ would not be preserved.

If instead we used the dependency $B \rightarrow C$ to decompose R, we would end up with two relations $R1(A, B)$ and $R2(B, C)$, which are in BCNF, and the decomposition is also dependency preserving. Clearly the decomposition into $R1(A, B)$ and $R2(B, C)$ is preferable. In general, the database designer should therefore look at alternative decompositions, and pick a dependency preserving decomposition where possible.

7.7 Third Normal Form

As we saw earlier, there are relational schemas where a BCNF decomposition cannot be dependency preserving. For such schemas, we have two alternatives if we wish to check if an update violates any functional dependencies:

- Pay the extra cost of computing joins to test for violations.
- Use an alternative decomposition, third normal form (3NF), which we present below, which makes testing of updates cheaper. Unlike BCNF, 3NF decompositions may contain some redundancy in the decomposed schema.

We shall see that it is always possible to find a lossless-join, dependency-preserving decomposition that is in 3NF. Which of the two alternatives to choose is a design decision to be made by the database designer on the basis of the application requirements.

7.7.1 Definition

BCNF requires that all nontrivial dependencies be of the form $\alpha \rightarrow \beta$, where α is a superkey. 3NF relaxes this constraint slightly by allowing nontrivial functional dependencies whose left side is not a superkey.

A relation schema R is in **third normal form (3NF)** with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- α is a superkey for R.

- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

Note that the third condition above does not say that a single candidate key should contain all the attributes in $\beta - \alpha$; each attribute A in $\beta - \alpha$ may be contained in a *different* candidate key.

The first two alternatives are the same as the two alternatives in the definition of BCNF. The third alternative of the 3NF definition seems rather unintuitive, and it is not obvious why it is useful. It represents, in some sense, a minimal relaxation of the BCNF conditions that helps ensure that every schema has a dependency-preserving decomposition into 3NF. Its purpose will become more clear later, when we study decomposition into 3NF.

Observe that any schema that satisfies BCNF also satisfies 3NF, since each of its functional dependencies would satisfy one of the first two alternatives. BCNF is therefore a more restrictive constraint than is 3NF.

The definition of 3NF allows certain functional dependencies that are not allowed in BCNF. A dependency $\alpha \rightarrow \beta$ that satisfies only the third alternative of the 3NF definition is not allowed in BCNF, but is allowed in 3NF.[1]

Let us return to our *Banker-schema* example (Section 7.6). We have shown that this relation schema does not have a dependency-preserving, lossless-join decomposition into BCNF. This schema, however, turns out to be in 3NF. To see that it is, we note that {*customer-name, branch-name*} is a candidate key for *Banker-schema*, so the only attribute not contained in a candidate key for *Banker-schema* is *banker-name*. The only nontrivial functional dependencies of the form

$$\alpha \rightarrow \textit{banker-name}$$

include {*customer-name, branch-name*} as part of α. Since {*customer-name, branch-name*} is a candidate key, these dependencies do not violate the definition of 3NF.

As an optimization when testing for 3NF, we can consider only functional dependencies in the given set F, rather than in F^+. Also, we can decompose the dependencies in F so that their right-hand side consists of only single attributes, and use the resultant set in place of F.

Given a dependency $\alpha \rightarrow \beta$, we can use the same attribute-closure–based technique that we used for BCNF to check if α is a superkey. If α is not a superkey, we have to verify whether each attribute in β is contained in a candidate key of R; this test is rather more expensive, since it involves finding candidate keys. In fact, testing for 3NF has been shown to be NP-hard; thus, it is very unlikely that there is a polynomial time complexity algorithm for the task.

7.7.2 Decomposition Algorithm

Figure 7.14 shows an algorithm for finding a dependency-preserving, lossless-join decomposition into 3NF. The set of dependencies F_c used in the algorithm is a canoni-

1. These dependencies are examples of **transitive dependencies** (see Exercise 7.25). The original definition of 3NF was in terms of transitive dependencies. The definition we use is equivalent but easier to understand.

```
let $F_c$ be a canonical cover for $F$;
$i := 0$;
**for each** functional dependency $\alpha \rightarrow \beta$ in $F_c$ **do**
   **if** none of the schemas $R_j, j = 1, 2, \ldots, i$ contains $\alpha\,\beta$
     **then begin**
            $i := i + 1$;
            $R_i := \alpha\,\beta$;
          **end**
**if** none of the schemas $R_j, j = 1, 2, \ldots, i$ contains a candidate key for $R$
   **then begin**
          $i := i + 1$;
          $R_i :=$ any candidate key for $R$;
     **end**
**return** $(R_1, R_2, \ldots, R_i)$
```

Figure 7.14 Dependency-preserving, lossless-join decomposition into 3NF.

cal cover for F. Note that the algorithm considers the set of schemas $R_j, j = 1, 2, \ldots, i$; initially $i = 0$, and in this case the set is empty.

To illustrate the algorithm of Figure 7.14, consider the following extension to the *Banker-schema* in Section 7.6:

Banker-info-schema = (*branch-name*, *customer-name*, *banker-name*, *office-number*)

The main difference here is that we include the banker's office number as part of the information. The functional dependencies for this relation schema are

banker-name → *branch-name office-number*
customer-name branch-name → *banker-name*

The **for** loop in the algorithm causes us to include the following schemas in our decomposition:

Banker-office-schema = (*banker-name*, *branch-name*, *office-number*)
Banker-schema = (*customer-name*, *branch-name*, *banker-name*)

Since *Banker-schema* contains a candidate key for *Banker-info-schema*, we are finished with the decomposition process.

The algorithm ensures the preservation of dependencies by explicitly building a schema for each dependency in a canonical cover. It ensures that the decomposition is a lossless-join decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed. Exercise 7.19 provides some insight into the proof that this suffices to guarantee a lossless join.

This algorithm is also called the **3NF synthesis algorithm**, since it takes a set of dependencies and adds one schema at a time, instead of decomposing the initial schema repeatedly. The result is not uniquely defined, since a set of functional dependencies can have more than one canonical cover, and, further, in some cases the result of the algorithm depends on the order in which it considers the dependencies in F_c.

If a relation R_i is in the decomposition generated by the synthesis algorithm, then R_i is in 3NF. Recall that when we test for 3NF, it suffices to consider functional dependencies whose right-hand side is a single attribute. Therefore, to see that R_i is in 3NF, you must convince yourself that any functional dependency $\gamma \rightarrow B$ that holds on R_i satisfies the definition of 3NF. Assume that the dependency that generated R_i in the synthesis algorithm is $\alpha \rightarrow \beta$. Now, B must be in α or β, since B is in R_i and $\alpha \rightarrow \beta$ generated R_i. Let us consider the three possible cases:

- B is in both α and β. In this case, the dependency $\alpha \rightarrow \beta$ would not have been in F_c since B would be extraneous in β. Thus, this case cannot hold.
- B is in β but not α. Consider two cases:
  - ☐ γ is a superkey. The second condition of 3NF is satisfied.
  - ☐ γ is not a superkey. Then α must contain some attribute not in γ. Now, since $\gamma \rightarrow B$ is in F^+, it must be derivable from F_c by using the attribute closure algorithm on γ. The derivation could not have used $\alpha \rightarrow \beta$— if it had been used, α must be contained in the attribute closure of γ, which is not possible, since we assumed γ is not a superkey. Now, using $\alpha \rightarrow (\beta - \{B\})$ and $\gamma \rightarrow B$, we can derive $\alpha \rightarrow B$ (since $\gamma \subseteq \alpha\beta$, and γ cannot contain B because $\gamma \rightarrow B$ is nontrivial). This would imply that B is extraneous in the right-hand side of $\alpha \rightarrow \beta$, which is not possible since $\alpha \rightarrow \beta$ is in the canonical cover F_c. Thus, if B is in β, then γ must be a superkey, and the second condition of 3NF must be satisfied.
- B is in α but not β.

  Since α is a candidate key, the third alternative in the definition of 3NF is satisfied.

Interestingly, the algorithm we described for decomposition into 3NF can be implemented in polynomial time, even though testing a given relation to see if it satisfies 3NF is NP-hard.

7.7.3 Comparison of BCNF and 3NF

Of the two normal forms for relational-database schemas, 3NF and BCNF, there are advantages to 3NF in that we know that it is always possible to obtain a 3NF design without sacrificing a lossless join or dependency preservation. Nevertheless, there are disadvantages to 3NF: If we do not eliminate all transitive relations schema dependencies, we may have to use null values to represent some of the possible meaningful relationships among data items, and there is the problem of repetition of information.

As an illustration of the null value problem, consider again the *Banker-schema* and its associated functional dependencies. Since *banker-name* $\rightarrow$ *branch-name*, we may want to represent relationships between values for *banker-name* and values for *branch-name* in our database. If we are to do so, however, either there must be a corresponding value for *customer-name*, or we must use a null value for the attribute *customer-name*.

customer-name	*banker-name*	*branch-name*
Jones	Johnson	Perryridge
Smith	Johnson	Perryridge
Hayes	Johnson	Perryridge
Jackson	Johnson	Perryridge
Curry	Johnson	Perryridge
Turner	Johnson	Perryridge

Figure 7.15 An instance of *Banker-schema*.

As an illustration of the repetition of information problem, consider the instance of *Banker-schema* in Figure 7.15. Notice that the information indicating that Johnson is working at the Perryridge branch is repeated.

Recall that our goals of database design with functional dependencies are:

1. BCNF
2. Lossless join
3. Dependency preservation

Since it is not always possible to satisfy all three, we may be forced to choose between BCNF and dependency preservation with 3NF.

It is worth noting that SQL does not provide a way of specifying functional dependencies, except for the special case of declaring superkeys by using the **primary key** or **unique** constraints. It is possible, although a little complicated, to write assertions that enforce a functional dependency (see Exercise 7.15); unfortunately, testing the assertions would be very expensive in most database systems. Thus even if we had a dependency-preserving decomposition, if we use standard SQL we would not be able to efficiently test a functional dependency whose left-hand side is not a key.

Although testing functional dependencies may involve a join if the decomposition is not dependency preserving, we can reduce the cost by using materialized views, which many database systems support. Given a BCNF decomposition that is not dependency preserving, we consider each dependency in a minimum cover F_c that is not preserved in the decomposition. For each such dependency $\alpha \rightarrow \beta$, we define a materialized view that computes a join of all relations in the decomposition, and projects the result on $\alpha\beta$. The functional dependency can be easily tested on the materialized view, by means of a constraint **unique** (α). On the negative side, there is a space and time overhead due to the materialized view, but on the positive side, the application programmer need not worry about writing code to keep redundant data consistent on updates; it is the job of the database system to maintain the materialized view, that is, keep up up to date when the database is updated. (Later in the book, in Section 14.5, we outline how a database system can perform materialized view maintenance efficiently.)

Thus, in case we are not able to get a dependency-preserving BCNF decomposition, it is generally preferable to opt for BCNF, and use techniques such as materialized views to reduce the cost of checking functional dependencies.

7.8 Fourth Normal Form

Some relation schemas, even though they are in BCNF, do not seem to be sufficiently normalized, in the sense that they still suffer from the problem of repetition of information. Consider again our banking example. Assume that, in an alternative design for the bank database schema, we have the schema

$$\textit{BC-schema} = (\textit{loan-number, customer-name, customer-street, customer-city})$$

The astute reader will recognize this schema as a non-BCNF schema because of the functional dependency

$$\textit{customer-name} \rightarrow \textit{customer-street customer-city}$$

that we asserted earlier, and because *customer-name* is not a key for *BC-schema*. However, assume that our bank is attracting wealthy customers who have several addresses (say, a winter home and a summer home). Then, we no longer wish to enforce the functional dependency *customer-name* → *customer-street customer-city*. If we remove this functional dependency, we find *BC-schema* to be in BCNF with respect to our modified set of functional dependencies. Yet, even though *BC-schema* is now in BCNF, we still have the problem of repetition of information that we had earlier.

To deal with this problem, we must define a new form of constraint, called a *multivalued dependency*. As we did for functional dependencies, we shall use multivalued dependencies to define a normal form for relation schemas. This normal form, called **fourth normal form** (4NF), is more restrictive than BCNF. We shall see that every 4NF schema is also in BCNF, but there are BCNF schemas that are not in 4NF.

7.8.1 Multivalued Dependencies

Functional dependencies rule out certain tuples from being in a relation. If $A \rightarrow B$, then we cannot have two tuples with the same A value but different B values. Multivalued dependencies, on the other hand, do not rule out the existence of certain tuples. Instead, they *require* that other tuples of a certain form be present in the relation. For this reason, functional dependencies sometimes are referred to as **equality-generating dependencies**, and multivalued dependencies are referred to as **tuple-generating dependencies**.

Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$$\alpha \twoheadrightarrow \beta$$

holds on R if, in any legal relation $r(R)$, for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that

$$\begin{aligned}
&t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]\\
&t_3[\beta] = t_1[\beta]\\
&t_3[R - \beta] = t_2[R - \beta]\\
&t_4[\beta] = t_2[\beta]\\
&t_4[R - \beta] = t_1[R - \beta]
\end{aligned}$$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

Figure 7.16 Tabular representation of $\alpha \twoheadrightarrow \beta$.

This definition is less complicated than it appears to be. Figure 7.16 gives a tabular picture of t_1, t_2, t_3, and t_4. Intuitively, the multivalued dependency $\alpha \twoheadrightarrow \beta$ says that the relationship between α and β is independent of the relationship between α and $R - \beta$. If the multivalued dependency $\alpha \twoheadrightarrow \beta$ is satisfied by all relations on schema R, then $\alpha \twoheadrightarrow \beta$ is a *trivial* multivalued dependency on schema R. Thus, $\alpha \twoheadrightarrow \beta$ is trivial if $\beta \subseteq \alpha$ or $\beta \cup \alpha = R$.

To illustrate the difference between functional and multivalued dependencies, we consider the *BC-schema* again, and the relation *bc* (*BC-schema*) of Figure 7.17. We must repeat the loan number once for each address a customer has, and we must repeat the address for each loan a customer has. This repetition is unnecessary, since the relationship between a customer and his address is independent of the relationship between that customer and a loan. If a customer (say, Smith) has a loan (say, loan number L-23), we want that loan to be associated with all Smith's addresses. Thus, the relation of Figure 7.18 is illegal. To make this relation legal, we need to add the tuples (L-23, Smith, Main, Manchester) and (L-27, Smith, North, Rye) to the *bc* relation of Figure 7.18.

Comparing the preceding example with our definition of multivalued dependency, we see that we want the multivalued dependency

$$\textit{customer-name} \twoheadrightarrow \textit{customer-street customer-city}$$

to hold. (The multivalued dependency *customer-name* $\twoheadrightarrow$ *loan-number* will do as well. We shall soon see that they are equivalent.)

As with functional dependencies, we shall use multivalued dependencies in two ways:

1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies

2. To specify constraints on the set of legal relations; we shall thus concern ourselves with *only* those relations that satisfy a given set of functional and multivalued dependencies

loan-number	*customer-name*	*customer-street*	*customer-city*
L-23	Smith	North	Rye
L-23	Smith	Main	Manchester
L-93	Curry	Lake	Horseneck

Figure 7.17 Relation *bc*: An example of redundancy in a BCNF relation.

loan-number	*customer-name*	*customer-street*	*customer-city*
L-23	Smith	North	Rye
L-27	Smith	Main	Manchester

Figure 7.18 An illegal *bc* relation.

Note that, if a relation r fails to satisfy a given multivalued dependency, we can construct a relation r' that *does* satisfy the multivalued dependency by adding tuples to r.

Let D denote a set of functional and multivalued dependencies. The **closure** D^+ of D is the set of all functional and multivalued dependencies logically implied by D. As we did for functional dependencies, we can compute D^+ from D, using the formal definitions of functional dependencies and multivalued dependencies. We can manage with such reasoning for very simple multivalued dependencies. Luckily, multivalued dependencies that occur in practice appear to be quite simple. For complex dependencies, it is better to reason about sets of dependencies by using a system of inference rules. (Section C.1.1 of the appendix outlines a system of inference rules for multivalued dependencies.)

From the definition of multivalued dependency, we can derive the following rule:

- If $\alpha \rightarrow \beta$, then $\alpha \rightarrow\!\!\!\rightarrow \beta$.

In other words, every functional dependency is also a multivalued dependency.

7.8.2 Definition of Fourth Normal Form

Consider again our *BC-schema* example in which the multivalued dependency *customer-name* $\rightarrow\!\!\!\rightarrow$ *customer-street customer-city* holds, but no nontrivial functional dependencies hold. We saw in the opening paragraphs of Section 7.8 that, although *BC-schema* is in BCNF, the design is not ideal, since we must repeat a customer's address information for each loan. We shall see that we can use the given multivalued dependency to improve the database design, by decomposing *BC-schema* into a **fourth normal form** decomposition.

A relation schema R is in **fourth normal form** (4NF) with respect to a set D of functional and multivalued dependencies if, for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\!\!\!\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds

- $\alpha \rightarrow\!\!\!\rightarrow \beta$ is a trivial multivalued dependency.
- α is a superkey for schema R.

A database design is in 4NF if each member of the set of relation schemas that constitutes the design is in 4NF.

Note that the definition of 4NF differs from the definition of BCNF in only the use of multivalued dependencies instead of functional dependencies. Every 4NF schema is in BCNF. To see this fact, we note that, if a schema R is not in BCNF, then there is

```
result := {R};
done := false;
compute $D^+$; Given schema $R_i$, let $D_i$ denote the restriction of $D^+$ to $R_i$
while (not done) do
    if (there is a schema $R_i$ in result that is not in 4NF w.r.t. $D_i$)
      then begin
                let $\alpha \rightarrow\!\!\!\rightarrow \beta$ be a nontrivial multivalued dependency that holds
                on $R_i$ such that $\alpha \rightarrow R_i$ is not in $D_i$, and $\alpha \cap \beta = \emptyset$;
                result := (result − $R_i$) $\cup$ ($R_i - \beta$) $\cup$ ($\alpha, \beta$);
            end
      else done := true;
```

Figure 7.19 4NF decomposition algorithm.

a nontrivial functional dependency $\alpha \rightarrow \beta$ holding on R, where α is not a superkey. Since $\alpha \rightarrow \beta$ implies $\alpha \rightarrow\!\!\!\rightarrow \beta$, R cannot be in 4NF.

Let R be a relation schema, and let $R_1, R_2, \ldots, R_n$ be a decomposition of R. To check if each relation schema R_i in the decomposition is in 4NF, we need to find what multivalued dependencies hold on each R_i. Recall that, for a set F of functional dependencies, the restriction F_i of F to R_i is all functional dependencies in F^+ that include *only* attributes of R_i. Now consider a set D of both functional and multivalued dependencies. The **restriction** of D to R_i is the set D_i consisting of

1. All functional dependencies in D^+ that include only attributes of R_i

2. All multivalued dependencies of the form

$$\alpha \rightarrow\!\!\!\rightarrow \beta \cap R_i$$

where $\alpha \subseteq R_i$ and $\alpha \rightarrow\!\!\!\rightarrow \beta$ is in D^+.

7.8.3 Decomposition Algorithm

The analogy between 4NF and BCNF applies to the algorithm for decomposing a schema into 4NF. Figure 7.19 shows the 4NF decomposition algorithm. It is identical to the BCNF decomposition algorithm of Figure 7.13, except that it uses multivalued, instead of functional, dependencies and uses the restriction of D^+ to R_i.

If we apply the algorithm of Figure 7.19 to *BC-schema*, we find that *customer-name* $\rightarrow\!\!\!\rightarrow$ *loan-number* is a nontrivial multivalued dependency, and *customer-name* is not a superkey for *BC-schema*. Following the algorithm, we replace *BC-schema* by two schemas:

Borrower-schema = (*customer-name*, *loan-number*)
Customer-schema = (*customer-name*, *customer-street*, *customer-city*).

This pair of schemas, which is in 4NF, eliminates the problem we encountered earlier with the redundancy of *BC-schema*.

As was the case when we were dealing solely with functional dependencies, we are interested in decompositions that are lossless-join decompositions and that preserve dependencies. The following fact about multivalued dependencies and lossless joins shows that the algorithm of Figure 7.19 generates only lossless-join decompositions:

- Let R be a relation schema, and let D be a set of functional and multivalued dependencies on R. Let R_1 and R_2 form a decomposition of R. This decomposition is a lossless-join decomposition of R if and only if at least one of the following multivalued dependencies is in D^+:

$$R_1 \cap R_2 \rightarrow\!\!\!\!\rightarrow R_1$$
$$R_1 \cap R_2 \rightarrow\!\!\!\!\rightarrow R_2$$

Recall that we stated in Section 7.5.1 that, if $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$, then R_1 and R_2 are a lossless-join decomposition of R. The preceding fact about multivalued dependencies is a more general statement about lossless joins. It says that, for *every* lossless-join decomposition of R into two schemas R_1 and R_2, one of the two dependencies $R_1 \cap R_2 \rightarrow\!\!\!\!\rightarrow R_1$ or $R_1 \cap R_2 \rightarrow\!\!\!\!\rightarrow R_2$ must hold.

The issue of dependency preservation when we decompose a relation becomes more complicated in the presence of multivalued dependencies. Section C.1.2 of the appendix pursues this topic.

7.9 More Normal Forms

The fourth normal form is by no means the "ultimate" normal form. As we saw earlier, multivalued dependencies help us understand and tackle some forms of repetition of information that cannot be understood in terms of functional dependencies. There are types of constraints called **join dependencies** that generalize multivalued dependencies, and lead to another normal form called **project-join normal form (PJNF)** (PJNF is called **fifth normal form** in some books). There is a class of even more general constraints, which leads to a normal form called **domain-key normal form**.

A practical problem with the use of these generalized constraints is that they are not only hard to reason with, but there is also no set of sound and complete inference rules for reasoning about the constraints. Hence PJNF and domain-key normal form are used quite rarely. Appendix C provides more details about these normal forms.

Conspicuous by its absence from our discussion of normal forms is **second normal form** (2NF). We have not discussed it, because it is of historical interest only. We simply define it, and let you experiment with it in Exercise 7.26.

7.10 Overall Database Design Process

So far we have looked at detailed issues about normal forms and normalization. In this section we study how normalization fits into the overall database design process.

Earlier in the chapter, starting in Section 7.4, we assumed that a relation schema R is given, and proceeded to normalize it. There are several ways in which we could have come up with the schema R:

1. R could have been generated when converting a E-R diagram to a set of tables.
2. R could have been a single relation containing *all* attributes that are of interest. The normalization process then breaks up R into smaller relations.
3. R could have been the result of some ad hoc design of relations, which we then test to verify that it satisfies a desired normal form.

In the rest of this section we examine the implications of these approaches. We also examine some practical issues in database design, including denormalization for performance and examples of bad design that are not detected by normalization.

7.10.1 E-R Model and Normalization

When we carefully define an E-R diagram, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization. However, there can be functional dependencies between attributes of an entity. For instance, suppose an *employee* entity had attributes *department-number* and *department-address*, and there is a functional dependency *department-number* → *department-address*. We would then need to normalize the relation generated from *employee*.

Most examples of such dependencies arise out of poor E-R diagram design. In the above example, if we did the E-R diagram correctly, we would have created a *department* entity with attribute *department-address* and a relationship between *employee* and *department*. Similarly, a relationship involving more than two entities may not be in a desirable normal form. Since most relationships are binary, such cases are relatively rare. (In fact, some E-R diagram variants actually make it difficult or impossible to specify nonbinary relations.)

Functional dependencies can help us detect poor E-R design. If the generated relations are not in desired normal form, the problem can be fixed in the E-R diagram. That is, normalization can be done formally as part of data modeling. Alternatively, normalization can be left to the designer's intuition during E-R modeling, and can be done formally on the relations generated from the E-R model.

7.10.2 The Universal Relation Approach

The second approach to database design is to start with a single relation schema containing all attributes of interest, and decompose it. One of our goals in choosing a decomposition was that it be a lossless-join decomposition. To consider losslessness, we assumed that it is valid to talk about the join of all the relations of the decomposed database.

Consider the database of Figure 7.20, showing a decomposition of the *loan-info* relation. The figure depicts a situation in which we have not yet determined the amount of loan L-58, but wish to record the remainder of the data on the loan. If we compute the natural join of these relations, we discover that all tuples referring to loan L-58 disappear. In other words, there is no *loan-info* relation corresponding to the relations of Figure 7.20. Tuples that disappear when we compute the join are *dangling tuples* (see Section 6.2.1). Formally, let $r_1(R_1),\ r_2(R_2), \ldots, r_n(R_n)$ be a set of relations. A

branch-name	*loan-number*
Round Hill	L-58

loan-number	*amount*

loan-number	*customer-name*
L-58	Johnson

Figure 7.20 Decomposition of *loan-info*.

tuple *t* of relation r_i is a dangling tuple if *t* is not in the relation

$$\Pi_{R_i}\ (r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n)$$

Dangling tuples may occur in practical database applications. They represent incomplete information, as they do in our example, where we wish to store data about a loan that is still in the process of being negotiated. The relation $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$ is called a **universal relation**, since it involves all the attributes in the universe defined by $R_1 \cup R_2 \cup \cdots \cup R_n$.

The only way that we can write a universal relation for the example of Figure 7.20 is to include *null values* in the universal relation. We saw in Chapter 3 that null values present several difficulties. Because of them, it may be better to view the relations of the decomposed design as representing the database, rather than as the universal relation whose schema we decomposed during the normalization process. (The bibliographical notes discuss research on null values and universal relations.)

Note that we cannot enter all incomplete information into the database of Figure 7.20 without resorting to null values. For example, we cannot enter a loan number unless we know at least one of the following:

- The customer name
- The branch name
- The amount of the loan

Thus, a particular decomposition defines a restricted form of incomplete information that is acceptable in our database.

The normal forms that we have defined generate good database designs from the point of view of representation of incomplete information. Returning again to the example of Figure 7.20, we would not want to allow storage of the following fact: "There is a loan (whose number is unknown) to Jones in the amount of $100." This is because

$$\textit{loan-number} \rightarrow \textit{customer-name amount}$$

and therefore the only way that we can relate *customer-name* and *amount* is through *loan-number*. If we do not know the loan number, we cannot distinguish this loan from other loans with unknown numbers.

In other words, we do not want to store data for which the key attributes are unknown. Observe that the normal forms that we have defined do not allow us to store that type of information unless we use null values. Thus, our normal forms allow representation of acceptable incomplete information via dangling tuples, while prohibiting the storage of undesirable incomplete information.

Another consequence of the universal relation approach to database design is that attribute names must be unique in the universal relation. We cannot use *name* to refer to both *customer-name* and to *branch-name*. It is generally preferable to use unique names, as we have done. Nevertheless, if we defined our relation schemas directly, rather than in terms of a universal relation, we could obtain relations on schemas such as the following for our banking example:

branch-loan (*name, number*)
loan-customer (*number, name*)
amt (*number, amount*)

Observe that, with the preceding relations, expressions such as *branch-loan* ⋈ *loan-customer* are meaningless. Indeed, the expression *branch-loan* ⋈ *loan-customer* finds loans made by branches to customers who have the same name as the name of the branch.

In a language such as SQL, however, a query involving *branch-loan* and *loan-customer* must remove ambiguity in references to *name* by prefixing the relation name. In such environments, the multiple roles for *name* (as branch name and as customer name) are less troublesome and may be simpler to use.

We believe that using the **unique-role assumption**—that each attribute name has a unique meaning in the database—is generally preferable to reusing of the same name in multiple roles. When the unique-role assumption is not made, the database designer must be especially careful when constructing a normalized relational-database design.

7.10.3 Denormalization for Performance

Occasionally database designers choose a schema that has redundant information; that is, it is not normalized. They use the redundancy to improve performance for specific applications. The penalty paid for not using a normalized schema is the extra work (in terms of coding time and execution time) to keep redundant data consistent.

For instance, suppose that the name of an account holder has to be displayed along with the account number and balance, every time the account is accessed. In our normalized schema, this requires a join of *account* with *depositor*.

One alternative to computing the join on the fly is to store a relation containing all the attributes of *account* and *depositor*. This makes displaying the account information faster. However, the balance information for an account is repeated for every person who owns the account, and all copies must be updated by the application, whenever the account balance is updated. The process of taking a normalized schema and making it non-normalized is called **denormalization**, and designers use it to tune performance of systems to support time-critical operations.

A better alternative, supported by many database systems today, is to use the normalized schema, and additionally store the join or *account* and *depositor* as a materialized view. (Recall that a materialized view is a view whose result is stored in the database, and brought up to date when the relations used in the view are updated.) Like denormalization, using materialized view does have space and time overheads; however, it has the advantage that keeping the view up to date is the job of the database system, not the application programmer.

7.10.4 Other Design Issues

There are some aspects of database design that are not addressed by normalization, and can thus lead to bad database design. We give examples here; obviously, such designs should be avoided.

Consider a company database, where we want to store earnings of companies in different years. A relation *earnings*(*company-id, year, amount*) could be used to store the earnings information. The only functional dependency on this relation is *company-id, year* → *amount*, and the relation is in BCNF.

An alternative design is to use multiple relations, each storing the earnings for a different year. Let us say the years of interest are 2000, 2001, and 2002; we would then have relations of the form *earnings-2000*, *earnings-2001*, *earnings-2002*, all of which are on the schema (*company-id, earnings*). The only functional dependency here on each relation would be *company-id* → *earnings*, so these relations are also in BCNF.

However, this alternative design is clearly a bad idea—we would have to create a new relation every year, and would also have to write new queries every year, to take each new relation into account. Queries would also be more complicated since they may have to refer to many relations.

Yet another way of representing the same data is to have a single relation *company-year*(*company-id, earnings-2000, earnings-2001, earnings-2002*). Here the only functional dependencies are from *company-id* to the other attributes, and again the relation is in BCNF. This design is also a bad idea since it has problems similar to the previous design—namely we would have to modify the relation schema and write new queries, every year. Queries would also be more complicated, since they may have to refer to many attributes.

Representations such as those in the *company-year* relation, with one column for each value of an attribute, are called **crosstabs**; they are widely used in spreadsheets and reports and in data analysis tools. While such representations are useful for display to users, for the reasons just given, they are not desirable in a database design. SQL extensions have been proposed to convert data from a normal relational representation to a crosstab, for display.

7.11 Summary

- We showed pitfalls in database design, and how to systematically design a database schema that avoids the pitfalls. The pitfalls included repeated information and inability to represent some information.

- We introduced the concept of functional dependencies, and showed how to reason with functional dependencies. We laid special emphasis on what dependencies are logically implied by a set of dependencies. We also defined the notion of a canonical cover, which is a minimal set of functional dependencies equivalent to a given set of functional dependencies.
- We introduced the concept of decomposition, and showed that decompositions must be lossless-join decompositions, and should preferably be dependency preserving.
- If the decomposition is dependency preserving, given a database update, all functional dependencies can be verifiable from individual relations, without computing a join of relations in the decomposition.
- We then presented Boyce−Codd Normal Form (BCNF); relations in BCNF are free from the pitfalls outlined earlier. We outlined an algorithm for decomposing relations into BCNF. There are relations for which there is no dependency-preserving BCNF decomposition.
- We used the canonical covers to decompose a relation into 3NF, which is a small relaxation of the BCNF condition. Relations in 3NF may have some redundancy, but there is always a dependency-preserving decomposition into 3NF.
- We presented the notion of multivalued dependencies, which specify constraints that cannot be specified with functional dependencies alone. We defined fourth normal form (4NF) with multivalued dependencies. Section C.1.1 of the appendix gives details on reasoning about multivalued dependencies.
- Other normal forms, such as PJNF and DKNF, eliminate more subtle forms of redundancy. However, these are hard to work with and are rarely used. Appendix C gives details on these normal forms.
- In reviewing the issues in this chapter, note that the reason we could define rigorous approaches to relational-database design is that the relational data model rests on a firm mathematical foundation. That is one of the primary advantages of the relational model compared with the other data models that we have studied.

Review Terms

- Atomic domains
- First normal form
- Pitfalls in relational-database design
- Functional dependencies
- Superkey
- F holds on R
- R satisfies F
- Trivial functional dependencies
- Closure of a set of functional dependencies
- Armstrong's axioms

- Closure of attribute sets
- Decomposition
- Lossless-join decomposition
- Legal relations
- Dependency preservation
- Restriction of F to R_i
- Boyce−Codd normal form (BCNF)
- BCNF decomposition algorithm
- Canonical cover
- Extraneous attributes
- Third normal form
- 3NF decomposition algorithm
- Multivalued dependencies
- Fourth normal form
- restriction of a multivalued dependency
- Project-join normal form (PJNF)
- Domain-key normal form
- E-R model and normalization
- Universal relation
- Unique-role assumption
- Denormalization

Exercises

7.1 Explain what is meant by *repetition of information* and *inability to represent information*. Explain why each of these properties may indicate a bad relational-database design.

7.2 Suppose that we decompose the schema $R = (A, B, C, D, E)$ into

$$(A, B, C)$$
$$(A, D, E)$$

Show that this decomposition is a lossless-join decomposition if the following set F of functional dependencies holds:

$$A \rightarrow BC$$
$$CD \rightarrow E$$
$$B \rightarrow D$$
$$E \rightarrow A$$

7.3 Why are certain functional dependencies called *trivial* functional dependencies?

7.4 List all functional dependencies satisfied by the relation of Figure 7.21.

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Figure 7.21 Relation of Exercise 7.4.

7.5 Use the definition of functional dependency to argue that each of Armstrong's axioms (reflexivity, augmentation, and transitivity) is sound.

7.6 Explain how functional dependencies can be used to indicate the following:

- A one-to-one relationship set exists between entity sets *account* and *customer*.
- A many-to-one relationship set exists between entity sets *account* and *customer*.

7.7 Consider the following proposed rule for functional dependencies: If $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, then $\alpha \rightarrow \gamma$. Prove that this rule is *not* sound by showing a relation *r* that satisfies $\alpha \rightarrow \beta$ and $\gamma \rightarrow \beta$, but does not satisfy $\alpha \rightarrow \gamma$.

7.8 Use Armstrong's axioms to prove the soundness of the union rule. (*Hint*: Use the augmentation rule to show that, if $\alpha \rightarrow \beta$, then $\alpha \rightarrow \alpha\beta$. Apply the augmentation rule again, using $\alpha \rightarrow \gamma$, and then apply the transitivity rule.)

7.9 Use Armstrong's axioms to prove the soundness of the decomposition rule.

7.10 Use Armstrong's axioms to prove the soundness of the pseudotransitivity rule.

7.11 Compute the closure of the following set *F* of functional dependencies for relation schema $R = (A, B, C, D, E)$.

$$\begin{aligned} &A \rightarrow BC\\ &CD \rightarrow E\\ &B \rightarrow D\\ &E \rightarrow A \end{aligned}$$

List the candidate keys for *R*.

7.12 Using the functional dependencies of Exercise 7.11, compute B^+.

7.13 Using the functional dependencies of Exercise 7.11, compute the canonical cover F_c.

7.14 Consider the algorithm in Figure 7.22 to compute α^+. Show that this algorithm is more efficient than the one presented in Figure 7.7 (Section 7.3.3) and that it computes α^+ correctly.

7.15 Given the database schema $R(a, b, c)$, and a relation r on the schema R, write an SQL query to test whether the functional dependency $b \rightarrow c$ holds on relation r. Also write an SQL assertion that enforces the functional dependency. Assume that no null values are present.

7.16 Show that the following decomposition of the schema *R* of Exercise 7.2 is not a lossless-join decomposition:

$$(A, B, C)$$
$$(C, D, E)$$

Hint: Give an example of a relation *r* on schema *R* such that

$$\Pi_{A,\,B,\,C}\,(r) \;\bowtie\; \Pi_{C,\,D,\,E}\,(r) \;\neq\; r$$

```
$result := \emptyset$;
/* fdcount is an array whose ith element contains the number
   of attributes on the left side of the ith FD that are
   not yet known to be in $\alpha^+$ */
for $i := 1$ to $|F|$ do
   begin
      let $\beta \rightarrow \gamma$ denote the ith FD;
      fdcount [i] $:= |\beta|$;
   end
/* appears is an array with one entry for each attribute. The
   entry for attribute A is a list of integers. Each integer
   i on the list indicates that A appears on the left side
   of the ith FD */
for each attribute A do
   begin
      appears [A] $:= NIL$;
      for $i := 1$ to $|F|$ do
         begin
            let $\beta \rightarrow \gamma$ denote the ith FD;
            if $A \in \beta$ then add i to appears [A];
         end
   end
addin $(\alpha)$;
return (result);

procedure addin $(\alpha)$;
for each attribute A in $\alpha$ do
   begin
      if $A \notin result$ then
         begin
            $result := result \cup \{A\}$;
            for each element i of appears[A] do
               begin
                  fdcount [i] := fdcount [i] $- 1$;
                  if fdcount [i] $:= 0$ then
                     begin
                        let $\beta \rightarrow \gamma$ denote the ith FD;
                        addin $(\gamma)$;
                     end
               end
         end
   end
```

Figure 7.22 An algorithm to compute α^+.

7.17 Let $R_1,\ R_2, \ldots, R_n$ be a decomposition of schema *U*. Let $u(U)$ be a relation, and let $r_i \ = \ \Pi_{R_I}(u)$. Show that

$$u \ \subseteq \ r_1 \ \bowtie \ r_2 \ \bowtie \ \cdots \ \bowtie \ r_n$$

7.18 Show that the decomposition in Exercise 7.2 is not a dependency-preserving decomposition.

7.19 Show that it is possible to ensure that a dependency-preserving decomposition into 3NF is a lossless-join decomposition by guaranteeing that at least one schema contains a candidate key for the schema being decomposed. (*Hint*: Show that the join of all the projections onto the schemas of the decomposition cannot have more tuples than the original relation.)

7.20 List the three design goals for relational databases, and explain why each is desirable.

7.21 Give a lossless-join decomposition into BCNF of schema *R* of Exercise 7.2.

7.22 Give an example of a relation schema R' and set F' of functional dependencies such that there are at least three distinct lossless-join decompositions of R' into BCNF.

7.23 In designing a relational database, why might we choose a non-BCNF design?

7.24 Give a lossless-join, dependency-preserving decomposition into 3NF of schema *R* of Exercise 7.2.

7.25 Let a *prime* attribute be one that appears in at least one candidate key. Let α and β be sets of attributes such that $\alpha \rightarrow \beta$ holds, but $\beta \rightarrow \alpha$ does not hold. Let *A* be an attribute that is not in α, is not in β, and for which $\beta \rightarrow A$ holds. We say that *A* is *transitively dependent* on α. We can restate our definition of 3NF as follows: A relation schema *R* is in 3NF with respect to a set *F* of functional dependencies if there are no nonprime attributes *A* in *R* for which *A* is transitively dependent on a key for *R*.

Show that this new definition is equivalent to the original one.

7.26 A functional dependency $\alpha \ \rightarrow \ \beta$ is called a **partial dependency** if there is a proper subset γ of α such that $\gamma \rightarrow \beta$. We say that β is *partially dependent* on α. A relation schema *R* is in **second normal form** (2NF) if each attribute *A* in *R* meets one of the following criteria:

- It appears in a candidate key.
- It is not partially dependent on a candidate key.

Show that every 3NF schema is in 2NF. (*Hint*: Show that every partial dependency is a transitive dependency.)

7.27 Given the three goals of relational-database design, is there any reason to design a database schema that is in 2NF, but is in no higher-order normal form? (See Exercise 7.26 for the definition of 2NF.)

7.28 Give an example of a relation schema *R* and a set of dependencies such that *R* is in BCNF, but is not in 4NF.

7.29 Explain why 4NF is a normal form more desirable than BCNF.

7.30 Explain how dangling tuples may arise. Explain problems that they may cause.

Bibliographical Notes

The first discussion of relational-database design theory appeared in an early paper by Codd [1970]. In that paper, Codd also introduced functional dependencies, and first, second, and third normal forms.

Armstrong's axioms were introduced in Armstrong [1974]. Ullman [1988] is an easily accessible source of proofs of soundness and completeness of Armstrong's axioms. Ullman [1988] also provides an algorithm for testing for lossless-join decomposition for general (nonbinary) decompositions, and many other algorithms, theorems, and proofs concerning dependency theory. Maier [1983] discusses the theory of functional dependencies.Graham et al. [1986] discusses formal aspects of the concept of a legal relation.

BCNF was introduced in Codd [1972]. The desirability of BCNF is discussed in Bernstein et al. [1980a]. A polynomial-time algorithm for BCNF decomposition appears in Tsou and Fischer [1982], and can also be found in Ullman [1988]. Biskup et al. [1979] gives the algorithm we used to find a lossless-join dependency-preserving decomposition into 3NF. Fundamental results on the lossless-join property appear in Aho et al. [1979a].

Multivalued dependencies are discussed in Zaniolo [1976]. Beeri et al. [1977] gives a set of axioms for multivalued dependencies, and proves that the authors axioms are sound and complete. Our axiomatization is based on theirs. The notions of 4NF, PJNF, and DKNF are from Fagin [1977], Fagin [1979], and Fagin [1981], respectively.

Maier [1983] presents the design theory of relational databases in detail. Ullman [1988] and Abiteboul et al. [1995] present a more theoretic coverage of many of the dependencies and normal forms presented here. See the bibliographical notes of Appendix C for further references to literature on normalization.