CHAPTER 27

Formal-Relational Query Languages

In Chapter 2 we introduced the relational model and presented the relational algebra (RA), which forms the basis of the widely used SQL query language. In this chapter we continue with our coverage of "pure" query languages. In particular, we cover the tuple relational calculus and the domain relational calculus, which are declarative query languages based on mathematical logic. We also cover Datalog, which has a syntax modeled after the Prolog language. Although not used commercially at present, Datalog has been used in several research database systems. For Datalog, 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.

27.1 The Tuple Relational Calculus

When we write a relational-algebra expression, we provide a sequence of procedures that generates the answer to our query. The **tuple relational calculus**, by contrast, is a **nonprocedural** query language. It describes the desired information without giving a specific procedure for obtaining that information.

A query in the tuple relational calculus is expressed as:

$$\{t \mid P(t)\}$$

That is, it is the set of all tuples t such that predicate P is true for t. Following our earlier notation, we use $t[A]$ to denote the value of tuple t on attribute A, and we use $t \in r$ to denote that tuple t is in relation r.

Before we give a formal definition of the tuple relational calculus, we return to some of the queries for which we wrote relational-algebra expressions in Section 2.6.

27.1.1 Example Queries

Find the *ID*, *name*, *dept_name*, *salary* for instructors whose salary is greater than $80,000:

$$\{t \mid t \in instructor \wedge t[salary] > 80000\}$$

Suppose that we want only the *ID* attribute, rather than all attributes of the *instructor* relation. To write this query in the tuple relational calculus, we need to write an expression for a relation on the schema (*ID*). We need those tuples on (*ID*) such that there is a tuple in *instructor* with the *salary* attribute > 80000. To express this request, we need the construct "there exists" from mathematical logic. The notation:

$$\exists\, t \in r\, (Q(t))$$

means "there exists a tuple t in relation r such that predicate $Q(t)$ is true."

Using this notation, we can write the query "Find the instructor *ID* for each instructor with a salary greater than $80,000" as:

$$\{t \mid \exists\, s \in instructor\, (t[ID] = s[ID] \wedge s[salary] > 80000)\}$$

In English, we read the preceding expression as "The set of all tuples t such that there exists a tuple s in relation *instructor* for which the values of t and s for the *ID* attribute are equal, and the value of s for the *salary* attribute is greater than $80,000."

Tuple variable t is defined on only the *ID* attribute, since that is the only attribute having a condition specified for t. Thus, the result is a relation on (*ID*).

Consider the query "Find the names of all instructors whose department is in the Watson building." This query is slightly more complex than the previous queries, since it involves two relations: *instructor* and *department*. As we shall see, however, all it requires is that we have two "there exists" clauses in our tuple-relational-calculus expression, connected by *and* ($\wedge$). We write the query as follows:

$$\{t \mid \exists\, s \in instructor\, (t[name] = s[name] \wedge \exists\, u \in department\, (u[dept\_name] = s[dept\_name] \wedge u[building] = \text{“Watson”}))\}$$

Tuple variable u is restricted to departments that are located in the Watson building, while tuple variable s is restricted to instructors whose *dept_name* matches that of tuple variable u. Figure 27.1 shows the result of this query.

To find the set of all courses taught in the Fall 2017 semester, the Spring 2018 semester, or both, we used the union operation in the relational algebra. In the tuple relational calculus, we shall need two "there exists" clauses, connected by *or* ($\vee$):

name
Einstein
Crick
Gold

Figure 27.1 Names of all instructors whose department is in the Watson building.

$$\{t \mid \exists\, s \in section\ (t[course\_id] = s[course\_id]) \\ \wedge s[semester] = \text{“Fall”} \wedge s[year] = 2017) \\ \vee \exists\, u \in section\ (u[course\_id] = t[course\_id]) \\ \wedge u[semester] = \text{“Spring”} \wedge u[year] = 2018)\}$$

This expression gives us the set of all *course_id* tuples for which at least one of the following holds:

- The *course_id* appears in some tuple of the *section* relation with *semester* = Fall and *year* = 2017.
- The *course_id* appears in some tuple of the *section* relation with *semester* = Spring and *year* = 2018.

If the same course is offered in both the Fall 2017 and Spring 2018 semesters, its *course_id* appears only once in the result, because the mathematical definition of a set does not allow duplicate members. The result of this query is shown in Figure 27.2.

If we now want *only* those *course_id* values for courses that are offered in *both* the Fall 2017 and Spring 2018 semesters, all we need to do is to change the *or* ($\vee$) to *and* ($\wedge$) in the preceding expression.

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

Figure 27.2 Courses offered in either Fall 2017, Spring 2018, or both semesters.

course_id
CS-101

Figure 27.3 Courses offered in both the Fall 2017 and Spring 2018 semesters.

$$\{t \mid \exists\, s \in section\ (t[course\_id] = s[course\_id]) \\ \wedge\, s[semester] = \text{“Fall”} \wedge\, s[year] = 2017) \\ \wedge \exists\, u \in section\ (u[course\_id] = t[course\_id]) \\ \wedge\, u[semester] = \text{“Spring”} \wedge\, u[year] = 2018)\}$$

The result of this query appears in Figure 27.3.

Now consider the query “Find all the courses taught in the Fall 2017 semester but not in Spring 2018 semester.” The tuple-relational-calculus expression for this query is similar to the expressions that we have just seen, except for the use of the *not* ($\neg$) symbol:

$$\{t \mid \exists\, s \in section\ (t[course\_id] = s[course\_id]) \\ \wedge\, s[semester] = \text{“Fall”} \wedge\, s[year] = 2017) \\ \wedge \neg \exists\, u \in section\ (u[course\_id] = t[course\_id]) \\ \wedge\, u[semester] = \text{“Spring”} \wedge\, u[year] = 2018)\}$$

This tuple-relational-calculus expression uses the $\exists s \in section\ (\ldots)$ clause to require that a particular *course_id* is taught in the Fall 2017 semester, and it uses the $\neg\, \exists\, u \in section\ (\ldots)$ clause to eliminate those *course_id* values that appear in some tuple of the *section* relation as having been taught in the Spring 2018 semester.

The query that we shall consider next uses implication, denoted by $\Rightarrow$. The formula $P \Rightarrow Q$ means “P implies Q”; that is, “if P is true, then Q must be true.” Note that $P \Rightarrow Q$ is logically equivalent to $\neg P \vee Q$. The use of implication rather than *not* and *or* often suggests a more intuitive interpretation of a query in English.

Consider the query that “Find all students who have taken all courses offered in the Biology department.” To write this query in the tuple relational calculus, we introduce the “for all” construct, denoted by $\forall$. The notation:

$$\forall\, t \in r\ (Q(t))$$

means “Q is true for all tuples t in relation r.”

We write the expression for our query as follows:

$$\{t \mid \exists\, r \in student\ (r[ID] = t[ID]) \wedge \\ (\forall\, u \in course\ (u[dept\_name] = \text{“ Biology”} \Rightarrow \\ \exists\, s \in takes\ (t[ID] = s[ID] \\ \wedge\, s[course\_id] = u[course\_id]))\}$$

In English, we interpret this expression as "The set of all students (i.e., (*ID*) tuples t) such that, for *all* tuples u in the *course* relation, if the value of u on attribute *dept_name* is 'Biology', then there exists a tuple in the *takes* relation that includes the student *ID* and the *course_id*."

Note that there is a subtlety in the preceding query: If there is no course offered in the Biology department, all student *ID*s satisfy the condition. The first line of the query expression is critical in this case—without the condition

$$\exists\, r \in student\ (r[ID] = t[ID])$$

if there is no course offered in the Biology department, any value of t (including values that are not student *ID*s in the *student* relation) would qualify.

27.1.2 Formal Definition

We are now ready for a formal definition. A tuple-relational-calculus expression is of the form:

$$\{t \mid P(t)\}$$

where P is a *formula*. Several tuple variables may appear in a formula. A tuple variable is said to be a *free variable* unless it is quantified by a $\exists$ or $\forall$. Thus, in:

$$t \in instructor \wedge \exists\, s \in department(t[dept\_name] = s[dept\_name])$$

t is a free variable. Tuple variable s is said to be a *bound* variable.

A tuple-relational-calculus formula is built up out of *atoms*. An atom has one of the following forms:

- $s \in r$, where s is a tuple variable and r is a relation (we do not allow use of the $\notin$ operator).
- $s[x]\ \Theta\ u[y]$, where s and u are tuple variables, x is an attribute on which s is defined, y is an attribute on which u is defined, and Θ is a comparison operator ($<$, $\leq$, $=$, $\neq$, $>$, $\geq$); we require that attributes x and y have domains whose members can be compared by Θ.
- $s[x]\ \Theta\ c$, where s is a tuple variable, x is an attribute on which s is defined, Θ is a comparison operator, and c is a constant in the domain of attribute x.

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).

- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $P_1 \Rightarrow P_2$.
- If $P_1(s)$ is a formula containing a free tuple variable s, and r is a relation, then

$$\exists s \in r\,(P_1(s)) \quad \text{and} \quad \forall s \in r\,(P_1(s))$$

are also formulae.

As we could for the relational algebra, we can write equivalent expressions that are not identical in appearance. In the tuple relational calculus, these equivalences include the following three rules:

1. $P_1 \wedge P_2$ is equivalent to $\neg\,(\neg(P_1) \vee \neg(P_2))$.
2. $\forall t \in r\,(P_1(t))$ is equivalent to $\neg\,\exists\, t \in r\,(\neg P_1(t))$.
3. $P_1 \Rightarrow P_2$ is equivalent to $\neg(P_1) \vee P_2$.

27.1.3 Safety of Expressions

There is one final issue to be addressed. A tuple-relational-calculus expression may generate an infinite relation. Suppose that we write the expression:

$$\{t \mid \neg\,(t \in \textit{instructor})\}$$

There are infinitely many tuples that are not in *instructor*. Most of these tuples contain values that do not even appear in the database! We do not wish to allow such expressions.

To help us define a restriction of the tuple relational calculus, we introduce the concept of the **domain** of a tuple relational formula, P. Intuitively, the domain of P, denoted $dom(P)$, is the set of all values referenced by P. They include values mentioned in P itself, as well as values that appear in a tuple of a relation mentioned in P. Thus, the domain of P is the set of all values that appear explicitly in P or that appear in one or more relations whose names appear in P. For example, $dom(t \in \textit{instructor} \wedge t[\textit{salary}] > 80000)$ is the set containing 80000 as well as the set of all values appearing in any attribute of any tuple in the *instructor* relation. Similarly, $dom(\neg\,(t \in \textit{instructor}))$ is also the set of all values appearing in *instructor*, since the relation *instructor* is mentioned in the expression.

We say that an expression $\{t \mid P(t)\}$ is *safe* if all values that appear in the result are values from $dom(P)$. The expression $\{t \mid \neg\,(t \in \textit{instructor})\}$ is not safe. Note that $dom(\neg\,(t \in \textit{instructor}))$ is the set of all values appearing in *instructor*. However, it is possible to have a tuple t not in *instructor* that contains values that do not appear in *instructor*. The other examples of tuple-relational-calculus expressions that we have written in this section are safe.

The number of tuples that satisfy an unsafe expression, such as $\{t \mid \neg (t \in instructor)\}$, could be infinite, whereas safe expressions are guaranteed to have finite results. The class of tuple-relational-calculus expressions that are allowed is therefore restricted to those that are safe.

27.2 The Domain Relational Calculus

A second form of relational calculus, called **domain relational calculus**, uses *domain* variables that take on values from an attributes domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus.

Domain relational calculus serves as the theoretical basis of the QBE language just as relational algebra serves as the basis for the SQL language.

27.2.1 Formal Definition

An expression in the domain relational calculus is of the form

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

where $x_1, x_2, \ldots, x_n$ represent domain variables. P represents a formula composed of atoms, as was the case in the tuple relational calculus. An atom in the domain relational calculus has one of the following forms:

- $< x_1, x_2, \ldots, x_n > \in r$, where r is a relation on n attributes and $x_1, x_2, \ldots, x_n$ are domain variables or domain constants.
- $x \Theta y$, where x and y are domain variables and Θ is a comparison operator ($<, \leq, =, \neq, >, \geq$). We require that attributes x and y have domains that can be compared by Θ.
- $x \Theta c$, where x is a domain variable, Θ is a comparison operator, and c is a constant in the domain of the attribute for which x is a domain variable.

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, and $P_1 \Rightarrow P_2$.
- If $P_1(x)$ is a formula in x, where x is a free domain variable, then

$$\exists x (P_1(x)) \text{ and } \forall x (P_1(x))$$

are also formulae.

As a notational shorthand, we write $\exists\, a, b, c\ (P(a, b, c))$ for $\exists\, a\ (\exists\, b\ (\exists\, c\ (P(a, b, c))))$.

27.2.2 Example Queries

We now give domain-relational-calculus queries for the examples that we considered earlier. Note the similarity of these expressions and the corresponding tuple-relational-calculus expressions.

- Find the instructor *ID*, *name*, *dept_name*, and *salary* for instructors whose salary is greater than $80,000:

$$\{ < i, n, d, s > \mid < i, n, d, s > \in \mathit{instructor} \wedge s > 80000\}$$

- Find all instructor *ID* for instructors whose salary is greater than $80,000:

$$\{ < i > \mid \exists\, n, d, s\ (< i, n, d, s > \in \mathit{instructor} \wedge s > 80000)\}$$

Although the second query appears similar to the one that we wrote for the tuple relational calculus, there is an important difference. In the tuple calculus, when we write $\exists\, s$ for some tuple variable s, we bind it immediately to a relation by writing $\exists\, s \in r$. However, when we write $\exists\, n$ in the domain calculus, n refers not to a tuple, but rather to a domain value. Thus, the domain of variable n is unconstrained until the subformula $< i, n, d, s > \in \mathit{instructor}$ constrains n to instructor names that appear in the *instructor* relation.

We now give several examples of queries in the domain relational calculus.

- Find the names of all instructors in the Physics department together with the *course_id* of all courses they teach:

$$\begin{aligned}\{ < n, c > \mid \exists\ & i, a, se, y\ (< i, c, a, se, y > \in \mathit{teaches} \\ & \wedge \exists\, d, s\ (< i, n, d, s > \in \mathit{instructor} \wedge d = \text{“Physics”}))\}\end{aligned}$$

- Find the set of all courses taught in the Fall 2017 semester, the Spring 2018 semester, or both:

$$\begin{aligned}\{ < c > \mid & \exists\, a, s, y, b, r, t\ (< c, a, s, y, b, r, t > \in \mathit{section} \\ & \wedge s = \text{“Fall”} \wedge y = \text{“2017”}) \\ & \vee \exists\, a, s, y, b, r, t\ (< c, a, s, y, b, r, t > \in \mathit{section} \\ & \wedge s = \text{“Spring”} \wedge y = \text{“2018”})\}\end{aligned}$$

- Find all students who have taken all courses offered in the Biology department:

$$\{ < i > \mid \exists\, n,\ d, tc\ (< i, n, d, tc > \in\ student) \wedge \\ \forall\, ci, ti, dn, cr\ (< ci, ti, dn, cr > \in\ course\ \wedge\ dn\ =\ \text{“Biology”}\ \Rightarrow \\ \exists\, si, se, y, g\ (< i, ci, si, se, y, g > \in\ takes\ ))\}$$

Note that as was the case for tuple relational calculus, if no courses are offered in the Biology department, all students would be in the result.

27.2.3 Safety of Expressions

We noted that, in the tuple relational calculus (Section 27.1), it is possible to write expressions that may generate an infinite relation. That led us to define *safety* for tuple-relational-calculus expressions. A similar situation arises for the domain relational calculus. An expression such as

$$\{ < i, n, d, s > \mid \neg(< i, n, d, s > \in\ instructor)\}$$

is unsafe, because it allows values in the result that are not in the domain of the expression.

For the domain relational calculus, we must be concerned also about the form of formulae within “there exists” and “for all” clauses. Consider the expression

$$\{ < x > \mid \exists\, y\ (< x,\ y > \in\ r)\ \wedge\ \exists\, z\ (\neg(< x,\ z > \in\ r)\ \wedge\ P(x, z))\}$$

where P is some formula involving x and z. We can test the first part of the formula, $\exists\, y\ (< x,\ y > \in\ r)$, by considering only the values in r. However, to test the second part of the formula, $\exists\, z\ (\neg\ (< x,\ z > \in\ r)\ \wedge\ P(x, z))$, we must consider values for z that do not appear in r. Since all relations are finite, an infinite number of values do not appear in r. Thus, it is not possible, in general, to test the second part of the formula without considering an infinite number of potential values for z. Instead, we add restrictions to prohibit expressions such as the preceding one.

In the tuple relational calculus, we restricted any existentially quantified variable to range over a specific relation. Since we did not do so in the domain calculus, we add rules to the definition of safety to deal with cases like our example. We say that an expression

$$\{ < x_1,\ x_2, \ldots, x_n > \mid P\,(x_1,\ x_2, \ldots, x_n)\}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from $dom(P)$.
2. For every “there exists” subformula of the form $\exists\ x\ (P_1(x))$, the subformula is true if and only if there is a value x in $dom(P_1)$ such that $P_1(x)$ is true.

3. For every "for all" subformula of the form $\forall x\ (P_1(x))$, the subformula is true if and only if $P_1(x)$ is true for all values x from $dom(P_1)$.

The purpose of the additional rules is to ensure that we can test "for all" and "there exists" subformulae without having to test infinitely many possibilities. Consider the second rule in the definition of safety. For $\exists\ x\ (P_1(x))$ to be true, we need to find only one x for which $P_1(x)$ is true. In general, there would be infinitely many values to test. However, if the expression is safe, we know that we can restrict our attention to values from $dom(P_1)$. This restriction reduces to a finite number the tuples we must consider.

The situation for subformulae of the form $\forall x\ (P_1(x))$ is similar. To assert that $\forall x\ (P_1(x))$ is true, we must, in general, test all possible values, so we must examine infinitely many values. As before, if we know that the expression is safe, it is sufficient for us to test $P_1(x)$ for those values taken from $dom(P_1)$.

All the domain-relational-calculus expressions that we have written in the example queries of this section are safe, except for the example unsafe query we saw earlier.

27.3 Expressive Power of Pure Relational Query Languages

The tuple relational calculus restricted to safe expressions is equivalent in **expressive power** to the basic relational algebra (with the operators $\cup$, $-$, $\times$, σ, Π, and ρ, but without the extended relational operations such as generalized projection and aggregation (γ)). Thus, for every relational-algebra expression using only the basic operations, there is an equivalent expression in the tuple relational calculus, and for every tuple-relational-calculus expression, there is an equivalent relational-algebra expression. We shall not prove this assertion here; the bibliographic notes contain references to the proof. Some parts of the proof are included in the exercises. We note that the tuple relational calculus does not have any equivalent of the aggregate operation, but it can be extended to support aggregation. Extending the tuple relational calculus to handle arithmetic expressions is straightforward.

When the domain relational calculus is restricted to safe expressions, it is equivalent in expressive power to the tuple relational calculus restricted to safe expressions. Since we noted earlier that the restricted tuple relational calculus is equivalent to the relational algebra, all three of the following are equivalent:

- The basic relational algebra (without the extended relational-algebra operations)
- The tuple relational calculus restricted to safe expressions
- The domain relational calculus restricted to safe expressions

We note that the domain relational calculus also does not have any equivalent of the aggregate operation, but it can be extended to support aggregation, and extending it to handle arithmetic expressions is straightforward.

account_number	*branch_name*	*balance*
A-101	Downtown	500
A-215	Minus	700
A-102	Perryridge	400
A-305	Round Hill	350
A-201	Perryridge	900
A-222	Redwood	700
A-217	Perryridge	750

Figure 27.4 The *account* relation.

27.4 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.

27.4.1 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 27.4. Then, the view relation *v1* contains the tuples in Figure 27.5.

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

account_number	*balance*
A-201	900
A-217	750

Figure 27.5 The *v1* relation.

$$? \; v1(\text{“A-217”}, B)$$

The answer to the query is

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

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

$$(\text{A-201}, 900)$$

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:

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

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 $10,000, then the interest rate is 5 percent, and if the balance is greater than or equal to $10,000, 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:

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

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

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

$$v1(\textit{account\_number}\ A, \textit{balance}\ B)\ :-$$
$$\textit{account}(\textit{account\_number}\ A, \textit{branch\_name}\ \text{“Perryridge”}, \textit{balance}\ B),$$
$$B > 700$$

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

27.4.2 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 27.4.3. 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:

$$\textit{account}(A, \text{“Perryridge”}, B)$$

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 $>$. The (conceptual) relation $>$ is infinite. Other arithmetic operations (such as $>$, $=$, $+$, and $-$) 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 $z = x + y$.

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

Figure 27.6 Datalog program that defines interest on Perryridge accounts.

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

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

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 27.6 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 preceding 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 until i_{n-1} depends on i_n.

In the example in Figure 27.6, 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 depends either 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**.

$$empl(X, Y) \text{ :- } manager(X, Y)$$
$$empl(X, Y) \text{ :- } manager(X, Z), empl(Z, Y)$$

Figure 27.7 Recursive Datalog program.

Consider the program in Figure 27.7. Here, the view relation *empl* depends on itself (because of the second rule), and is therefore recursive. In contrast, the program in Figure 27.6 is nonrecursive.

27.4.3 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 27.4.6. We define the semantics of a program by starting with the semantics of a single rule.

27.4.3.1 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:

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

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,

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

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:

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

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** $q_i(v_{i,1}, v_{i,2}, \ldots, v_{i,n_i})$, 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 $q_j(v_{j,1}, \ldots, v_{j,n_j})$.

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

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

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

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

Suppose that we are given a set of facts I containing the tuples for relation *account* in Figure 27.4. One possible instantiation of our running-example rule R is

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

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 27.8.

27.4.3.2 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:

account_number	*balance*
A-201	900
A-217	750

Figure 27.8 Result of *infer*(*R*, *I*).

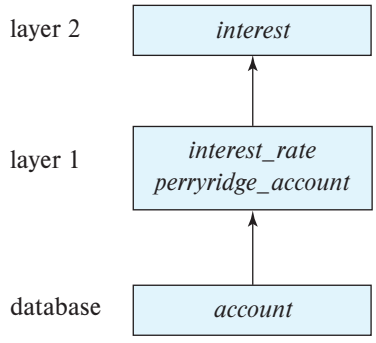

Figure 27.9 Layering of view relations.

- 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 27.6 with the additional rule:

$$perryridge\_account(X, Y) \text{ :- } account(X, \text{“Perryridge”}, Y)$$

The layering of view relations in the program appears in Figure 27.9. The relation *account* is in the database. Relation *interest_rate* is in layer 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 we define I_1 as

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

- 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:

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

- 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 27.6, 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.

27.4.4 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:

$$not\_in\_loan(L, B, A) \text{ :- } \textbf{not } loan(L, B, A)$$

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 loan_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.

27.4.5 Relational Operations in Datalog

Nonrecursive Datalog expressions without arithmetic operations are equivalent in expressive power to expressions using the basic operations in relational algebra ($\cup$, $-$, $\times$, σ, Π, 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 relational update operations (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

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

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

27.4.6 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 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

$$\textit{Manager\_schema} = (\textit{employee\_name, manager\_name})$$

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*:

$$\textit{empl\_jones}(X) \text{ :- } \textit{manager}(X, \text{“Jones”})$$
$$\textit{empl\_jones}(X) \text{ :- } \textit{manager}(X, Y), \textit{empl\_jones}(Y)$$

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

```
procedure Datalog-Fixpoint
    I = set of facts in the database
    repeat
        Old_I = I
        I = I ∪ infer(R, I)
    until I = Old_I
```

Figure 27.10 Datalog-Fixpoint procedure.

negative literals. The reason will become clear later. The bibliographical 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 27.10. The recursion in the Datalog program has been turned into an iteration in the procedure. At the end of the procedure, $infer(\mathcal{R}, I) \cup D = I$, where D is the set of facts in the database, and I is called a **fixed point** of the program.

Consider the program defining *empl_jones*, with the relation *manager*, as in Figure 27.11. The set of facts computed for the view relation *empl_jones* in each iteration appears in Figure 27.12. 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

$$? \; empl\_jones(N)$$

employee_name	*manager_name*
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Figure 27.11 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 27.12 Employees of Jones in iterations of procedure Datalog-Fixpoint.

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.

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 27.7):

$$empl(X, Y) \text{ :- } manager(X, Y)$$
$$empl(X, Y) \text{ :- } manager(X, Z), empl(Z, Y)$$

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

[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).

? *empl*(*X*, "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 at which 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*.

27.4.7 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 27.10. 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:

$$number(0)$$
$$number(A) \text{ :- } number(B), A = B + 1$$

The program generates *number*(*n*) for all positive integers *n*, which is infinite and will not terminate. The second rule of the program does not satisfy the safety condition in Section 27.4.4. 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.

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

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

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 27.4.6, we can then show that procedure Datalog-Fixpoint is sound (i.e., 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

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

and let

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

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 bibliographical 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.

27.5 Summary

- The tuple relational calculus and the domain relational calculus are nonprocedural languages that represent the basic power required in a relational query language. The basic relational algebra is a procedural language that is equivalent in power to both forms of the relational calculus when they are restricted to safe expressions.
- The relational calculi are terse, formal languages that are inappropriate for casual users of a database system. These two formal languages form the basis for two more user-friendly languages, QBE and Datalog.
- The tuple relational calculus and the domain relational calculus are terse, formal languages that are inappropriate for casual users of a database system. Commercial database systems, therefore, use languages with more "syntactic sugar." We have considered two query languages: QBEand Datalog.
- 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 make 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.

Review Terms

- Tuple relational calculus
- Domain relational calculus
- Safety of expressions
- Expressive power of languages
- Datalog
- Rules

- Uses
- Defines
- Positive literal
- Negative literal
- Fact
- Recursive view
- Nonrecursive view
- Instantiation
- Infer
- Semantics
- Safety
- Fixed point
- Transitive closure
- Monotonic view definition

Practice Exercises

27.1 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 the tuple relational calculus that is equivalent to each of the following:

a. $\Pi_A(r)$

b. $\sigma_{B=17}(r)$

c. $r \times s$

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

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

a. $\Pi_A(r_1)$

b. $\sigma_{B=17}(r_1)$

c. $r_1 \cup r_2$

d. $r_1 \cap r_2$

e. $r_1 - r_2$

f. $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

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

a. $\{ < a > \mid \exists\, b\, (< a, b > \in r \wedge b = 7)\}$

b. $\{ < a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$

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

Figure 27.13 Employee database.

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

27.4 Consider the relational database of Figure 27.13 where the primary keys are underlined. Give an expression in tuple relational calculus for each of the following queries:

a. Find all employees who work directly for "Jones."

b. Find all cities of residence of all employees who work directly for "Jones."

c. Find the name of the manager of the manager of "Jones."

d. Find those employees who earn more than all employees living in the city "Mumbai."

27.5 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Write expressions in 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. $\{ < a > \mid \exists\, c\, (< a, c > \in s \wedge \exists\, b_1, b_2\, (< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

27.6 Consider the relational database of Figure 27.13 where the primary keys are underlined. Give an expression in Datalog 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.

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

Exercises

27.8 Consider the employee database of Figure 27.13. Give expressions in tuple relational calculus for each of the following queries:

a. Find the names of all employees who work for "FBC".

b. Find the names and cities of residence of all employees who work for "FBC".

c. Find the names, street addresses, and cities of residence of all employees who work for "FBC" and earn more than $10,000.

d. Find all employees who live in the same city as that in which 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 "FBC".

g. Find all employees who earn more than every employee of "SBC".

h. Assume that the companies may be located in several cities. Find all companies located in every city in which "SBC" is located.

27.9 Repeat Exercise 27.8, writing domain relational calculus queries instead of tuple relational calculus queries.

27.10 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Write relational-algebra expressions equivalent to the following domain-relational-calculus expressions:

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. $\{ < a > \mid \exists\, b\, (< a, b > \in r) \vee \forall\, c\, (\exists\, d\, (< d, c > \in s) \Rightarrow < a, c > \in s)\}$

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

27.11 Repeat Exercise 27.10, writing SQL queries instead of relational-algebra expressions.

27.12 Let $R = (A, B)$ and $S = (A, C)$, and let $r(R)$ and $s(S)$ be relations. Using the special constant *null*, write tuple-relational-calculus expressions equivalent to each of the following:

a. $r \ltimes s$

b. $r \rtimes s$

c. $r \bowtie s$

27.13 Give a tuple-relational-calculus expression to find the maximum value in relation $r(A)$.

27.14 Give a tuple-relational-calculus expression to find the maximum value in relation $r(A)$.

27.15 Repeat Exercise 27.8 using Datalog.

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

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

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

Tools

The Coral system from the University of Wisconsin – Madison (research.cs.wisc.edu/coral) is an implementation of Datalog. The XSB system from Stony Brook University (xsb.sourceforge.net) is a widely used Prolog implementation that supports database querying; recall that Datalog is a nonprocedural subset of Prolog.

Further Reading

Extensions to the relational model and discussions of incorporation of null values in the relational algebra (the RM/T model), as well as outer joins, are in [Codd (1979)].

[Codd (1990)] is a compendium of E. F. Codd's papers on the relational model. Outer joins are also discussed in [Date (1983)].

The original definition of tuple relational calculus is in [Codd (1972)]. A formal proof of the equivalence of tuple relational calculus and relational algebra is in [Codd (1972)]. Several extensions to the relational calculus have been proposed. [Klug (1982)] and [Escobar-Molano et al. (1993)] describe extensions to scalar aggregate functions.

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. (1992)].

Bibliography

[Apt and Pugin (1987)] K. R. Apt and J. M. Pugin, "Maintenance of Stratified Database Viewed as a Belief Revision System", In *Proc. of the ACM Symposium on Principles of Database Systems* (1987), pages 136–145.

[Chandra and Harel (1982)] A. K. Chandra and D. Harel, "Structure and Complexity of Relational Queries", *Journal of Computer and System Sciences*, Volume 15, Number 10 (1982), pages 99–128.

[Codd (1972)] E. F. Codd. "Further Normalization of the Data Base Relational Model", In *[Rustin (1972)]*, pages 33–64 (1972).

[Codd (1979)] E. F. Codd, "Extending the Database Relational Model to Capture More Meaning", *ACM Transactions on Database Systems*, Volume 4, Number 4 (1979), pages 397–434.

[Codd (1990)] E. F. Codd, *The Relational Model for Database Management: Version 2*, Addison Wesley (1990).

[Date (1983)] C. J. Date, "The Outer Join", In *Proc. of the International Conference on Databases*, John Wiley and Sons (1983), pages 76–106.

[Escobar-Molano et al. (1993)] M. Escobar-Molano, R. Hull, and D. Jacobs, "Safety and Translation of Calculus Queries with Scalar Functions", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1993), pages 253–264.

[Klug (1982)] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions", *Journal of the ACM*, Volume 29, Number 3 (1982), pages 699–717.

[Ramakrishnan et al. (1992)] R. Ramakrishnan, D. Srivastava, and S. Sudarshan, *Controlling the Search in Bottom-up Evaluation* (1992).

[Ross (1990)] K. A. Ross, "Modular Stratification and Magic Sets for DATALOG Programs with Negation", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 161–171.

[Rustin (1972)] R. Rustin, *Data Base Systems*, Prentice Hall (1972).

Credits

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

CHAPTER 28

Advanced Relational Database Design

In this chapter we cover advanced topics in relational database design. We first present the theory of multivalued dependencies, including a set of sound and complete inference rules for multivalued dependencies. We then present PJNF and DKNF, two normal forms based on classes of constraints that generalize multivalued dependencies.

In this chapter we illustrate our concepts using a bank enterprise with the schema shown in Figure 28.1.

28.1 Multivalued Dependencies

As we did for functional dependencies and 3NF and BCNF, we shall need to determine all the multivalued dependencies that are logically implied by a given set of multivalued dependencies.

28.1.1 Theory of Multivalued Dependencies

We take the same approach here that we did earlier for functional dependencies. 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

---

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 28.1 Bank database.

for functional dependencies, we can compute D^+ from D, using the formal definitions of functional dependencies and multivalued dependencies. However, it is usually easier to reason about sets of dependencies by using a system of inference rules.

The following list of inference rules for functional and multivalued dependencies is *sound* and *complete*. Recall that *sound* rules do not generate any dependencies that are not logically implied by D, and *complete* rules allow us to generate all dependencies in D^+. The first three rules are Armstrong's axioms, which we saw in Chapter 7.

1. **Reflexivity rule**. If α is a set of attributes, and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
2. **Augmentation rule**. If $\alpha \rightarrow \beta$ holds, and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
3. **Transitivity rule**. If $\alpha \rightarrow \beta$ holds, and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.
4. **Complementation rule**. If $\alpha \twoheadrightarrow \beta$ holds, then $\alpha \twoheadrightarrow R - \beta - \alpha$ holds.
5. **Multivalued augmentation rule**. If $\alpha \twoheadrightarrow \beta$ holds, and $\gamma \subseteq R$ and $\delta \subseteq \gamma$, then $\gamma\alpha \twoheadrightarrow \delta\beta$ holds.
6. **Multivalued transitivity rule**. If $\alpha \twoheadrightarrow \beta$ holds, and $\beta \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \gamma - \beta$ holds.
7. **Replication rule**. If $\alpha \rightarrow \beta$ holds, then $\alpha \twoheadrightarrow \beta$.
8. **Coalescence rule**. If $\alpha \twoheadrightarrow \beta$ holds, and $\gamma \subseteq \beta$, and there is a δ such that $\delta \subseteq R$, and $\delta \cap \beta = \emptyset$, and $\delta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ holds.

The bibliographical notes provide references to proofs that the preceding rules are sound and complete. The following examples provide insight into how the formal proofs proceed.

Let $R = (A, B, C, G, H, I)$ be a relation schema. Suppose that $A \twoheadrightarrow BC$ holds. The definition of multivalued dependencies implies that, if $t_1[A] = t_2[A]$, then there exist tuples t_3 and t_4 such that

$$
\begin{aligned}
&t_1[A] = t_2[A] = t_3[A] = t_4[A] \\
&t_3[BC] = t_1[BC] \\
&t_3[GHI] = t_2[GHI] \\
&t_4[GHI] = t_1[GHI] \\
&t_4[BC] = t_2[BC]
\end{aligned}
$$

The complementation rule states that, if $A \twoheadrightarrow BC$, then $A \twoheadrightarrow GHI$. Observe that t_3 and t_4 satisfy the definition of $A \twoheadrightarrow GHI$ if we simply change the subscripts.

We can provide similar justification for rules 5 and 6 (see Exercise 28.2) using the definition of multivalued dependencies.

Rule 7, the replication rule, involves functional and multivalued dependencies. Suppose that $A \rightarrow BC$ holds on R. If $t_1[A] = t_2[A]$ and $t_1[BC] = t_2[BC]$, then t_1 and t_2

themselves serve as the tuples t_3 and t_4 required by the definition of the multivalued dependency $A \twoheadrightarrow BC$.

Rule 8, the coalescence rule, is the most difficult of the eight rules to verify (see Exercise 28.4).

We can simplify the computation of the closure of D by using the following rules, which we can prove using rules 1 to 8 (see Exercise 28.5):

- **Multivalued union rule**. If $\alpha \twoheadrightarrow \beta$ holds, and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta\gamma$ holds.
- **Intersection rule**. If $\alpha \twoheadrightarrow \beta$ holds, and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta \cap \gamma$ holds.
- **Difference rule**. If $\alpha \twoheadrightarrow \beta$ holds, and $\alpha \twoheadrightarrow \gamma$ holds, then $\alpha \twoheadrightarrow \beta - \gamma$ holds and $\alpha \twoheadrightarrow \gamma - \beta$ holds.

Let us apply our rules to the following example. Let $R = (A, B, C, G, H, I)$ with the following set of dependencies D given:

$$A \twoheadrightarrow B$$
$$B \twoheadrightarrow HI$$
$$CG \rightarrow H$$

We list several members of D^+ here:

- $A \twoheadrightarrow CGHI$: Since $A \twoheadrightarrow B$, the complementation rule (rule 4) implies that $A \twoheadrightarrow R - B - A$. $R - B - A = CGHI$, so $A \twoheadrightarrow CGHI$.
- $A \twoheadrightarrow HI$: Since $A \twoheadrightarrow B$ and $B \twoheadrightarrow HI$, the multivalued transitivity rule (rule 6) implies that $A \twoheadrightarrow HI - B$. Since $HI - B = HI$, $A \twoheadrightarrow HI$.
- $B \rightarrow H$: To show this fact, we need to apply the coalescence rule (rule 8). $B \twoheadrightarrow HI$ holds. Since $H \subseteq HI$ and $CG \rightarrow H$ and $CG \cap HI = \emptyset$, we satisfy the statement of the coalescence rule, with α being B, β being HI, δ being CG, and γ being H. We conclude that $B \rightarrow H$.
- $A \twoheadrightarrow CG$: We already know that $A \twoheadrightarrow CGHI$ and $A \twoheadrightarrow HI$. By the difference rule, $A \twoheadrightarrow CGHI - HI$. Since $CGHI - HI = CG$, $A \twoheadrightarrow CG$.

28.1.2 Dependency Preservation

The question of dependency preservation when we have multivalued dependencies is not as simple as it is when we have only functional dependencies.

A decomposition of schema R into schemas $R_1, R_2, \ldots, R_n$ is a **dependency-preserving decomposition** with respect to a set D of functional and multivalued dependencies if, for every set of relations $r_1(R_1), r_2(R_2), \ldots, r_n(R_n)$ such that for all i, r_i satisfies D_i (the restriction of D to R_i), there exists a relation $r(R)$ that satisfies D and for which $r_i = \Pi_{R_i}(r)$ for all i.

```
result := {R};
done := false;
compute D+; Given schema Ri, let Di denote the restriction of D+ to Ri
while (not done) do
   if (there is a schema Ri in result that is not in 4NF w.r.t. Di)
     then begin
            let α →→ β be a nontrivial multivalued dependency that holds
            on Ri such that α → Ri is not in Di, and α ∩ β = ∅;
            result := (result − Ri) ∪ (Ri − β) ∪ (α, β);
          end
     else done := true;
```

Figure 28.2 4NF decomposition algorithm.

Let us apply the 4NF decomposition algorithm that we presented in Chapter 7 to the schema $R = (A, B, C, G, H, I)$ with $D = \{A \twoheadrightarrow B,\ B \twoheadrightarrow HI,\ CG \rightarrow H\}$ (for convenience, we repeat the algorithm in Figure 28.2). We shall then test the resulting decomposition for dependency preservation. We first need to compute the closure of D. The nontrivial dependencies in closure include all the dependencies in D and the multivalued dependency $A \twoheadrightarrow HI$, as we saw in Section 28.1.1.

R is not in 4NF. Observe that $A \twoheadrightarrow B$ is not trivial, yet A is not a superkey. Using $A \twoheadrightarrow B$ in the first iteration of the **while** loop, we replace R with two schemas, (A, B) and (A, C, G, H, I). It is easy to see that (A, B) is in 4NF since all multivalued dependencies that hold on (A, B) are trivial. However, the schema (A, C, G, H, I) is not in 4NF. Applying the multivalued dependency $CG \twoheadrightarrow H$ (which follows from the given functional dependency $CG \rightarrow H$ by the replication rule), we replace (A, C, G, H, I) with the two schemas (C, G, H) and (A, C, G, I).

Schema (C, G, H) is in 4NF, but schema (A, C, G, I) is not. To see that (A, C, G, I) is not in 4NF, we note that since $A \twoheadrightarrow HI$ is in D^+, $A \twoheadrightarrow I$ is in the restriction of D to (A, C, G, I). Thus, in a third iteration of the **while** loop, we replace (A, C, G, I) with two schemas (A, I) and (A, C, G). The algorithm then terminates, and the resulting 4NF decomposition is $\{(A, B), (C, G, H), (A, I), (A, C, G)\}$.

This 4NF decomposition is not dependency preserving, since it fails to preserve the multivalued dependency $B \twoheadrightarrow HI$. Consider Figure 28.3, which shows the four relations that may result from the projection of a relation on (A, B, C, G, H, I) onto the four schemas of our decomposition. The restriction of D to (A, B) is $A \twoheadrightarrow B$ and some trivial dependencies. It is easy to see that r_1 satisfies $A \twoheadrightarrow B$, because there is no pair of tuples with the same A value. Observe that r_2 satisfies *all* functional and multivalued dependencies, since no two tuples in r_2 have the same value on any attribute. A similar statement can be made for r_3 and r_4. Therefore, the decomposed version of

r_1:

A	B
a_1	b_1
a_2	b_1

r_2:

C	G	H
c_1	g_1	h_1
c_2	g_2	h_2

r_3:

A	I
a_1	i_1
a_2	i_2

r_4:

A	C	G
a_1	c_1	g_1
a_2	c_2	g_2

Figure 28.3 Projection of relation r onto a 4NF decomposition of R.

our database satisfies all the dependencies in the restriction of D. However, there is no relation r on (A, B, C, G, H, I) that satisfies D and decomposes into r_1, r_2, r_3, and r_4. Figure 28.4 shows the relation $r = r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$. Relation r does not satisfy $B \twoheadrightarrow HI$. Any relation s containing r and satisfying $B \twoheadrightarrow HI$ must include the tuple $(a_2, b_1, c_2, g_2, h_1, i_1)$. However, $\Pi_{CGH}(s)$ includes a tuple (c_2, g_2, h_1) that is not in r_2. Thus, our decomposition fails to detect a violation of $B \twoheadrightarrow HI$.

We have seen that, if we are given a set of multivalued and functional dependencies, it is advantageous to find a database design that meets the three criteria of

1. 4NF
2. Dependency preservation
3. Lossless join

If all we have are functional dependencies, then the first criterion is just BCNF.

We have seen also that it is not always possible to meet all three of these criteria. We succeeded in finding such a decomposition for the bank example, but failed for the example of schema $R = (A, B, C, G, H, I)$.

A	B	C	G	H	I
a_1	b_1	c_1	g_1	h_1	i_1
a_2	b_1	c_2	g_2	h_2	i_2

Figure 28.4 A relation $r(R)$ that does not satisfy $B \twoheadrightarrow HI$.

When we cannot achieve our three goals, we have to compromise on 4NF or dependency preservation.

28.2 Join Dependencies

We have seen that the lossless-join property is one of several properties of a good database design. Indeed, this property is essential: Without it, information is lost. When we restrict the set of legal relations to those satisfying a set of functional and multivalued dependencies, we are able to use these dependencies to show that certain decompositions are lossless-join decompositions.

Because of the importance of the concept of lossless join, it is useful to be able to constrain the set of legal relations over a schema R to those relations for which a given decomposition is a lossless-join decomposition. In this section, we define such a constraint, called a **join dependency**. Just as types of dependency led to other normal forms, join dependencies will lead to a normal form called **project-join normal form (PJNF)**.

28.2.1 Definition of Join Dependencies

Let R be a relation schema and $R_1, R_2, \ldots, R_n$ be a decomposition of R. The join dependency $*(R_1, R_2, \ldots, R_n)$ is used to restrict the set of legal relations to those for which $R_1, R_2, \ldots, R_n$ is a lossless-join decomposition of R. Formally, if $R = R_1 \cup R_2 \cup \ldots \cup R_n$, we say that a relation $r(R)$ satisfies the *join dependency* $*(R_1, R_2, \ldots, R_n)$ if

$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie \cdots \bowtie \Pi_{R_n}(r)$$

A join dependency is *trivial* if one of the R_i is R itself.

Consider the join dependency $*(R_1, R_2)$ on schema R. This dependency requires that, for all legal $r(R)$,

$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$$

Let r contain the two tuples t_1 and t_2, defined as follows:

$$\begin{array}{ll} t_1[R_1 - R_2] = (a_1, a_2, \ldots, a_i) & t_2[R_1 - R_2] = (b_1, b_2, \ldots, b_i) \\ t_1[R_1 \cap R_2] = (a_{i+1}, \ldots, a_j) & t_2[R_1 \cap R_2] = (a_{i+1}, \ldots, a_j) \\ t_1[R_2 - R_1] = (a_{j+1}, \ldots, a_n) & t_2[R_2 - R_1] = (b_{j+1}, \ldots, b_n) \end{array}$$

Thus, $t_1[R_1 \cap R_2] = t_2[R_1 \cap R_2]$, but t_1 and t_2 have different values on all other attributes. Let us compute $\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$. Figure 28.5 shows $\Pi_{R_1}(r)$ and $\Pi_{R_2}(r)$. When we compute the join, we get two tuples in addition to t_1 and t_2, shown by t_3 and t_4 in Figure 28.6.

If $*(R_1, R_2)$ holds, then, whenever we have tuples t_1 and t_2, we must also have t_3 and t_4. Thus, Figure 28.6 shows a tabular representation of the join dependency $*(R_1, R_2)$.

	$R_1 - R_2$	$R_1 \cap R_2$
$\Pi_{R_1}(t_1)$	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$
$\Pi_{R_1}(t_2)$	$b_1 \ldots b_i$	$a_{i+1} \ldots a_j$

	$R_1 \cap R_2$	$R_2 - R_1$
$\Pi_{R_2}(t_1)$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
$\Pi_{R_2}(t_2)$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$

Figure 28.5 $\Pi_{R_1}(r)$ *and* $\Pi_{R_2}(r)$.

Compare Figure 28.6 with Figure 7.13, in which we gave a tabular representation of $\alpha \twoheadrightarrow \beta$. If we let $\alpha = R_1 \cap R_2$ and $\beta = R_1$, then we can see that the two tabular representations in these figures are the same. Indeed, $*(R_1, R_2)$ is just another way of stating $R_1 \cap R_2 \twoheadrightarrow R_1$. Using the complementation and augmentation rules for multivalued dependencies, we can show that $R_1 \cap R_2 \twoheadrightarrow R_1$ implies $R_1 \cap R_2 \twoheadrightarrow R_2$. Thus, $*(R_1, R_2)$ is equivalent to $R_1 \cap R_2 \twoheadrightarrow R_2$. This observation is not surprising in light of the fact we noted earlier that R_1 and R_2 form a lossless-join decomposition of R if and only if $R_1 \cap R_2 \twoheadrightarrow R_2$ or $R_1 \cap R_2 \twoheadrightarrow R_1$.

Every join dependency of the form $*(R_1, R_2)$ is therefore equivalent to a multivalued dependency. However, there are join dependencies that are not equivalent to any multivalued dependency. The simplest example of such a dependency is on schema $R = (A, B, C)$. The join dependency

$$*((A, B), (B, C), (A, C))$$

is not equivalent to any collection of multivalued dependencies. Figure 28.7 shows a tabular representation of this join dependency. To see that no set of multivalued dependencies logically implies $*((A, B), (B, C), (A, C))$, we consider Figure 28.7 as a relation $r(A, B, C)$, as in Figure 28.8. Relation r satisfies the join dependency $*((A, B), (B, C), (A, C))$, as we can verify by computing

$$\Pi_{AB}(r) \bowtie \Pi_{BC}(r) \bowtie \Pi_{AC}(r)$$

	$R_1 - R_2$	$R_1 \cap R_2$	$R_2 - R_1$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$b_1 \ldots b_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$b_1 \ldots b_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$

Figure 28.6 Tabular representation of $*(R_1, R_2)$.

A	B	C
a_1	b_1	c_2
a_2	b_1	c_1
a_1	b_2	c_1
a_1	b_1	c_1

Figure 28.7 Tabular representation of *((*A, B*), (*B, C*), (*A, C*)).

and by showing that the result is exactly r. However, r does not satisfy any nontrivial multivalued dependency. To see that it does not, we verify that r fails to satisfy any of $A \twoheadrightarrow B$, $A \twoheadrightarrow C$, $B \twoheadrightarrow A$, $B \twoheadrightarrow C$, $C \twoheadrightarrow A$, or $C \twoheadrightarrow B$.

Just as a multivalued dependency is a way of stating the independence of a pair of relationships, a join dependency is a way of stating that the members of a *set* of relationships are all independent. This notion of independence of relationships is a natural consequence of the way that we generally define a relation. Consider

$$Loan\_info\_schema = (branch\_name, customer\_name, loan\_number, amount)$$

from our banking example. We can define a relation *loan_info* (*Loan_info_schema*) as the set of all tuples on *Loan_info_schema* such that

- The loan represented by *loan_number* is made by the branch named *branch_name*.
- The loan represented by *loan_number* is made to the customer named *customer_name*.
- The loan represented by *loan_number* is in the amount given by *amount*.

The preceding definition of the *loan_info* relation is a conjunction of three predicates: one on *loan_number* and *branch_name*, one on *loan_number* and *customer_name*, and one on *loan_number* and *amount*. Surprisingly, it can be shown that the preceding intuitive definition of *loan_info* logically implies the join dependency *((*loan_number, branch_name*), (*loan_number, customer_name*), (*loan_number, amount*)).

Thus, join dependencies have an intuitive appeal and correspond to one of our three criteria for a good database design.

A	B	C
a_1	b_1	c_2
a_2	b_1	c_1
a_1	b_2	c_1
a_1	b_1	c_1

Figure 28.8 Relation *r* (*A, B, C*).

For functional and multivalued dependencies, we were able to give a system of inference rules that are sound and complete. Unfortunately, no such set of rules is known for join dependencies. It appears that we must consider more general classes of dependencies than join dependencies to construct a sound and complete set of inference rules. The bibliographical notes contain references to research in this area.

28.2.2 Project-Join Normal Form

Project-join normal form (*PJNF*) is defined in the same way as BCNF and 4NF, except that join dependencies are used. A relation schema R is in PJNF with respect to a set D of functional, multivalued, and join dependencies if, for all join dependencies in D^+ of the form $*(R_1, R_2, \ldots, R_n)$, where each $R_i \subseteq R$ and $R = R_1 \cup R_2 \cup \ldots \cup R_n$, at least one of the following holds:

- $*(R_1, R_2, \ldots, R_n)$ is a trivial join dependency.
- Every R_i is a superkey for R.

A database design is in PJNF if each member of the set of relation schemas that constitutes the design is in PJNF. PJNF is called *fifth normal form* (5NF) in some of the literature on database normalization.

Consider again our banking example. Given the join dependency *((*loan_number, branch_name*), (*loan_number, customer_name*), (*loan_number, amount*)), *Loan_info_schema* is not in PJNF. To put *Loan_info_schema* into PJNF, we must decompose it into the three schemas specified by the join dependency: (*loan_number, branch_name*), (*loan_number, customer_name*), and (*loan_number, amount*).

Because every multivalued dependency is also a join dependency, it is easy to see that every PJNF schema is also in 4NF. Thus, in general, we may not be able to find a dependency-preserving decomposition into PJNF for a given schema.

28.3 Domain-Key Normal Form

The approach we have taken to normalization is to define a form of constraint (functional, multivalued, or join dependency), and then to use that form of constraint to define a normal form. *Domain-key normal form* (*DKNF*) is based on three notions.

1. **Domain declaration**. Let A be an attribute, and let **dom** be a set of values. The domain declaration $A \subseteq$ **dom** requires that the A value of all tuples be values in **dom**.
2. **Key declaration**. Let R be a relation schema with $K \subseteq R$. The key declaration **key** (K) requires that K be a superkey for schema R—that is, $K \rightarrow R$. Note that all key declarations are functional dependencies, but not all functional dependencies are key declarations.

3. **General constraint**. A *general constraint* is a predicate on the set of all relations on a given schema. The dependencies that we have studied in this chapter are examples of general constraints. In general, a general constraint is a predicate expressed in some agreed-on form, such as first-order logic.

We now give an example of a general constraint that is not a functional, multivalued, or join dependency. Suppose that all accounts whose *account_number* begins with the digit 9 are special high-interest accounts with a minimum balance of $2500. Then, we include as a general constraint, "If the first digit of t[*account_number*] is 9, then t[*balance*] $\geq$ 2500."

Domain declarations and key declarations are easy to test in a practical database system. General constraints, however, may be extremely costly (in time and space) to test. The purpose of a DKNF database design is to allow us to test the general constraints using only domain and key constraints.

Formally, let **D** be a set of domain constraints and let **K** be a set of key constraints for a relation schema R. Let **G** denote the general constraints for R. Schema R is in DKNF if **D** $\cup$ **K** logically implies **G**.

Let us return to the general constraint that we gave on accounts. The constraint implies that our database design is not in DKNF. To create a DKNF design, we need two schemas in place of *Account_schema*:

Regular_acct_schema = (*account_number, branch_name, balance*)
Special_acct_schema = (*account_number, branch_name, balance*)

We retain all the dependencies that we had on *Account_schema* as general constraints. The domain constraints for *Special_acct_schema* require that, for each account,

- The account number begins with 9.
- The balance is greater than 2500.

The domain constraints for *Regular_acct_schema* require that the account number does not begin with 9. The resulting design is in DKNF, although the proof of this fact is beyond the scope of this text.

Let us compare DKNF to the other normal forms that we have studied. Under the other normal forms, we did not take into consideration domain constraints. We assumed (implicitly) that the domain of each attribute was some infinite domain, such as the set of all integers or the set of all character strings. We allowed key constraints (indeed, we allowed functional dependencies). For each normal form, we allowed a restricted form of general constraint (a set of functional, multivalued, or join dependencies). Thus, we can rewrite the definitions of PJNF, 4NF, BCNF, and 3NF in a manner that shows them to be special cases of DKNF.

We now present a DKNF-inspired rephrasing of our definition of PJNF. Let $R = (A_1, A_2, \ldots, A_n)$ be a relation schema. Let $\text{dom}(A_i)$ denote the domain of attribute A_i, and let all these domains be infinite. Then all domain constraints **D** are of the form $A_i \subseteq \text{dom}(A_i)$. Let the general constraints be a set **G** of functional, multivalued, or join dependencies. If F is the set of functional dependencies in **G**, let the set **K** of key constraints be those nontrivial functional dependencies in F^+ of the form $\alpha \rightarrow R$. Schema R is in PJNF if and only if it is in DKNF with respect to **D**, **K**, and **G**.

A consequence of DKNF is that all insertion and deletion anomalies are eliminated.

DKNF represents an "ultimate" normal form because it allows arbitrary constraints, rather than dependencies, yet it allows efficient testing of these constraints. Of course, if a schema is not in DKNF, we may be able to achieve DKNF via decomposition, but such decompositions, as we have seen, are not always dependency-preserving decompositions. Thus, although DKNF is a goal of a database designer, it may have to be sacrificed in a practical design.

28.4 Summary

- We presented the theory of multivalued dependencies, including a set of sound and complete inference rules for multivalued dependencies.
- Join dependencies are a generalization of multivalued dependencies and lead to the definition of PJNF.
- DKNF is an idealized normal form that may be difficult to achieve in practice. Yet DKNF has desirable properties that should be included to the extent possible in a good database design.

Review Terms

- Dependency-preserving decomposition
- Join dependency
- project-join normal form (PJNF)

Exercises

28.1 List all the nontrivial multivalued dependencies satisfied by the relation in Figure 28.9.

28.2 Use the definition of multivalued dependency (Section 7.6.1) to argue that each of the following axioms is sound:

a. The complementation rule

b. The multivalued augmentation rule

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 28.9 Relation of Exercise 28.1.

c. The multivalued transitivity rule

28.3 Use the definitions of functional and multivalued dependencies: (Section 7.4 and Section 7.6.1) to show the soundness of the replication rule.

28.4 Show that the coalescence rule is sound. (*Hint*: Apply the definition of $\alpha \twoheadrightarrow \beta$ to a pair of tuples t_1 and t_2 such that $t_1[\alpha] = t_2[\alpha]$. Observe that since $\delta \cap \beta = \emptyset$, if two tuples have the same value on $R - \beta$, then they have the same value on δ.)

28.5 Use the axioms for functional and multivalued dependencies to show that each of the following rules is sound:

a. The multivalued union rule

b. The intersection rule

c. The difference rule

28.6 Give a lossless-join decomposition of schema R in Exercise 28.11 into 4NF. AVI -check reference

28.7 Give an example of relation schema R and a set of dependencies such that R is in 4NF but is not in PJNF.

28.8 Explain why PJNF is a normal form more desirable than is 4NF.

28.9 Rewrite the definitions of 4NF and BCNF using the notions of domain constraints and general constraints.

28.10 Explain why DKNF is a highly desirable normal form, yet is one that is difficult to achieve in practice.

28.11 Let $R = (A, B, C, D, E)$, and let M be the following set of multivalued dependencies:

$$A \twoheadrightarrow BC$$
$$B \twoheadrightarrow CD$$
$$E \twoheadrightarrow AD$$

List the nontrivial dependencies in M^+.

Further Reading

The notions of 4NF, PJNF, and DKNF are from [Fagin (1977)], [Fagin (1979)], and [Fagin (1981)], respectively. Additional dependencies are discussed in detail in [Maier (1983)].

Bibliography

[Fagin (1977)] R. Fagin, "Multivalued Dependencies and a New Normal Form for Relational Databases", *ACM Transactions on Database Systems*, Volume 2, Number 3 (1977), pages 262–278.

[Fagin (1979)] R. Fagin, "Normal Forms and Relational Database Operators", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1979), pages 153–160.

[Fagin (1981)] R. Fagin, "A Normal Form for Relational Databases That Is Based on Domains and Keys", *ACM Transactions on Database Systems*, Volume 6, Number 3 (1981), pages 387–415.

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

Credits

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

CHAPTER 29

Object-Based Databases

Traditional database applications consist of data-processing tasks, such as banking and payroll management, with relatively simple data types that are well suited to the relational data model. In particular, tables that are in 1NF. As database systems were applied to a wider range of applications, such as computer-aided design and geographical information systems, limitations imposed by the relational model emerged as an obstacle. The solution was the introduction of more complex data types—tables that are not in 1NF, array and multiset types, and object-based databases.

29.1 Complex Data Types

Traditional database applications have conceptually simple data types. The basic data items are records that are fairly small and whose fields are atomic—that is, they are not further structured, and first normal form holds (see Chapter 7). Further, there are only a few record types.

In recent years, demand has grown for ways to deal with more **complex data types**. Consider, for example, addresses. While an entire address could be viewed as an atomic data item of type string, this view would hide details such as the street address, city, state, and postal code, which could be of interest to queries. On the other hand, if an address were represented by breaking it into the components (street address, city, state, and postal code), writing queries would be more complicated since they would have to mention each field. A better alternative is to allow **structured data types** that allow a type *address* with subparts *street_address, city, state*, and *postal_code*.

As another example, consider multivalued attributes from the E-R model. Such attributes are natural, for example, for representing phone numbers, since people may have more than one phone. The alternative of normalization by creating a new relation is expensive and artificial for this example.

With complex type systems we can represent E-R model concepts, such as composite attributes, multivalued attributes, generalization, and specialization directly, without a complex translation to the relational model.

In Chapter 7, we defined *first normal form* (1NF), which requires that all attributes have *atomic domains*. Recall that a domain is *atomic* if elements of the domain are considered to be indivisible units.

The assumption of 1NF is a natural one in the database application examples we have considered. However, not all applications are best modeled by 1NF relations. For example, rather than view a database as a set of records, users of certain applications view it as a set of objects (or entities). These objects may require several records for their representation. A simple, easy-to-use interface requires a one-to-one correspondence between the user's intuitive notion of an object and the database system's notion of a data item.

Consider, for example, a library application, and suppose we wish to store the following information for each book:

- Book title
- List of authors
- Publisher
- Set of keywords

We can see that, if we define a relation for the preceding information, several domains will be non atomic.

- **Authors**. A book may have a list of authors, which we can represent as an array. Nevertheless, we may want to find all books of which Jones was one of the authors. Thus, we are interested in a subpart of the domain element "authors."
- **Keywords**. If we store a set of keywords for a book, we expect to be able to retrieve all books whose keywords include one or more specified keywords. Thus, we view the domain of the set of keywords as non atomic.
- **Publisher**. Unlike *keywords* and *authors*, *publisher* does not have a set-valued domain. However, we may view *publisher* as consisting of the subfields *name* and *branch*. This view makes the domain of *publisher* non atomic.

Figure 29.1 shows an example relation, *books*.

title	*author_array*	*publisher*	*keyword_set*
		(*name, branch*)	
Compilers	[Smith, Jones]	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	[Jones, Frick]	(Oxford, London)	{Internet, Web }

Figure 29.1 Non-1NF books relation, *books*.

title	*author*	*position*
Compilers	Smith	1
Compilers	Jones	2
Networks	Jones	1
Networks	Frick	2

authors

title	*keyword*
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

title	*pub_name*	*pub_branch*
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Figure 29.2 4NF version of the relation *books*.

For simplicity, we assume that the title of a book uniquely identifies the book.[1] We can then represent the same information using the following schema, where the primary key attributes are underlined:

- *authors*(*title, author, position*)
- *keywords*(*title, keyword*)
- *books4*(*title, pub_name, pub_branch*)

The above schema satisfies 4NF. Figure 29.2 shows the normalized representation of the data from Figure 29.1.

Although our example book database can be adequately expressed without using **nested relations**, the use of nested relations leads to an easier-to-understand model. The typical user or programmer of an information-retrieval system thinks of the database in terms of books having sets of authors, as the non-1NF design models. The 4NF design requires queries to join multiple relations, whereas the non-1NF design makes many types of queries easier.

[1]This assumption does not hold in the real world. Books are usually identified by a 10-digit ISBN number that uniquely identifies each published book.

On the other hand, it may be better to use a first normal form representation in other situations. For instance, consider the *takes* relationship in our university example. The relationship is many-to-many between *student* and *section*. We could conceivably store a set of sections with each student, or a set of students with each section, or both. If we store both, we would have data redundancy (the relationship of a particular student to a particular section would be stored twice).

The ability to use complex data types such as sets and arrays can be useful in many applications but should be used with care.

29.2 SQL Extensions to Deal with Complex Data Types

Before SQL:1999, the SQL type system consisted of a fairly simple set of predefined types. SQL:1999 added an extensive type system to SQL, allowing structured types and type inheritance.

Structured types allow composite attributes of E-R designs to be represented directly. For instance, we can define the following structured type to represent a composite attribute *name* with component attribute *firstname* and *lastname*:

```
create type Name as
    (firstname varchar(20),
    lastname varchar(20))
    final;
```

Similarly, the following structured type can be used to represent a composite attribute *address*:

```
create type Address as
    (street varchar(20),
    city varchar(20),
    not final;
```

Such types are called **user-defined** types in SQL.[2] The above definition corresponds to the E-R diagram in Figure 6.7. The **final** and **not final** specifications are related to subtyping, which we describe in Section 29.3.1.[3]

We can now use these types to create composite attributes in a relation, by simply declaring an attribute to be of one of these types. For example, we could create a table *person* as follows:

[2]To illustrate our earlier note about commercial implementations defining their syntax before the standards were developed, we point out that Oracle requires the keyword **object** following **as**.

[3]The **final** specification for *Name* indicates that we cannot create subtypes for *name*, whereas the **not final** specification for *Address* indicates that we can create subtypes of *address*.

```
create table person (
        name Name,
        address Address,
        dateOfBirth date);
```

The components of a composite attribute can be accessed using a "dot" notation; for instance *name.firstname* returns the firstname component of the name attribute. An access to attribute *name* would return a value of the structured type *Name*.

We can also create a table whose rows are of a user-defined type. For example, we could define a type *PersonType* and create the table *person* as follows:[4]

```
create type PersonType as (
        name Name,
        address Address,
        dateOfBirth date)
        not final
create table person of PersonType;
```

An alternative way of defining composite attributes in SQL is to use unnamed **row types**. For instance, the relation representing person information could have been created using row types as follows:

```
create table person_r (
        name row (firstname varchar(20),
                  lastname varchar(20)),
        address row (street varchar(20),
                     city varchar(20),
        dateOfBirth date);
```

This definition is equivalent to the preceding table definition, except that the attributes *name* and *address* have unnamed types, and the rows of the table also have an unnamed type.

The following query illustrates how to access component attributes of a composite attribute. The query finds the last name and city of each person.

```
select name.lastname, address.city
from person;
```

A structured type can have **methods** defined on it. We declare methods as part of the type definition of a structured type:

[4]Most actual systems, being case insensitive, would not permit *name* to be used both as an attribute name and as a data type.

```
create type PersonType as (
        name Name,
        address Address,
        dateOfBirth date)
        not final
method ageOnDate(onDate date)
        returns interval year;
```

We create the method body separately:

```
create instance method ageOnDate (onDate date)
        returns interval year
        for PersonType
begin
        return onDate − self.dateOfBirth;
end
```

Note that the **for** clause indicates which type this method is for, while the keyword **instance** indicates that this method executes on an instance of the *Person* type. The variable **self** refers to the *Person* instance on which the method is invoked. The body of the method can contain procedural statements, which we saw in Section 5.2. Methods can update the attributes of the instance on which they are executed.

Methods can be invoked on instances of a type. If we had created a table *person* of type *PersonType*, we could invoke the method *ageOnDate*( ) as illustrated below, to find the age of each person.

```
select name.lastname, ageOnDate(current_date)
from person;
```

In SQL:1999, **constructor functions** are used to create values of structured types. A function with the same name as a structured type is a constructor function for the structured type. For instance, we could declare a constructor for the type *Name* like this:

```
create function Name (firstname varchar(20), lastname varchar(20))
returns Name
begin
        set self.firstname = firstname;
        set self.lastname = lastname;
end
```

We can then use **new** *Name*('John', 'Smith') to create a value of the type *Name*. We can construct a row value by listing its attributes within parentheses. For instance, if

we declare an attribute *name* as a row type with components *firstname* and *lastname* we can construct this value for it: ('Ted', 'Codd') without using a constructor.

By default, every structured type has a constructor with no arguments, which sets the attributes to their default values. Any other constructors have to be created explicitly. There can be more than one constructor for the same structured type; although they have the same name, they must be distinguishable by the number of arguments and types of their arguments.

The following statement illustrates how we can create a new tuple in the *Person* relation. We assume that a constructor has been defined for *Address*, just like the constructor we defined for *Name*.

```
insert into Person
    values
        (new Name('John', 'Smith'),
        new Address('20 Main St', 'New York', '11001'),
        date '1960-8-22');
```

29.3 Type and Table Inheritance

29.3.1 Type Inheritance

Suppose that we have the following type definition for people:

```
create type Person
    (name varchar(20),
     address varchar(20));
```

We may want to store extra information in the database about people who are students, and about people who are teachers. Since students and teachers are also people, we can use **inheritance** to define the student and teacher types in SQL:

```
create type Student
    under Person
    (degree varchar(20),
     department varchar(20));

create type Teacher
    under Person
    (salary integer,
     department varchar(20));
```

Both *Student* and *Teacher* inherit the attributes of *Person*—namely, *name* and *address*. *Student* and *Teacher* are said to be subtypes of *Person*, and *Person* is a supertype of *Student*, as well as of *Teacher*.

Methods of a structured type are inherited by its subtypes, just as attributes are. However, a subtype can redefine the effect of a method by declaring the method again, using **overriding method** in place of **method** in the method declaration.

The SQL standard requires an extra field at the end of the type definition, whose value is either **final** or **not final**. The keyword **final** says that subtypes may not be created from the given type, while **not final** says that subtypes may be created.

Now suppose that we want to store information about teaching assistants, who are simultaneously students and teachers, perhaps even in different departments. We can do this if the type system supports **multiple inheritance**, where a type is declared as a subtype of multiple types. Note that the SQL standard does not support multiple inheritance, although future versions of the SQL standard may support it, so we discuss the concept here.

For instance, if our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

create type *TeachingAssistant*
under *Student*, *Teacher*;

TeachingAssistant inherits all the attributes of *Student* and *Teacher*. There is a problem, however, since the attributes *name*, *address*, and *department* are present in *Student*, as well as in *Teacher*.

The attributes *name* and *address* are actually inherited from a common source, *Person*. So there is no conflict caused by inheriting them from *Student* as well as *Teacher*. However, the attribute *department* is defined separately in *Student* and *Teacher*. In fact, a teaching assistant may be a student of one department and a teacher in another department. To avoid a conflict between the two occurrences of *department*, we can rename them by using an **as** clause, as in this definition of the type *TeachingAssistant*:

create type *TeachingAssistant*
under *Student* **with** (*department* **as** *student_dept)*,
Teacher **with** (*department* **as** *teacher_dept*);

In SQL, as in most other languages, a value of a structured type must have exactly one *most-specific type.* That is, each value must be associated with one specific type, called its **most-specific type**, when it is created. By means of inheritance, it is also associated with each of the supertypes of its most-specific type. For example, suppose that an entity has the type *Person*, as well as the type *Student*. Then, the most-specific type of the entity is *Student*, since *Student* is a subtype of *Person*. However, an entity cannot have the type *Student* as well as the type *Teacher* unless it has a type, such as *Teachin-*

gAssistant, that is a subtype of *Teacher*, as well as of *Student* (which is not possible in SQL since multiple inheritance is not supported by SQL).

29.3.2 Table Inheritance

Subtables in SQL correspond to the E-R notion of specialization/generalization. For instance, suppose we define the *people* table as follows:

create table *people* **of** *Person*;

We can then define tables *students* and *teachers* as **subtables** of *people*, as follows:

create table *students* **of** *Student*
under *people*;

create table *teachers* **of** *Teacher*
under *people*;

The types of the subtables (*Student* and *Teacher* in the above example) are subtypes of the type of the parent table (*Person* in the above example). As a result, every attribute present in the table *people* is also present in the subtables *students* and *teachers*.

Further, when we declare *students* and *teachers* as subtables of *people*, every tuple present in *students* or *teachers* becomes implicitly present in *people*. Thus, if a query uses the table *people*, it will find not only tuples directly inserted into that table, but also tuples inserted into its subtables, namely *students* and *teachers*. However, only those attributes that are present in *people* can be accessed by that query.

SQL permits us to find tuples that are in *people* but not in its subtables by using "**only** *people*" in place of *people* in a query. The **only** keyword can also be used in delete and update statements. Without the **only** keyword, a delete statement on a supertable, such as *people*, also deletes tuples that were originally inserted in subtables (such as *students*); for example, a statement:

delete from *people* **where** *P*;

would delete all tuples from the table *people*, as well as its subtables *students* and *teachers*, that satisfy *P*. If the **only** keyword is added to the above statement, tuples that were inserted in subtables are not affected, even if they satisfy the **where** clause conditions. Subsequent queries on the supertable would continue to find these tuples.

Conceptually, multiple inheritance is possible with tables, just as it is possible with types. For example, we can create a table of type *TeachingAssistant*:

create table *teaching_assistants*
of *TeachingAssistant*
under *students*, *teachers*;

As a result of the declaration, every tuple present in the *teaching_assistants* table is also implicitly present in the *teachers* and in the *students* table, and in turn in the *people* table. We note, however, that multiple inheritance of tables is not supported by SQL.

There are some consistency requirements for subtables. Before we state the constraints, we need a definition: we say that tuples in a subtable and parent table **correspond** if they have the same values for all inherited attributes. Thus, corresponding tuples represent the same entity.

The consistency requirements for subtables are:

1. Each tuple of the supertable can correspond to at most one tuple in each of its immediate subtables.
2. SQL has an additional constraint that all the tuples corresponding to each other must be derived from one tuple (inserted into one table).

For example, without the first condition, we could have two tuples in *students* (or *teachers*) that correspond to the same person.

The second condition rules out a tuple in *people* corresponding to both a tuple in *students* and a tuple in *teachers*, unless all these tuples are implicitly present because a tuple was inserted in a table *teaching_assistants*, which is a subtable of both *teachers* and *students*.

Since SQL does not support multiple inheritance, the second condition actually prevents a person from being both a teacher and a student. Even if multiple inheritance were supported, the same problem would arise if the subtable *teaching_assistants* were absent. It would be useful to model a situation where a person can be a teacher and a student, even if a common subtable *teaching_assistants* is not present. Thus, it can be useful to remove the second consistency constraint. Doing so would allow an object to have multiple types, without requiring it to have a most-specific type.

For example, suppose we again have the type *Person*, with subtypes *Student* and *Teacher*, and the corresponding table *people*, with subtables *teachers* and *students*. We can then have a tuple in *teachers* and a tuple in *students* corresponding to the same tuple in *people*. There is no need to have a type *TeachingAssistant* that is a subtype of both *Student* and *Teacher*. We need not create a type *TeachingAssistant* unless we wish to store extra attributes or redefine methods in a manner specific to people who are both students and teachers.

We note, however, that SQL unfortunately prohibits such a situation, because of consistency requirement 2. Since SQL also does not support multiple inheritance, we cannot use inheritance to model a situation where a person can be both a student and a teacher. As a result, SQL subtables cannot be used to represent overlapping specializations from the E-R model.

We can of course create separate tables to represent the overlapping specializations/generalizations without using inheritance. The process was described in Section 6.8.6.1. In the above example, we would create tables *people*, *students*, and *teachers*, with

the *students* and *teachers* tables containing the primary-key attribute of *Person* and other attributes specific to *Student* and *Teacher*, respectively. The *people* table would contain information about all persons, including students and teachers. We would then have to add appropriate referential-integrity constraints to ensure that students and teachers are also represented in the *people* table.

In other words, we can create our own improved implementation of the subtable mechanism using existing features of SQL, with some extra effort in defining the table, as well as some extra effort at query time to specify joins to access required attributes.

We note that SQL defines a privilege called **under**, which is required in order to create a subtype or subtable under another type or table. The motivation for this privilege is similar to that for the **references** privilege.

29.4 Array and Multiset Types in SQL

SQL supports two collection types: **arrays** and **multisets**; array types were added in SQL:1999, while multiset types were added in SQL:2003. Recall that a *multiset* is an unordered collection, where an element may occur multiple times. Multisets are like sets, except that a set allows each element to occur at most once.

Suppose we wish to record information about books, including a set of keywords for each book. Suppose also that we wished to store the names of authors of a book as an array; unlike elements in a multiset, the elements of an array are ordered, so we can distinguish the first author from the second author, and so on. The following example illustrates how these array and multiset-valued attributes can be defined in SQL:

```
create type Publisher as
        (name varchar(20),
         branch varchar(20));

create type Book as
        (title varchar(20),
         author_array varchar(20) array [10],
         pub_date date,
         publisher Publisher,
         keyword_set varchar(20) multiset);

create table books of Book;
```

The first statement defines a type called *Publisher* with two components: a name and a branch. The second statement defines a structured type *Book* that contains a *title*, an *author_array*, which is an array of up to 10 author names, a publication date, a publisher (of type *Publisher*), and a multiset of keywords. Finally, a table *books* containing tuples of type *Book* is created.

Note that we used an array, instead of a multiset, to store the names of authors, since the ordering of authors generally has some significance, whereas we believe that the ordering of keywords associated with a book is not significant.

In general, multivalued attributes from an E-R schema can be mapped to multiset-valued attributes in SQL; if ordering is important, SQL arrays can be used instead of multisets.

29.4.1 Creating and Accessing Collection Values

An array of values can be created in SQL:1999 in this way:

array['Silberschatz', 'Korth', 'Sudarshan']

Similarly, a multiset of keywords can be constructed as follows:

multiset['computer', 'database', 'SQL']

Thus, we can create a tuple of the type defined by the *books* relation as:

('Compilers', **array**['Smith', 'Jones'], **new** *Publisher*('McGraw-Hill', 'New York'), **multiset**['parsing', 'analysis'])

Here we have created a value for the attribute *Publisher* by invoking a *constructor* function for *Publisher* with appropriate arguments. Note that this constructor for *Publisher* must be created explicitly and is not present by default; it can be declared just like the constructor for *Name*, which we saw in Section 29.2.

If we want to insert the preceding tuple into the relation *books*, we can execute the statement:

```
insert into books
values ('Compilers', array['Smith', 'Jones'],
                     new Publisher('McGraw-Hill', 'New York'),
                     multiset['parsing', 'analysis']);
```

We can access or update elements of an array by specifying the array index, for example *author_array*[1].

29.4.2 Querying Collection-Valued Attributes

We now consider how to handle collection-valued attributes in queries. An expression evaluating to a collection can appear anywhere that a relation name may appear, such as in a **from** clause, as the following paragraphs illustrate. We use the table *books* that we defined earlier.

If we want to find all books that have the word "database" as one of their keywords, we can use this query:

```
select title
from books
where 'database' in (unnest(keyword_set));
```

Note that we have used **unnest**(*keyword_set*) in a position where SQL without nested relations would have required a **select-from-where** subexpression.

If we know that a particular book has three authors, we could write:

```
select author_array[1], author_array[2], author_array[3]
from books
where title = 'Database System Concepts';
```

Now, suppose that we want a relation containing pairs of the form "title, author_name" for each book and each author of the book. We can use this query:

```
select B.title, A.author
from books as B, unnest(B.author_array) as A(author);
```

Since the *author_array* attribute of *books* is a collection-valued field, **unnest**(*B.author_array*) can be used in a **from** clause, where a relation is expected. Note that the tuple variable *B* is visible to this expression since it is defined *earlier* in the **from** clause.

When unnesting an array, the previous query loses information about the ordering of elements in the array. The **unnest with ordinality** clause can be used to get this information, as illustrated by the following query. This query can be used to generate the *authors* relation, which we saw earlier, from the *books* relation.

```
select title, A.author, A.position
from books as B,
        unnest(B.author_array) with ordinality as A(author, position);
```

The **with ordinality** clause generates an extra attribute which records the position of the element in the array. A similar query, but without the **with ordinality** clause, can be used to generate the *keyword* relation.

29.4.3 Nesting and Unnesting

The transformation of a nested relation into a form with fewer (or no) relation-valued attributes is called **unnesting**. The *books* relation has two attributes, *author_array* and *keyword_set*, that are collections, and two attributes, *title* and *publisher*, that are not. Suppose that we want to convert the relation into a single flat relation, with no nested relations or structured types as attributes. We can use the following query to carry out the task:

title	*author*	*pub_name*	*pub_branch*	*keyword*
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

Figure 29.3 *flat_books*: result of unnesting attributes *author_array* and *keyword_set* of relation *books*.

```
select title, A.author, publisher.name as pub_name, publisher.branch
        as pub_branch, K.keyword
from books as B, unnest(B.author_array) as A(author),
        unnest (B.keyword_set) as K(keyword);
```

The variable *B* in the **from** clause is declared to range over *books*. The variable *A* is declared to range over the authors in *author_array* for the book *B*, and *K* is declared to range over the keywords in the *keyword_set* of the book *B*. Figure 29.1 shows an instance of the *books* relation, and Figure 29.3 shows the relation, which we call *flat_books*, that is the result of the preceding query. Note that the relation *flat_books* is in 1NF, since all its attributes are atomic valued.

The reverse process of transforming a 1NF relation into a nested relation is called **nesting**. Nesting can be carried out by an extension of grouping in SQL. In the normal use of grouping in SQL, a temporary multiset relation is (logically) created for each group, and an aggregate function is applied on the temporary relation to get a single (atomic) value. The **collect** function returns the multiset of values, so instead of creating a single value, we can create a nested relation. Suppose that we are given the 1NF relation *flat_books*, as in Figure 29.3. The following query nests the relation on the attribute *keyword*:

```
select title, author, Publisher(pub_name, pub_branch) as publisher,
        collect(keyword) as keyword_set
from flat_books
group by title, author, publisher;
```

The result of the query on the *flat_books* relation from Figure 29.3 appears in Figure 29.4.

If we want to nest the author attribute also into a multiset, we can use the query:

title	*author*	*publisher* (*pub_name*,*pub_branch*)	*keyword_set*
Compilers	Smith	(McGraw-Hill, NewYork)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, NewYork)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}

Figure 29.4 A partially nested version of the *flat_books* relation.

```
select title, collect(author) as author_set,
       Publisher(pub_name, pub_branch) as publisher,
            collect(keyword) as keyword_set
from flat_books
group by title, publisher;
```

Another approach to creating nested relations is to use subqueries in the **select** clause. An advantage of the subquery approach is that an **order by** clause can be used in the subquery to generate results in the order desired for the creation of an array. The following query illustrates this approach; the keywords **array** and **multiset** specify that an array and multiset (respectively) are to be created from the results of the subqueries.

```
select title,
       array( select author
         from authors as A
         where A.title = B.title
         order by A.position) as author_array,
       Publisher(pub_name, pub_branch) as publisher,
       multiset( select keyword
         from keywords as K
         where K.title = B.title) as keyword_set,
from books4 as B;
```

The system executes the nested subqueries in the **select** clause for each tuple generated by the **from** and **where** clauses of the outer query. Observe that the attribute *B.title* from the outer query is used in the nested queries, to ensure that only the correct sets of authors and keywords are generated for each title.

SQL:2003 provides a variety of operators on multisets, including a function **set**(*M*) that returns a duplicate-free version of a multiset *M*, an **intersection** aggregate operation, which returns the intersection of all the multisets in a group, a **fusion** aggregate operation, which returns the union of all multisets in a group, and a **submultiset** predicate, which checks if a multiset is contained in another multiset.

The SQL standard does not provide any way to update multiset attributes except by assigning a new value. For example, to delete a value v from a multiset attribute A, we would have to set it to (A **except all multiset**[v]).

29.5 Summary

- Collection types include nested relations, sets, multisets, and arrays, and the object-relational model permits attributes of a table to be collections.
- The SQL standard includes extensions of the SQL data-definition and query language to deal with new data types and with object orientation. These include support for collection-valued attributes, inheritance, and tuple references. Such extensions attempt to preserve the relational foundations—in particular, the declarative access to data—while extending the modeling power.

Review Terms

- Nested relations
- Nested relational model
- Complex types
- Collection types
- Sets
- Arrays
- Multisets
- Structured types
- Row types
- Constructors
- Inheritance
  - Single inheritance
  - Multiple inheritance
- Type inheritance
- Most-specific type
- Table inheritance
- Subtable
- Overlapping subtables
- Reference types
- Scope of a reference
- Self-referential attribute
- Path expressions
- Nesting and unnesting
- SQL functions and procedures
- Object-relational mapping

Practice Exercises

29.1 A car-rental company maintains a database for all vehicles in its current fleet. For all vehicles, it includes the vehicle identification number, license number, manufacturer, model, date of purchase, and color. Special data are included for certain types of vehicles:

- Trucks: cargo capacity
- Sports cars: horsepower, renter age requirement
- Vans: number of passengers
- Off-road vehicles: ground clearance, drivetrain (four- or two-wheel drive)

Construct an SQL schema definition for this database. Use inheritance where appropriate.

29.2 Consider a database schema with a relation *Emp* whose attributes are as shown below, with types specified for multivalued attributes.

Emp = (ename, ChildrenSet **multiset***(Children), SkillSet* **multiset***(Skills))*
Children = (name, birthday)
Skills = (type, ExamSet **setof***(Exams))*
Exams = (year, city)

Answer the following:

a. Define the above schema in SQL, with appropriate types for each attribute.

b. Using the above schema, write the following queries in SQL.

i. Find the names of all employees who have a child born on or after January 1, 2000.

ii. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

iii. List all skill types in the relation *Emp*.

29.3 Consider the E-R diagram in Figure 29.5, which contains composite, multivalued, and derived attributes.

a. Give an SQL schema definition corresponding to the E-R diagram.

b. Give constructors for each of the structured types defined above.

29.4 Consider the relational schema shown in Figure 29.6.

a. Give a schema definition in SQL corresponding to the relational schema, but using references to express foreign-key relationships.

b. Write each of the queries below on the schema in Figure 29.6, using SQL.

i. Find the company with the most employees.

ii. Find the company with the smallest payroll.

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

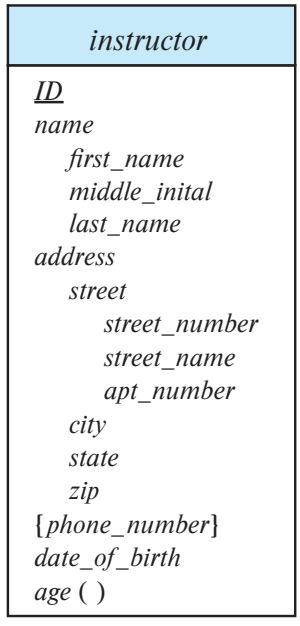

Figure 29.5 E-R diagram with composite, multivalued, and derived attributes.

29.5 Suppose that you have been hired as a consultant to choose a database system for your client's application. For each of the following applications, state what type of database system (relational, persistent programming language-based OODB, object-relational; do not specify a commercial product) you would recommend. Justify your recommendation.

a. A computer-aided design system for a manufacturer of airplanes.

b. A system to track contributions made to candidates for public office.

c. An information system to support the making of movies.

29.6 How does the concept of an object in the object-oriented model differ from the concept of an entity in the entity-relationship model?

---

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

Figure 29.6 Relational database for Exercise 29.4.

Exercises

29.7 Redesign the database of Exercise 29.2 into first normal form and fourth normal form. List any functional or multivalued dependencies that you assume. Also list all referential-integrity constraints that should be present in the first and fourth normal form schemas.

29.8 Consider the schema from Exercise 29.2.

a. Give SQL DDL statements to create a relation *EmpA* which has the same information as *Emp*, but where multiset-valued attributes *ChildrenSet*, *SkillsSet* and *ExamsSet* are replaced by array-valued attributes *ChildrenArray*, *SkillsArray* and *ExamsArray*.

b. Write a query to convert data from the schema of *Emp* to that of *EmpA*, with the array of children sorted by birthday, the array of skills by the skill type, and the array of exams by the year.

c. Write an SQL statement to update the *Emp* relation by adding a child Jeb, with a birthdate of February 5, 2001, to the employee named George.

d. Write an SQL statement to perform the same update as above but on the *EmpA* relation. Make sure that the array of children remains sorted by year.

29.9 Consider the schemas for the table *people*, and the tables *students* and *teachers*, which were created under *people*, in Section 29.3.2. Give a relational schema in third normal form that represents the same information. Recall the constraints on subtables, and give all constraints that must be imposed on the relational schema so that every database instance of the relational schema can also be represented by an instance of the schema with inheritance.

29.10 Explain the distinction between a type x and a reference type **ref**(x). Under what circumstances would you choose to use a reference type?

29.11 Consider the E-R diagram in Figure 29.7, which contains specializations, using subtypes and subtables.

a. Give an SQL schema definition of the E-R diagram.

b. Give an SQL query to find the names of all people who are not secretaries.

c. Give an SQL query to print the names of people who are neither employees nor students.

d. Can you create a person who is an employee and a student with the schema you created? Explain how, or explain why it is not possible.

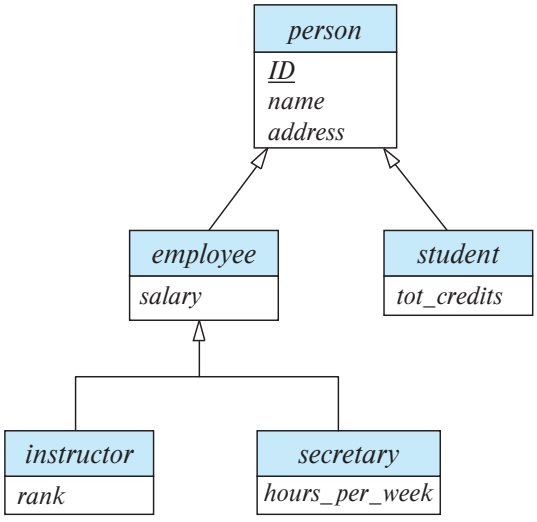

Figure 29.7 Specialization and generalization.

29.12 Suppose a JDO database had an object A, which references object B, which in turn references object C. Assume all objects are on disk initially. Suppose a program first dereferences A, then dereferences B by following the reference from A, and then finally dereferences C. Show the objects that are represented in memory after each dereference, along with their state (hollow or filled, and values in their reference fields).

Tools

There are considerable differences between database products in their support for object-relational features. Oracle probably has the most extensive support among the major database vendors. The Informix database system provides support for many object-relational features. Both Oracle and Informix provided object-relational features before the SQL:1999 standard was finalized, and they have some features that are not part of SQL:1999.

Further Reading

Several object-oriented extensions to SQL have been proposed. POSTGRES ([Stonebraker and Rowe (1986)] and [Stonebraker (1986)]) was an early implementation of an object-relational system.

Bibliography

[Stonebraker (1986)] M. Stonebraker, "Inclusion of New Types in Relational Database Systems", In *Proc. of the International Conf. on Data Engineering* (1986), pages 262–269.

[Stonebraker and Rowe (1986)] M. Stonebraker and L. Rowe, "The Design of POSTGRES", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1986), pages 340–355.

Credits

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

CHAPTER 30

XML

The **Extensible Markup Language** (**XML**) was not designed for database applications. In fact, like the **Hyper-Text Markup Language** (**HTML**) on which the World Wide Web is based, XML has its roots in document management and is derived from a language for structuring large documents known as the **Standard Generalized Markup Language** (**SGML**). However, unlike SGML and HTML, XML is designed to represent data. It is particularly useful as a data format when an application must communicate with another application or integrate information from several other applications. When XML is used in these contexts, many database issues arise, including how to organize, manipulate, and query the XML data. In this chapter, we introduce XML and discuss both the management of XML data with database techniques and the exchange of data formatted as XML documents.

30.1 Motivation

To understand XML, it is important to understand its roots as a document markup language. The term **markup** refers to anything in a document that is not intended to be part of the printed output. For example, a writer creating text that will eventually be typeset in a magazine may want to make notes about how the typesetting should be done. It would be important to type these notes in a way that they could be distinguished from the actual content, so that a note like "set this word in large size, bold font" or "insert a line break here" does not end up printed in the magazine. Such notes convey extra information about the text. In electronic document processing, a **markup language** is a formal description of what part of the document is content, what part is markup, and what the markup means.

Just as database systems evolved from physical file processing to provide a separate logical view, markup languages evolved from specifying instructions for how to print parts of the document to specifying the *function* of the content. For instance, with functional markup, text representing section headings (for this section, the word *Motivation* would be marked up as being a section heading, instead of being marked up as text

to be printed in large size, bold font). From the viewpoint of typesetting, such functional markup allows the document to be formatted differently in different situations. It also helps different parts of a large document, or different pages in a large web site, to be formatted in a uniform manner. More importantly, functional markup also helps record what each part of the text represents semantically, and correspondingly helps automate extraction of key parts of documents.

For the family of markup languages that includes HTML, SGML, and XML, the markup takes the form of **tags** enclosed in angle brackets, <>. Tags are used in pairs, with <tag> and </tag> delimiting the beginning and the end of the portion of the document to which the tag refers. For example, the title of a document might be marked up as follows:

```
<title>Database System Concepts</title>
```

```
<university>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course>
        <course_id> CS-101 </course_id>
        <title> Intro. to Computer Science </title>
        <dept_name> Comp. Sci </dept_name>
        <credits> 4 </credits>
    </course>
    <course>
        <course_id> BIO-301 </course_id>
        <title> Genetics </title>
        <dept_name> Biology </dept_name>
        <credits> 4 </credits>
    </course>
```

continued in Figure Figure 30.2

Figure 30.1 XML representation of (part of) university information.

Unlike HTML, XML does not prescribe the set of tags allowed, and the set may be chosen as needed by each application. This feature is the key to XML's major role in data representation and exchange, whereas HTML is used primarily for document formatting.

For example, in our running university application, department, course and instructor information can be represented as part of an XML document as in Figure 30.1 and Figure 30.2. Observe the use of tags such as department, course, instructor, and teaches. To keep the example short, we use a simplified version of the university

```
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000 </salary>
    </instructor>
    <instructor>
        <IID> 83821 </IID>
        <name> Brandt </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 92000 </salary>
    </instructor>
    <instructor>
        <IID> 76766 </IID>
        <name> Crick </name>
        <dept_name> Biology </dept_name>
        <salary> 72000 </salary>
    </instructor>
    <teaches>
        <IID> 10101 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    <teaches>
        <IID> 83821 </IID>
        <course_id> CS-101 </course_id>
    </teaches>
    <teaches>
        <IID> 76766 </IID>
        <course_id> BIO-301 </course_id>
    </teaches>
</university>
```

Figure 30.2 Continuation of Figure 30.1.

schema that ignores section information for courses. We have also used the tag IID to denote the identifier of the instructor, for reasons we shall see later.

These tags provide context for each value and allow the semantics of the value to be identified. For this example, the XML data representation does not provide any significant benefit over the traditional relational data representation; however, we use this example as our running example because of its simplicity.

Figure 30.3, which shows how information about a purchase order can be represented in XML, illustrates a more realistic use of XML. Purchase orders are typically generated by one organization and sent to another. Traditionally they were printed on paper by the purchaser and sent to the supplier; the data would be manually re-entered

```
<purchase_order>
    <identifier> P-101 </identifier>
    <purchaser>
        <name> Cray Z. Coyote </name>
        <address> Mesa Flats, Route 66, Arizona 12345, USA  </address>
    </purchaser>
    <supplier>
        <name> Acme Supplies  </name>
        <address> 1 Broadway, New York, NY, USA  </address>
    </supplier>
    <itemlist>
        <item>
            <identifier>  RS1  </identifier>
            <description> Atom powered rocket sled  </description>
            <quantity> 2 </quantity>
            <price> 199.95 </price>
        </item>
        <item>
            <identifier>  SG2  </identifier>
            <description> Superb glue  </description>
            <quantity> 1 </quantity>
            <unit-of-measure> liter </unit-of-measure>
            <price> 29.95 </price>
        </item>
    </itemlist>
    <total_cost> 429.85 </total_cost>
    <payment_terms> Cash-on-delivery </payment_terms>
    <shipping_mode> 1-second-delivery </shipping_mode>
</purchaseorder>
```

Figure 30.3 XML representation of a purchase order.

into a computer system by the supplier. This slow process can be greatly sped up by sending the information electronically between the purchaser and supplier. The nested representation allows all information in a purchase order to be represented naturally in a single document. (Real purchase orders have considerably more information than that depicted in this simplified example.) XML provides a standard way of tagging the data; the two organizations must agree on what tags appear in the purchase order, and what they mean.

Compared to storage of data in a relational database, the XML representation may be inefficient, since tag names are repeated throughout the document. However, in spite of this disadvantage, an XML representation has significant advantages when it is used to exchange data between organizations, and for storing complex structured information in files:

- First, the presence of the tags makes the message **self-documenting**; that is, a schema need not be consulted to understand the meaning of the text. We can readily read the fragment in Figure 30.1 and Figure 30.2, for example.
- Second, the format of the document is not rigid. For example, if some sender adds additional information, such as a tag last_accessed noting the last date on which an account was accessed, the recipient of the XML data may simply ignore the tag. As another example, in Figure 30.3, the item with identifier SG2 has a tag called unit-of-measure specified, which the first item does not. The tag is required for items that are ordered by weight or volume and may be omitted for items that are simply ordered by number.

  The ability to recognize and ignore unexpected tags allows the format of the data to evolve over time, without invalidating existing applications. Similarly, the ability to have multiple occurrences of the same tag makes it easy to represent multivalued attributes.
- Third, XML allows nested structures. The purchase order shown in Figure 30.3 illustrates the benefits of having a nested structure. Each purchase order has a purchaser and a list of items as two of its nested structures. Each item in turn has an item identifier, description, and a price nested within it, while the purchaser has a name and address nested within it.

  Such information would have been split into multiple relations in a relational schema. Item information would have been stored in one relation, purchaser information in a second relation, purchase orders in a third, and the relationship between purchase orders, purchasers, and items would have been stored in a fourth relation.

  The relational representation helps to avoid redundancy; for example, item descriptions would be stored only once for each item identifier in a normalized relational schema. In the XML purchase order, however, the descriptions may be repeated in multiple purchase orders that order the same item. However, gathering all information related to a purchase order into a single nested structure, even at

the cost of redundancy, is attractive when information has to be exchanged with external parties.

- Finally, since the XML format is widely accepted, a wide variety of tools are available to assist in its processing, including programming language APIs to create and to read XML data, browser software, and database tools.

We describe several applications for XML data in Section 30.7. Just as SQL is the dominant *language* for querying relational data, XML has become the dominant *format* for data exchange.

30.2 Structure of XML Data

The fundamental construct in an XML document is the **element**. An element is simply a pair of matching start- and end-tags and all the text that appears between them.

XML documents must have a single **root element** that encompasses all other elements in the document. In the example in Figure 30.1, the <university> element forms the root element. Further, elements in an XML document must **nest** properly. For instance,

<course> … <title> … </title> … </course>

is properly nested, whereas

<course> … <title> … </course> … </title>

is not properly nested.

While proper nesting is an intuitive property, we may define it more formally. Text is said to appear **in the context of** an element if it appears between the start-tag and end-

```
…
    <course>
        This course is being offered for the first time in 2009.
        <course_id> BIO-399 </course_id>
        <title> Computational Biology </title>
        <dept_name> Biology </dept_name>
        <credits> 3 </credits>
    </course>
…
```

Figure 30.4 Mixture of text with subelements.

tag of that element. Tags are properly nested if every start-tag has a unique matching end-tag that is in the context of the same parent element.

Note that text may be mixed with the subelements of an element, as in Figure 30.4. As with several other features of XML, this freedom makes more sense in a document-processing context than in a data-processing context, and it is not particularly useful for representing more-structured data such as database content in XML.

```
<university-1>
    <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
        <course>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <credits> 4 </credits>
        </course>
        <course>
            <course_id> CS-347 </course_id>
            <title> Database System Concepts </title>
            <credits> 3 </credits>
        </course>
    </department>
    <department>
        <dept_name> Biology </dept_name>
        <building> Watson </building>
        <budget> 90000 </budget>
        <course>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <credits> 4 </credits>
        </course>
    </department>
    <instructor>
        <IID> 10101 </IID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci. </dept_name>
        <salary> 65000. </salary>
        <course_id> CS-101 </course_id>
    </instructor>
</university-1>
```

Figure 30.5 Nested XML representation of university information.

```
<university-2>
    <instructor>
        <ID> 10101 </ID>
        <name> Srinivasan </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 65000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>

    <instructor>
        <ID> 83821 </ID>
        <name> Brandt </name>
        <dept_name> Comp. Sci.</dept_name>
        <salary> 92000 </salary>
        <teaches>
            <course>
                <course_id> CS-101 </course_id>
                <title> Intro. to Computer Science </title>
                <dept_name> Comp. Sci. </dept_name>
                <credits> 4 </credits>
            </course>
        </teaches>
    </instructor>
</university-2>
```

Figure 30.6 Redundancy in nested XML representation.

The ability to nest elements within other elements provides an alternative way to represent information. Figure 30.5 shows a representation of part of the university information from Figure 30.1, but with course elements nested within department elements. The nested representation makes it easy to find all courses offered by a department. Similarly, identifiers of courses taught by an instructor are nested within the instructor elements. If an instructor teaches more than one course, there would be multiple course_id elements within the corresponding instructor element. Details of

instructors Brandt and Crick are omitted from Figure 30.5 for lack of space but are similar in structure to that for Srinivasan.

Although nested representations are natural in XML, they may lead to redundant storage of data. For example, suppose details of courses taught by an instructor are stored nested within the instructor element as shown in Figure 30.6. If a course is taught by more than one instructor, course information such as title, department, and credits would be stored redundantly with every instructor associated with the course.

Nested representations are widely used in XML data interchange applications to avoid joins. For instance, a purchase order would store the full address of sender and receiver redundantly on multiple purchase orders, whereas a normalized representation may require a join of purchase order records with a *company_address* relation to get address information.

In addition to elements, XML specifies the notion of an **attribute**. For instance, the course identifier of a course can be represented as an attribute, as shown in Figure 30.7. The attributes of an element appear as *name=value* pairs before the closing ">" of a tag. Attributes are strings and do not contain markup. Furthermore, attributes can appear only once in a given tag, unlike subelements, which may be repeated.

Note that in a document construction context, the distinction between subelement and attribute is important—an attribute is implicitly text that does not appear in the printed or displayed document. However, in database and data exchange applications of XML, this distinction is less relevant, and the choice of representing data as an attribute or a subelement is frequently arbitrary. In general, it is advisable to use attributes only to represent identifiers, and to store all other data as subelements.

One final syntactic note is that an element of the form <element></element> that contains no subelements or text can be abbreviated as <element/>; abbreviated elements may, however, contain attributes.

Since XML documents are designed to be exchanged between applications, a **namespace** mechanism has been introduced to allow organizations to specify globally unique names to be used as element tags in documents. The idea of a namespace is to prepend each tag or attribute with a universal resource identifier (e.g., a web address). Thus, for example, if Yale University wanted to ensure that XML documents it created would

```
...
    <course course_id= "CS-101">
        <title> Intro. to Computer Science</title>
        <dept_name> Comp. Sci. </dept_name>
        <credits> 4 </credits>
    </course>
...
```

Figure 30.7 Use of attributes.

```
<university xmlns:yale="http://www.yale.edu">
    ...
    <yale:course>
        <yale:course_id> CS-101 </yale:course_id>
        <yale:title> Intro. to Computer Science</yale:title>
        <yale:dept_name> Comp. Sci. </yale:dept_name>
        <yale:credits> 4 </yale:credits>
    </yale:course>
    ...
</university>
```

Figure 30.8 Unique tag names can be assigned by using namespaces.

not duplicate tags used by any business partner's XML documents, it could prepend a unique identifier with a colon to each tag name. The university may use a web URL such as

http://www.yale.edu

as a unique identifier. Using long unique identifiers in every tag would be rather inconvenient, so the namespace standard provides a way to define an abbreviation for identifiers.

In Figure 30.8, the root element (university) has an attribute xmlns:yale, which declares that yale is defined as an abbreviation for the URL given above. The abbreviation can then be used in various element tags, as illustrated in the figure.

A document can have more than one namespace, declared as part of the root element. Different elements can then be associated with different namespaces. A **default namespace** can be defined by using the attribute xmlns instead of xmlns:yale in the root element. Elements without an explicit namespace prefix would then belong to the default namespace.

Sometimes we need to store values containing tags without having the tags interpreted as XML tags. So that we can do so, XML allows this construct:

<![CDATA[<course> ···</course>]]>

Because it is enclosed within CDATA, the text <course> is treated as normal text data, not as a tag. The term *CDATA* stands for character data.

30.3 XML Document Schema

Databases have schemas, which are used to constrain what information can be stored in the database and to constrain the data types of the stored information. In contrast,

```
<!DOCTYPE university [
    <!ELEMENT university ( (department|course|instructor|teaches)+)>
    <!ELEMENT department ( dept_name, building, budget)>
    <!ELEMENT course ( course_id, title, dept_name, credits)>
    <!ELEMENT instructor (IID, name, dept_name, salary)>
    <!ELEMENT teaches (IID, course_id)>
    <!ELEMENT dept_name( #PCDATA )>
    <!ELEMENT building( #PCDATA )>
    <!ELEMENT budget( #PCDATA )>
    <!ELEMENT course_id ( #PCDATA )>
    <!ELEMENT title ( #PCDATA )>
    <!ELEMENT credits( #PCDATA )>
    <!ELEMENT IID( #PCDATA )>
    <!ELEMENT name( #PCDATA )>
    <!ELEMENT salary( #PCDATA )>
] >
```

Figure 30.9 Example of a DTD.

by default, XML documents can be created without any associated schema: an element may then have any subelement or attribute. While such freedom may occasionally be acceptable given the self-describing nature of the data format, it is not generally useful when XML documents must be processed automatically as part of an application, or even when large amounts of related data are to be formatted in XML.

Here, we describe the first **schema-definition** language included as part of the XML standard, the *document type definition*, as well as its more recently defined replacement, *XML Schema*. Another XML schema-definition language called Relax NG is also in use, but we do not cover it here; for more information on Relax NG, see the references in the bibliographical notes section.

30.3.1 Document Type Definition

The **document type definition** (**DTD**) is an optional part of an XML document. The main purpose of a DTD is much like that of a schema: to constrain and type the information present in the document. However, the DTD does not in fact constrain types in the sense of basic types like integer or string. Instead, it constrains only the appearance of subelements and attributes within an element. The DTD is primarily a list of rules for what pattern of subelements may appear within an element. Figure 30.9 shows a part of an example DTD for a university information document; the XML document in Figure 30.1 conforms to this DTD.

Each declaration is in the form of a regular expression for the subelements of an element. Thus, in the DTD in Figure 30.9, a university element consists of one or more course, department, or instructor elements; the | operator specifies "or" while the +

operator specifies "one or more." Although not shown here, the ∗ operator is used to specify "zero or more," while the ? operator is used to specify an optional element (i.e., "zero or one").

The course element contains subelements course_id, title, dept_name, and credits (in that order). Similarly, department and instructor have the attributes of their relational schema defined as subelements in the DTD.

Finally, the elements course_id, title, dept_name, credits, building, budget, IID, name, and salary are all declared to be of type #PCDATA. The keyword #PCDATA indicates text data; it derives its name, historically, from "parsed character data." Two other special type declarations are empty, which says that the element has no contents, and any, which says that there is no constraint on the subelements of the element; that is, any elements, even those not mentioned in the DTD, can occur as subelements of the element. The absence of a declaration for an element is equivalent to explicitly declaring the type as any.

The allowable attributes for each element are also declared in the DTD. Unlike subelements, no order is imposed on attributes. Attributes may be specified to be of type CDATA, ID, IDREF, or IDREFS; the type CDATA simply says that the attribute contains character data, while the other three are not so simple; they are explained in more detail shortly. For instance, the following line from a DTD specifies that element course has an attribute of type course_id, and a value must be present for this attribute:

```
<!ATTLIST course course_id CDATA #REQUIRED>
```

Attributes must have a type declaration and a default declaration. The default declaration can consist of a default value for the attribute or #REQUIRED, meaning that a value must be specified for the attribute in each element, or #IMPLIED, meaning that no default value has been provided, and the document may omit this attribute. If an attribute has a default value, for every element that does not specify a value for the attribute, the default value is filled in automatically when the XML document is read.

An attribute of type ID provides a unique identifier for the element; a value that occurs in an ID attribute of an element must not occur in any other element in the same document. At most one attribute of an element is permitted to be of type ID. (We renamed the attribute *ID* of the *instructor* relation to IID in the XML representation, in order to avoid confusion with the type ID.)

An attribute of type IDREF is a reference to an element; the attribute must contain a value that appears in the ID attribute of some element in the document. The type IDREFS allows a list of references, separated by spaces.

Figure 30.10 shows an example DTD in which identifiers of course, department, and instructor are represented by ID attributes, and relationships between them are represented by IDREF and IDREFS attributes. The course elements use course_id as their identifier attribute; to do so, course_id has been made an attribute of course instead of a subelement. Additionally, each course element also contains an IDREF of the department corresponding to the course and an IDREFS attribute instructors identifying

```
<!DOCTYPE university-3 [
    <!ELEMENT university ( (department|course|instructor)+)>
    <!ELEMENT department ( building, budget )>
    <!ATTLIST department
        dept_name ID #REQUIRED >
    <!ELEMENT course (title, credits )>
    <!ATTLIST course
        course_id ID #REQUIRED
        dept_name IDREF #REQUIRED
        instructors IDREFS #IMPLIED >
    <!ELEMENT instructor ( name, salary )>
    <!ATTLIST instructor
        IID ID #REQUIRED
        dept_name IDREF #REQUIRED >
    ··· declarations for title, credits, building,
        budget, name and salary ···
] >
```

Figure 30.10 DTD with ID and IDREFS attribute types.

the instructors who teach the course. The department elements have an identifier attribute called dept_name. The instructor elements have an identifier attribute called IID and an IDREF attribute dept_name identifying the department to which the instructor belongs.

Figure 30.11 shows an example XML document based on the DTD in Figure 30.10.

The ID and IDREF attributes serve the same role as reference mechanisms in object-oriented and object-relational databases, permitting the construction of complex data relationships.

Document type definitions are strongly connected to the document formatting heritage of XML. Because of this, they are unsuitable in many ways for serving as the type structure of XML for data-processing applications. Nevertheless, a number of data exchange formats have been defined in terms of DTDs, since they were part of the original standard. Here are some of the limitations of DTDs as a schema mechanism:

- Individual text elements and attributes cannot be typed further. For instance, the element balance cannot be constrained to be a positive number. The lack of such constraints is problematic for data processing and exchange applications, which must then contain code to verify the types of elements and attributes.
- It is difficult to use the DTD mechanism to specify unordered sets of subelements. Order is seldom important for data exchange (unlike document layout, where it is crucial). While the combination of alternation (the | operation) and the $*$ or the

```
<university-3>
    <department dept_name="Comp. Sci.">
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept_name="Biology">
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course_id="CS-101" dept_name="Comp. Sci"
                                instructors="10101 83821">
        <title> Intro. to Computer Science </title>
        <credits> 4 </credits>
    </course>
    <course course_id="BIO-301" dept_name="Biology"
                                    instructors="76766">
        <title> Genetics </title>
        <credits> 4 </credits>
    </course>
    <instructor IID="10101" dept_name="Comp. Sci.">
        <name> Srinivasan </name>
        <salary> 65000 </salary>
    </instructor>
    <instructor IID="83821" dept_name="Comp. Sci.">
        <name> Brandt </name>
        <salary> 72000 </salary>
    </instructor>
    <instructor IID="76766" dept_name="Biology">
        <name> Crick </name>
        <salary> 72000 </salary>
    </instructor>
</university-3>
```

Figure 30.11 XML data with ID and IDREF attributes.

+ operation as in Figure 30.9 permits the specification of unordered collections of tags, it is much more difficult to specify that each tag may only appear once.

- There is a lack of typing in IDs and IDREFSs. Thus, there is no way to specify the type of element to which an IDREF or IDREFS attribute should refer. As a result, the DTD in Figure 30.10 does not prevent the "dept_name" attribute of a course element from referring to other courses, even though this makes no sense.

30.3.2 XML Schema

An effort to redress the deficiencies of the DTD mechanism resulted in the development of a more sophisticated schema language, **XML Schema**. We provide a brief overview of XML Schema, and then we list some areas in which it improves DTDs.

XML Schema defines a number of built-in types such as string, integer, decimal date, and boolean. In addition, it allows user-defined types; these may be simple types

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="building" type="xs:string"/>
            <xs:element name="budget" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="course">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="course_id" type="xs:string"/>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="credits" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept_name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

continued in Figure 30.13.

Figure 30.12 XML Schema version of DTD from Figure 30.9.

```
<xs:element name="teaches">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="course_id" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="UniversityType">
    <xs:sequence>
        <xs:element ref="department" minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element ref="course" minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element ref="instructor" minOccurs="0"
                    maxOccurs="unbounded"/>
        <xs:element ref="teaches" minOccurs="0"
                    maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Figure 30.13 Continuation of Figure 30.12.

with added restrictions, or complex types constructed using constructors such as complexType and sequence.

Figure 30.12 and Figure 30.13 show how the DTD in Figure 30.9 can be represented by XML Schema; we describe next XML Schema features illustrated by the figures.

The first thing to note is that schema definitions in XML Schema are themselves specified in XML syntax, using a variety of tags defined by XML Schema. To avoid conflicts with user-defined tags, we prefix the XML Schema tag with the namespace prefix "xs:"; this prefix is associated with the XML Schema namespace by the xmlns:xs specification in the root element:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Note that any namespace prefix could be used in place of xs; thus, we could replace all occurrences of "xs:" in the schema definition with "xsd:" without changing the meaning of the schema definition. All types defined by XML Schema must be prefixed by this namespace prefix.

The first element is the root element university, whose type is specified to be UniversityType, which is declared later. The example then defines the types of elements department, course, instructor, and teaches.

Note that each of these is specified by an element with tag xs:element, whose body contains the type definition.

The type of department is defined to be a complex type, which is further specified to consist of a sequence of elements dept_name, building, and budget. Any type that has either attributes or nested subelements must be specified to be a complex type.

Alternatively, the type of an element can be specified to be a predefined type by the attribute type; observe how the XML Schema types xs:string and xs:decimal are used to constrain the types of data elements such as dept_name and credits.

Finally the example defines the type UniversityType as containing zero or more occurrences of each of department, course, instructor, and teaches. Note the use of ref to specify the occurrence of an element defined earlier. XML Schema can define the minimum and maximum number of occurrences of subelements by using minOccurs and maxOccurs. The default for both minimum and maximum occurrences is 1, so these have to be specified explicitly to allow zero or more department, course, instructor, and teaches elements.

Attributes are specified using the xs:attribute tag. For example, we could have defined dept_name as an attribute by adding

```
<xs:attribute name = "dept_name"/>
```

within the declaration of the department element. Adding the attribute use = "required" to the above attribute specification declares that the attribute must be specified, whereas the default value of use is optional. Attribute specifications would appear directly under the enclosing complexType specification, even if elements are nested within a sequence specification.

We can use the xs:complexType element to create named complex types; the syntax is the same as that used for the xs:complexType element in Figure 30.12, except that we add an attribute name = typeName to the xs:complexType element, where typeName is the name we wish to give to the type. We can then use the named type to specify the type of an element using the type attribute, just as we used xs:decimal and xs:string in our example.

In addition to defining types, a relational schema also allows the specification of constraints. XML Schema allows the specification of keys and key references, corresponding to the primary-key and foreign-key definition in SQL. In SQL, a primary-key constraint or unique constraint ensures that the attribute values do not recur within the relation. In the context of XML, we need to specify a scope within which values are unique and form a key. The selector is a path expression that defines the scope for the constraint, and field declarations specify the elements or attributes that form the key.[1] To specify that dept_name forms a key for department elements under the root university element, we add the following constraint specification to the schema definition:

[1] We use simple path expressions here that are in a familiar syntax. XML has a rich syntax for path expressions, called XPath, which we explore in Section 30.4.2.

```
<xs:key name = "deptKey">
    <xs:selector xpath = "/university/department"/>
    <xs:field xpath = "dept_name"/>
</xs:key>
```

Correspondingly a foreign-key constraint from course to department may be defined as follows:

```
<xs: name = "courseDeptFKey" refer="deptKey">
    <xs:selector xpath = "/university/course"/>
    <xs:field xpath = "dept_name"/>
</xs:keyref>
```

Note that the refer attribute specifies the name of the key declaration that is being referenced, while the field specification identifies the referring attributes.

XML Schema offers several benefits over DTDs and is widely used today. Among the benefits that we have seen in the preceding examples are these:

- It allows the text that appears in elements to be constrained to specific types, such as numeric types in specific formats or complex types such as sequences of elements of other types.
- It allows user-defined types to be created.
- It allows uniqueness and foreign-key constraints.
- It is integrated with namespaces to allow different parts of a document to conform to different schemas.

In addition to the features we have seen, XML Schema supports several other features that DTDs do not, such as these:

- It allows types to be restricted to create specialized types, for instance by specifying minimum and maximum values.
- It allows complex types to be extended by using a form of inheritance.

Our description of XML Schema is just an overview; to learn more about XML Schema, see the references in the bibliographical notes.

30.4 Querying and Transformation

Given the increasing number of applications that use XML to exchange, mediate, and store data, tools for effective management of XML data are becoming increasingly important. In particular, tools for querying and transformation of XML data are essential

to extract information from large bodies of XML data and to convert data between different representations (schemas) in XML. Just as the output of a relational query is a relation, the output of an XML query can be an XML document. As a result, querying and transformation can be combined into a single tool.

In this section, we describe the XPath and XQuery languages:

- **XPath** is a language for path expressions and is actually a building block for XQuery.
- **XQuery** is the standard language for querying XML data. It is modeled after SQL but is significantly different, since it has to deal with nested XML data. XQuery also incorporates XPath expressions.

The XSLT language is another language designed for transforming XML. However, it is used primarily in document-formatting applications, rather in data-management applications, so we do not discuss it in this book.

The tools section at the end of this chapter provides references to software that can be used to execute queries written in XPath and XQuery.

30.4.1 Tree Model of XML

A **tree model** of XML data are used in all these languages. An XML document is modeled as a **tree**, with **nodes** corresponding to elements and attributes. Element nodes can have child nodes, which can be subelements or attributes of the element. Correspondingly, each node (whether attribute or element), other than the root element, has a parent node, which is an element. The order of elements and attributes in the XML document is modeled by the ordering of children of nodes of the tree. The terms *parent*, *child*, *ancestor*, *descendant*, and *siblings* are used in the tree model of XML data.

The text content of an element can be modeled as a text-node child of the element. Elements containing text broken up by intervening subelements can have multiple text-node children. For instance, an element containing "this is a <bold> wonderful </bold> book" would have a subelement child corresponding to the element bold and two text node children corresponding to "this is a" and "book." Since such structures are not commonly used in data representation, we shall assume that elements do not contain both text and subelements.

30.4.2 XPath

XPath addresses parts of an XML document by means of path expressions. The language can be viewed as an extension of the simple path expressions in object-oriented and object-relational databases. The current version of the XPath standard is XPath 2.0, and our description is based on this version.

A **path expression** in XPath is a sequence of location steps separated by "/" (instead of the "." operator that separates location steps in SQL). The result of a path expression is a set of nodes. For instance, on the document in Figure 30.11, the XPath expression

```
/university-3/instructor/name
```

returns these elements:

```
<name>Srinivasan</name>
<name>Brandt</name>
```

The expression

```
/university-3/instructor/name/text()
```

returns the same names, but without the enclosing tags.

Path expressions are evaluated from left to right. Like a directory hierarchy, the initial '/' indicates the root of the document. Note that this is an abstract root "above" <university-3> that is the document tag.

As a path expression is evaluated, the result of the path at any point consists of an ordered set of nodes from the document. Initially, the "current" set of elements contains only one node, the abstract root. When the next step in a path expression is an element name, such as instructor, the result of the step consists of the nodes corresponding to elements of the specified name that are children of elements in the current element set. These nodes then become the current element set for the next step of the path expression evaluation. Thus, the expression

```
/university-3
```

returns a single node corresponding to the

```
<university-3>
```

tag, while

```
/university-3/instructor
```

returns the two nodes corresponding to the

```
instructor
```

elements that are children of the

```
university-3
```

node.

The result of a path expression is then the set of nodes after the last step of path expression evaluation. The nodes returned by each step appear in the same order as their appearance in the document.

Since multiple children can have the same name, the number of nodes in the node set can increase or decrease with each step. Attribute values may also be accessed, using the "@" symbol. For instance, /university-3/course/@course_id returns a set of all values of course_id attributes of course elements. By default, IDREF links are not followed; we shall see how to deal with IDREFs later.

XPath supports a number of other features:

- Selection predicates may follow any step in a path and are contained in square brackets. For example,

  /university-3/course[credits >= 4]

  returns course elements with a credits value greater than or equal to 4, while

  /university-3/course[credits >= 4]/@course_id

  returns the course identifiers of those courses.

  We can test the existence of a subelement by listing it without any comparison operation; for instance, if we removed just ">= 4" from the above, the expression would return course identifiers of all courses that have a credits subelement, regardless of its value.

- XPath provides several functions that can be used as part of predicates, including testing the position of the current node in the sibling order and the aggregate function count(), which counts the number of nodes matched by the expression to which it is applied. For example, on the XML representation in Figure 30.6, the path expression

  /university-2/instructor[count(./teaches/course)> 2]

  returns instructors who teach more than two courses. Boolean connectives and and or can be used in predicates, while the function not(…) can be used for negation.

- The function id("foo") returns the node (if any) with an attribute of type ID and value "foo". The function id can even be applied on sets of references, or even strings containing multiple references separated by blanks, such as IDREFS. For instance, the path

  /university-3/course/id(@dept_name)

returns all department elements referred to from the dept_name attribute of course elements, while

```
/university-3/course/id(@instructors)
```

returns the instructor elements referred to in the instructors attribute of course elements.

- The | operator allows expression results to be unioned. For example, given data using the schema from Figure 30.11, we could find the union of Computer Science and Biology courses using the expression:

```
/university-3/course[@dept_name="Comp. Sci"]  |
    /university-3/course[@dept_name="Biology"]
```

However, the | operator cannot be nested inside other operators. It is also worth noting that the nodes in the union are returned in the order in which they appear in the document.

- An XPath expression can skip multiple levels of nodes by using "//". For instance, the expression /university-3//name finds all name elements *anywhere* under the /university-3 element, regardless of the elements in which they are contained, and regardless of how many levels of enclosing elements are present between the university-3 and name elements. This example illustrates the ability to find required data without full knowledge of the schema.

- A step in the path need not just select from the children of the nodes in the current node set. In fact, this is just one of several directions along which a step in the path may proceed, such as parents, siblings, ancestors, and descendants. We omit details, but note that "//", described above, is a short form for specifying "all descendants," while ".." specifies the parent.

- The built-in function doc(name) returns the root of a named document; the name could be a file name or a URL. The root returned by the function can then be used in a path expression to access the contents of the document. Thus, a path expression can be applied on a specified document, instead of being applied on the current default document.

  For example, if the university data in our university example are contained in a file "university.xml", the following path expression would return all departments at the university:

```
doc("university.xml")/university/department
```

The function collection(name) is similar to doc but returns a collection of documents identified by name. The function collection can be used, for example, to

open an XML database, which can be viewed as a collection of documents; the following element in the XPath expression would select the appropriate document(s) from the collection.

In most of our examples, we assume that the expressions are evaluated in the context of a database, which implicitly provides a collection of "documents" on which XPath expressions are evaluated. In such cases, we do not need to use the functions doc and collection.

30.4.3 XQuery

The World Wide Web Consortium (W3C) has developed XQuery as the standard query language for XML. Our discussion is based on XQuery 1.0, which was released as a W3C recommendation on 23 January 2007.

30.4.3.1 FLWOR Expressions

XQuery queries are modeled after SQL queries but differ significantly from SQL. They are organized into five sections: **for**, **let**, **where**, **order by**, and **return**. They are referred to as "FLWOR" (pronounced "flower") expressions, with the letters in FLWOR denoting the five sections.

A simple FLWOR expression that returns course identifiers of courses with greater than three credits, shown below, is based on the XML document of Figure 30.11, which uses ID and IDREFS:

```
for $x in /university-3/course
let $courseId := $x/@course_id
where $x/credits > 3
return <course_id> { $courseId } </course_id>
```

The **for** clause is like the **from** clause of SQL and specifies variables that range over the results of XPath expressions. When more than one variable is specified, the results include the Cartesian product of the possible values the variables can take, just as the SQL **from** clause does.

The **let** clause simply allows the results of XPath expressions to be assigned to variable names for simplicity of representation. The **where** clause, like the SQL **where** clause, performs additional tests on the joined tuples from the **for** clause. The **order by** clause, like the SQL **order by** clause, allows sorting of the output. Finally, the **return** clause allows the construction of results in XML.

A FLWOR query need not contain all the clauses; for example a query may contain just the **for** and **return** clauses, and omit the **let**, **where**, and **order by** clauses. The preceding XQuery query did not contain an **order by** clause. In fact, since this query is simple, we can easily do away with the **let** clause, and the variable $courseId in the **return** clause could be replaced with $x/@course_id. Note further that, since the **for** clause uses XPath expressions, selections may occur within the XPath expression. Thus, an equivalent query may have only **for** and **return** clauses:

```
for $x in /university-3/course[credits > 3]
return <course_id> { $x/@course_id } </course_id>
```

However, the **let** clause helps simplify complex queries. Note also that variables assigned by **let** clauses may contain sequences with multiple elements or values, if the path expression on the right-hand side returns a sequence of multiple elements or values.

Observe the use of curly brackets ("{}") in the **return** clause. When XQuery finds an element such as <course_id> starting an expression, it treats its contents as regular XML text, except for portions enclosed within curly brackets, which are evaluated as expressions. Thus, if we omitted the curly brackets in the above **return** clause, the result would contain several copies of the string "$x/@course_id" each enclosed in a course_id tag. The contents within the curly brackets are, however, treated as expressions to be evaluated. Note that this convention applies even if the curly brackets appear within quotes. Thus, we could modify the preceding query to return an element with tag course, with the course identifier as an attribute, by replacing the **return** clause with the following:

```
return <course course_id="{$x/@course_id}" />
```

XQuery provides another way of constructing elements using the **element** and **attribute** constructors. For example, if the **return** clause in the previous query is replaced by the following **return** clause, the query would return course elements with course_id and dept_name as attributes and title and credits as subelements.

```
return element course {
        attribute course_id {$x/@course_id},
        attribute dept_name {$x/dept_name},
        element title {$x/title},
        element credits {$x/credits}
}
```

Note that, as before, the curly brackets are required to treat a string as an expression to be evaluated.

30.4.3.2 Joins

Joins are specified in XQuery much as they are in SQL. The join of course, instructor, and teaches elements in Figure 30.1 can be written in XQuery this way:

```
for $c in /university/course,
      $i in /university/instructor,
      $t in /university/teaches
where $c/course_id= $t/course_id
      and $t/IID = $i/IID
return <course_instructor> { $c $i } </course_instructor>
```

The same query can be expressed with the selections specified as XPath selections:

```
for $c in /university/course,
      $i in /university/instructor,
      $t in /university/teaches[ $c/course_id= $t/course_id
           and $t/IID = $i/IID]
return <course_instructor> { $c $i } </course_instructor>
```

Path expressions in XQuery are the same as path expressions in XPath 2.0. Path expressions may return a single value or element, or a sequence of values or elements. In the absence of schema information, it may not be possible to infer whether a path expression returns a single value or a sequence of values. Such path expressions may participate in comparison operations such as =, <, and >=.

XQuery has an interesting definition of comparison operations on sequences. For example, the expression `$x/credits > 3` would have the usual interpretation if the result of `$x/credits` is a single value, but if the result is a sequence containing multiple values, the expression evaluates to true if at least one of the values is greater than 3. Similarly, the expression `$x/credits = $y/credits` evaluates to true if any one of the values returned by the first expression is equal to any one of the values returned by the second expression. If this behavior is not appropriate, the operators `eq, ne, lt, gt, le, ge` can be used instead. These raise an error if either of their inputs is a sequence with multiple values.

30.4.3.3 Nested Queries

XQuery FLWOR expressions can be nested in the **return** clause in order to generate element nestings that do not appear in the source document. For instance, the XML structure shown in Figure 30.5, with course elements nested within department elements, can be generated from the structure in Figure 30.1 by the query shown in Figure 30.14.

The query also introduces the syntax `$d/*`, which refers to all the children of the node (or sequence of nodes) bound to the variable $d. Similarly, `$d/text()` gives the text content of an element, without the tags.

XQuery provides a variety of aggregate functions such as `sum()` and `count()` that can be applied on sequences of elements or values. The function `distinct-values()` applied on a sequence returns a sequence without duplication. The sequence (collection)

```
<university-1>
{
    for $d in /university/department
    return
        <department>
            { $d/* }
            { for $c in /university/course[dept_name = $d/dept_name]
              return $c }
        </department>
}
{
    for $i in /university/instructor
    return
        <instructor>
            { $i/* }
            { for $c in /university/teaches[IID = $i/IID]
              return $c/course_id }
        </instructor>
}
</university-1>
```

Figure 30.14 Creating nested structures in XQuery.

of values returned by a path expression may have some values repeated because they are repeated in the document, although an XPath expression result can contain at most one occurrence of each node in the document. XQuery supports many other functions; see the references in the bibliographical notes for more information. These functions are actually common to XPath 2.0 and XQuery and can be used in any XPath path expression.

To avoid namespace conflicts, functions are associated with a namespace:

http://www.w3.org/2005/xpath-functions

which has a default namespace prefix of fn. Thus, these functions can be referred to unambiguously as fn:sum or fn:count.

While XQuery does not provide a **group by** construct, aggregate queries can be written by using the aggregate functions on path or FLWOR expressions nested within the **return** clause. For example, the following query on the university XML schema finds the total salary of all instructors in each department:

```
for $d in /university/department
return
    <department-total-salary>
        <dept_name> { $d/dept_name } </dept_name>
        <total_salary> { fn:sum(
            for $i in /university/instructor[dept_name = $d/dept_name]
            return $i/salary
        ) } </total_salary>
    </department-total-salary>
```

30.4.3.4 Sorting of Results

Results can be sorted in XQuery by using the **order by** clause. For instance, this query outputs all instructor elements sorted by the name subelement:

```
for $i in /university/instructor
order by $i/name
return <instructor> { $i/* } </instructor>
```

To sort in descending order, we can use **order by** $i/name **descending**.

Sorting can be done at multiple levels of nesting. For instance, we can get a nested representation of university information with departments sorted in department name order, with courses sorted by course identifiers, as follows:

```
<university-1> {
    for $d in /university/department
    order by $d/dept_name
    return
        <department>
            { $d/* }
            { for $c in /university/course[dept_name = $d/dept_name]
            order by $c/course_id
            return <course> { $c/* } </course> }
        </department>
} </university-1>
```

30.4.3.5 Functions and Types

XQuery provides a variety of built-in functions, such as numeric functions and string matching and manipulation functions. In addition, XQuery supports user-defined functions. The following user-defined function takes as input an instructor identifier and returns a list of all courses offered by the department to which the instructor belongs:

```
declare function local:dept_courses($iid as xs:string) as element(course)* {
    for $i in /university/instructor[IID = $iid],
        $c in /university/courses[dept_name = $i/dept_name]
    return $c
}
```

The namespace prefix xs: used in the above example is predefined by XQuery to be associated with the XML Schema namespace, while the namespace local: is predefined to be associated with XQuery local functions.

The type specifications for function arguments and return values are optional, and may be omitted. XQuery uses the type system of XML Schema. The type element allows elements with any tag, while element(course) allows elements with the tag course. Types can be suffixed with a * to indicate a sequence of values of that type; for example, the definition of function dept_courses specifies the return value as a sequence of course elements.

The following query, which illustrates function invocation, prints out the department courses for the instructor(s) named Srinivasan:

```
for $i in /university/instructor[name = "Srinivasan"],
return local:inst_dept_courses($i/IID)
```

XQuery performs type conversion automatically whenever required. For example, if a numeric value represented by a string is compared to a numeric type, type conversion from string to the numeric type is done automatically. When an element is passed to a function that expects a string value, type conversion to a string is done by concatenating all the text values contained (nested) within the element. Thus, the function contains(a,b), which checks if string a contains string b, can be used with its first argument set to an element, in which case it checks if the element a contains the string b nested anywhere inside it. XQuery also provides functions to convert between types. For instance, number(x) converts a string to a number.

30.4.3.6 Other Features

XQuery offers a variety of other features, such as if-then-else constructs that can be used within **return** clauses, and existential and universal quantification that can be used in predicates in **where** clauses. For example, existential quantification can be expressed in the **where** clause by using:

```
some $e in path satisfies P
```

where path is a path expression and P is a predicate that can use $e. Universal quantification can be expressed by using **every** in place of **some**.

For example, to find departments where every instructor has a salary greater than $50,000, we can use the following query:

```
for $d in /university/department
where every $i in /university/instructor[dept_name=$d/dept_name]
        satisfies $i/salary > 50000
return $d
```

Note, however, that if a department has no instructor, it will trivially satisfy the above condition. An extra clause:

```
and fn:exists(/university/instructor[dept_name=$d/dept_name])
```

can be used to ensure that there is at least one instructor in the department. The built-in function exists() used in the clause returns true if its input argument is nonempty.

The XQJ standard provides an API to submit XQuery queries to an XML database system and to retrieve the XML results. Its functionality is similar to the JDBC API.

30.5 Application Program Interfaces to XML

With the wide acceptance of XML as a data representation and exchange format, software tools are widely available for manipulation of XML data. There are two standard models for programmatic manipulation of XML, each available for use with a number of popular programming languages. Both these APIs can be used to parse an XML document and create an in-memory representation of the document. They are used for applications that deal with individual XML documents. Note, however, that they are not suitable for querying large collections of XML data; declarative querying mechanisms such as XPath and XQuery are better suited to this task.

One of the standard APIs for manipulating XML is based on the *document object model* (DOM), which treats XML content as a tree, with each element represented by a node, called a DOMNode. Programs may access parts of the document in a navigational fashion, beginning with the root.

DOM libraries are available for most common programming languages and are even present in web browsers, where they may be used to manipulate the document displayed to the user. We outline here some of the interfaces and methods in the Java API for DOM, to give a flavor of DOM.

- The Java DOM API provides an interface called Node, and interfaces Element and Attribute, which inherit from the Node interface.
- The Node interface provides methods such as getParentNode(), getFirstChild(), and getNextSibling(), to navigate the DOM tree, starting with the root node.
- Subelements of an element can be accessed by name, using getElementsByTagName(name), which returns a list of all child elements with a specified tag name; individual members of the list can be accessed by the method item(i), which returns the *i*th element in the list.

- Attribute values of an element can be accessed by name, using the method getAttribute(name).
- The text value of an element is modeled as a Text node, which is a child of the element node; an element node with no subelements has only one such child node. The method getData() on the Text node returns the text contents.

DOM also provides a variety of functions for updating the document by adding and deleting attribute and element children of a node, setting node values, and so on.

Many more details are required for writing an actual DOM program; see the bibliographical notes for references to further information.

DOM can be used to access XML data stored in databases, and an XML database can be built with DOM as its primary interface for accessing and modifying data. However, the DOM interface does not support any form of declarative querying.

The second commonly used programming interface, the *Simple API for XML* (SAX), is an *event* model designed to provide a common interface between parsers and applications. This API is built on the notion of *event handlers*, which consist of user-specified functions associated with parsing events. Parsing events correspond to the recognition of parts of a document; for example, an event is generated when the start-tag is found for an element, and another event is generated when the end-tag is found. The pieces of a document are always encountered in order from start to finish.

The SAX application developer creates handler functions for each event and registers them. When a document is read in by the SAX parser, as each event occurs, the handler function is called with parameters describing the event (such as element tag or text contents). The handler functions then carry out their task. For example, to construct a tree representing the XML data, the handler functions for an attribute or element start event could add a node (or nodes) to a partially constructed tree. The start- and end-tag event handlers would also have to keep track of the current node in the tree to which new nodes must be attached; the element start event would set the new element as the node that is the point where further child nodes must be attached. The corresponding element end event would set the parent of the node as the current node where further child nodes must be attached.

SAX generally requires more programming effort than DOM, but it helps avoid the overhead of creating a DOM tree in situations where the application needs to create its own data representation. If DOM were used for such applications, there would be unnecessary space and time overhead for constructing the DOM tree.

30.6 Storage of XML Data

Many applications require storage of XML data. One way to store XML data are to store them as documents in a file system, while a second is to build a special-purpose database to store XML data. Another approach is to convert the XML data to a rela-

tional representation and store them in a relational database. Several alternatives for storing XML data are briefly outlined in this section.

30.6.1 Non-relational Data Stores

There are several alternatives for storing XML data in non-relational data-storage systems:

- **Store in flat files.** Since XML is primarily a file format, a natural storage mechanism is simply a flat file. This approach has many of the drawbacks, outlined in Chapter 1, of using file systems as the basis for database applications. In particular, it lacks data isolation, atomicity, concurrent access, and security. However, the wide availability of XML tools that work on file data makes it relatively easy to access and query XML data stored in files. Thus, this storage format may be sufficient for some applications.
- **Create an XML database.** XML databases are databases that use XML as their basic data model. Early XML databases implemented the Document Object Model on a C++-based object-oriented database. This allows much of the object-oriented database infrastructure to be reused, while providing a standard XML interface. The addition of XQuery or other XML query languages provides declarative querying. Other implementations have built the entire XML storage and querying infrastructure on top of a storage manager that provides transactional support.

Although several databases designed specifically to store XML data have been built, building a full-featured database system from the ground up is a very complex task. Such a database must support not only XML data storage and querying but also other database features such as transactions, security, support for data access from clients, and a variety of administration facilities. It makes sense to instead use an existing database system to provide these facilities and implement XML data storage and querying either on top of the relational abstraction or as a layer parallel to the relational abstraction. We study these approaches in Section 30.6.2.

30.6.2 Relational Databases

Since relational databases are widely used in existing applications, there is a great benefit to be had in storing XML data in relational databases, so that the data can be accessed from existing applications.

Converting XML data to relational form is usually straightforward if the data were generated from a relational schema in the first place and XML is used merely as a data exchange format for relational data. However, there are many applications where the XML data are not generated from a relational schema, and translating the data to relational form for storage may not be straightforward. In particular, nested elements and elements that recur (corresponding to set-valued attributes) complicate storage of

XML data in relational format. Several alternative approaches are available, which we describe below.

30.6.2.1 Store as String

Small XML documents can be stored as string (**clob**) values in tuples in a relational database. Large XML documents with the top-level element having many children can be handled by storing each child element as a string in a separate tuple. For instance, the XML data in Figure 30.1 could be stored as a set of tuples in a relation *elements*(*data*), with the attribute *data* of each tuple storing one XML element (department, course, instructor, or teaches) in string form.

While this representation is easy to use, the database system does not know the schema of the stored elements. As a result, it is not possible to query the data directly. In fact, it is not even possible to implement simple selections such as finding all department elements, or finding the department element with department name "Comp. Sci.", without scanning all tuples of the relation and examining the string contents.

A partial solution to this problem is to store different types of elements in different relations and also store the values of some critical elements as attributes of the relation to enable indexing. For instance, in our example, the relations would be *department_elements, course_elements, instructor_elements*, and *teaches_elements*, each with an attribute *data*. Each relation may have extra attributes to store the values of some subelements, such as *dept_name*, *course_id*, or *name*. Thus, a query that requires department elements with a specified department name can be answered efficiently with this representation. Such an approach depends on type information about XML data, such as the DTD of the data.

Some database systems, such as Oracle, support **function indices**, which can help avoid replication of attributes between the XML string and relation attributes. Unlike normal indices, which are on attribute values, function indices can be built on the result of applying user-defined functions on tuples. For instance, a function index can be built on a user-defined function that returns the value of the dept_name subelement of the XML string in a tuple. The index can then be used in the same way as an index on a *dept_name* attribute.

The approaches have the drawback that a large part of the XML information is stored within strings. It is possible to store all the information in relations in one of several ways that we examine next.

30.6.2.2 Tree Representation

Arbitrary XML data can be modeled as a tree and stored using a relation

$$nodes(id, parent\_id, type, label, value)$$

Each element and attribute in the XML data are given a unique identifier. A tuple is inserted in the *nodes* relation for each element and attribute with its identifier (*id*), the

identifier of its parent node (*parent_id*), the type of the node (attribute or element), the name of the element or attribute (*label*), and the text value of the element or attribute (*value*).

If order information of elements and attributes must be preserved, an extra attribute *position* can be added to the *nodes* relation to indicate the relative position of the child among the children of the parent. As an exercise, you can represent the XML data of Figure 30.1 by using this technique.

This representation has the advantage that all XML information can be represented directly in relational form, and many XML queries can be translated into relational queries and executed inside the database system. However, it has the drawback that each element gets broken up into many pieces, and a large number of joins are required to reassemble subelements into an element.

30.6.2.3 Map to Relations

In this approach, XML elements whose schema is known are mapped to relations and attributes. Elements whose schema is unknown are stored as strings or as a tree.

A relation is created for each element type (including subelements) whose schema is known and whose type is a complex type (i.e., contains attributes or subelements). The root element of the document can be ignored in this step if it does not have any attributes. The attributes of the relation are defined as follows:

- All attributes of these elements are stored as string-valued attributes of the relation.
- If a subelement of the element is a simple type (i.e., cannot have attributes or subelements), an attribute is added to the relation to represent the subelement. The type of the relation attribute defaults to a string value, but if the subelement had an XML Schema type, a corresponding SQL type may be used.

  For example, when applied to the element *department* in the schema (DTD or XML Schema) of the data in Figure 30.1, the subelements *dept_name*, *building*, and *budget* of the element *department* all become attributes of a relation *department*. Applying this procedure to the remaining elements, we get back the original relational schema that we have used in earlier chapters.
- Otherwise, a relation is created corresponding to the subelement (using the same rules recursively on its subelements). Further:
  - An identifier attribute is added to the relations representing the element. (The identifier attribute is added only once even if an element has several subelements.)
  - An attribute *parent_id* is added to the relation representing the subelement, storing the identifier of its parent element.
  - If ordering is to be preserved, an attribute *position* is added to the relation representing the subelement.

For example, if we apply the above procedure to the schema corresponding to the data in Figure 30.5, we get the following relations:

department(*id*, *dept_name*, *building*, *budget*)
course(*parent_id*, *course_id*, *dept_name*, *title*, *credits*)

Variants of this approach are possible. For example, the relations corresponding to subelements that can occur at most once can be "flattened" into the parent relation by moving all their attributes into the parent relation. The bibliographical notes provide references to different approaches to represent XML data as relations.

30.6.2.4 Publishing and Shredding XML Data

When XML is used to exchange data between business applications, the data most often originate in relational databases. Data in relational databases must be *published*, that is, converted to XML form, for export to other applications. Incoming data must be *shredded*, that is, converted back from XML to normalized relation form and stored in a relational database. While application code can perform the publishing and shredding operations, the operations are so common that the conversions should be done automatically, without writing application code, where possible. Database vendors have spent a lot of effort to *XML-enable* their database products.

An XML-enabled database supports an automatic mechanism for publishing relational data as XML. The mapping used for publishing data may be simple or complex. A simple relation to XML mapping might create an XML element for every row of a table and make each column in that row a subelement of the XML element. The XML schema in Figure 30.1 can be created from a relational representation of university information, using such a mapping. Such a mapping is straightforward to generate automatically. Such an XML view of relational data can be treated as a *virtual* XML document, and XML queries can be executed against the virtual XML document.

A more complicated mapping would allow nested structures to be created. Extensions of SQL with nested queries in the **select** clause have been developed to allow easy creation of nested XML output. We outline these extensions in Section 30.6.3.

Mappings also have to be defined to shred XML data into a relational representation. For XML data created from a relational representation, the mapping required to shred the data are a straightforward inverse of the mapping used to publish the data. For the general case, a mapping can be generated as outlined in Section 30.6.2.3.

30.6.2.5 Native Storage within a Relational Database

Some relational databases support **native storage** of XML. Such systems store XML data as strings or in more efficient binary representations, without converting the data to relational form. A new data type **xml** is introduced to represent XML data, although the CLOB and BLOB data types may provide the underlying storage mechanism. XML query languages such as XPath and XQuery are supported to query XML data.

A relation with an attribute of type **xml** can be used to store a collection of XML documents; each document is stored as a value of type **xml** in a separate tuple. Special-purpose indices are created to index the XML data.

Several database systems provide native support for XML data. They provide an **xml** data type and allow XQuery queries to be embedded within SQL queries. An XQuery query can be executed on a single XML document and can be embedded within an SQL query to allow it to execute on each of a collection of documents, with each document stored in a separate tuple.

30.6.3 SQL/XML

While XML is used widely for data interchange, structured data are still widely stored in relational databases. There is often a need to convert relational data to XML representation. The SQL/XML standard, developed to meet this need, defines a standard extension of SQL, allowing the creation of nested XML output. The standard has several parts, including a standard way of mapping SQL types to XML Schema types, and a standard way to map relational schemas to XML schemas, as well as SQL query language extensions.

For example, the SQL/XML representation of the *department* relation would have an XML schema with outermost element *department*, with each tuple mapped to an XML element *row*, and each relation attribute mapped to an XML element of the same name (with some conventions to resolve incompatibilities with special characters in names). An entire SQL schema, with multiple relations, can also be mapped to XML in a similar fashion. Figure 30.15 shows the SQL/XML representation of (part of) the *university* data from Figure 30.1, containing the relations *department* and *course*.

SQL/XML adds several operators and aggregate operations to SQL to allow the construction of XML output directly from the extended SQL. The **xmlelement** function can be used to create XML elements, while **xmlattributes** can be used to create attributes, as illustrated by the following query.

```
select xmlelement (name "course",
                   xmlattributes (course_id as course_id, dept_name as dept_name),
                   xmlelement (name "title", title),
                   xmlelement (name "credits", credits))
        from course
```

This query creates an XML element for each course, with the course identifier and department name represented as attributes, and title and credits as subelements. The result would look like the course elements shown in Figure 30.11, but without the instructor attribute. The **xmlattributes** operator creates the XML attribute name using the SQL attribute name, which can be changed using an **as** clause as shown.

The **xmlforest** operator simplifies the construction of XML structures. Its syntax and behavior are similar to those of **xmlattributes**, except that it creates a forest (collection) of subelements, instead of a list of attributes. It takes multiple arguments, creating

```
<university>
    <department>
        <row>
            <dept_name> Comp. Sci. </dept_name>
            <building> Taylor </building>
            <budget> 100000 </budget>
        </row>
        <row>
            <dept_name> Biology </dept_name>
            <building> Watson </building>
            <budget> 90000 </budget>
        </row>
    </department>
    <course>
        <row>
            <course_id> CS-101 </course_id>
            <title> Intro. to Computer Science </title>
            <dept_name> Comp. Sci </dept_name>
            <credits> 4 </credits>
        </row>
        <row>
            <course_id> BIO-301 </course_id>
            <title> Genetics </title>
            <dept_name> Biology </dept_name>
            <credits> 4 </credits>
        </row>
    </course>
</university>
```

Figure 30.15 SQL/XML representation of (part of) university information.

an element for each argument, with the attribute's SQL name used as the XML element name. The **xmlconcat** operator can be used to concatenate elements created by subexpressions into a forest.

When the SQL value used to construct an attribute is null, the attribute is omitted. Null values are omitted when the body of an element is constructed.

SQL/XML also provides an aggregate function **xmlagg** that creates a forest (collection) of XML elements from the collection of values on which it is applied. The following query creates an element for each department with a course, containing as subelements all the courses in that department. Since the query has a clause **group by** *dept_name*, the aggregate function is applied on all courses in each department, creating a sequence of *course_id* elements.

```
select xmlelement (name "department",
                   dept_name,
                   xmlagg (xmlforest(course_id)
                           order by course_id))
from course
group by dept_name
```

SQL/XML allows the sequence created by **xmlagg** to be ordered, as illustrated in the preceding query. See the bibliographical notes for references to more information on SQL/XML.

30.7 XML Applications

We now outline several applications of XML for storing and communicating (exchanging) data and for accessing web services (information resources).

30.7.1 Storing Data with Complex Structure

Many applications need to store data that are structured, but are not easily modeled as relations. Consider, such as, user preferences that must be stored by an application such as a browser. There are usually a large number of fields, such as home page, security settings, language settings, and display settings, that must be recorded. Some of the fields are multivalued, for example, a list of trusted sites, or maybe ordered lists, for example, a list of bookmarks. Applications traditionally used some type of textual representation to store such data. Today, a majority of such applications prefer to store such configuration information in XML format. The ad hoc textual representations used earlier require effort to design and effort to create parsers that can read the file and convert the data into a form that a program can use. The XML representation avoids both these steps.

XML-based representations are now widely used for storing documents, spreadsheet data, and other data that are part of office application packages. The *Open Document Format* (*ODF*), supported by the Open Office software suite as well as other office suites, and the *Office Open XML* (*OOXML*) format, supported by the Microsoft Office suite, are document representation standards based on XML. They are the two most widely used formats for editable document representation.

XML is also used to represent data with complex structure that must be exchanged between different parts of an application. For example, a database system may represent a query execution plan (a relational-algebra expression with extra information on how to execute operations) by using XML. This allows one part of the system to generate the query execution plan and another part to display it, without using a shared data structure. For example, the data may be generated at a server system and sent to a client system where the data are displayed.

30.7.2 Standardized Data Exchange Formats

XML-based standards for representation of data have been developed for a variety of specialized applications, ranging from business applications such as banking and shipping to scientific applications such as chemistry and molecular biology. Some examples:

- The chemical industry needs information about chemicals, such as their molecular structure, and a variety of important properties, such as boiling and melting points, calorific values, and solubility in various solvents. *ChemML* is a standard for representing such information.
- In shipping, carriers of goods and customs and tax officials need shipment records containing detailed information about the goods being shipped, from whom and to where they were sent, to whom and to where they are being shipped, the monetary value of the goods, and so on.
- An online marketplace in which business can buy and sell goods [a so-called business-to-business (B2B) market] requires information such as product catalogs, including detailed product descriptions and price information, product inventories, quotes for a proposed sale, and purchase orders. For example, the *RosettaNet* standards for e-business applications define XML schemas and semantics for representing data as well as standards for message exchange.

Using normalized relational schemas to model such complex data requirements would result in a large number of relations that do not correspond directly to the objects that are being modeled. The relations would often have large numbers of attributes; explicit representation of attribute/element names along with values in XML helps avoid confusion between attributes. Nested element representations help reduce the number of relations that must be represented, as well as the number of joins required to get required information, at the possible cost of redundancy. For instance, in our university example, listing departments with course elements nested within department elements, as in Figure 30.5, results in a format that is more natural for some applications—in particular, for humans to read—than is the normalized representation in Figure 30.1.

30.7.3 Web Services

Applications often require data from outside of the organization, or from another department in the same organization that uses a different database. In many such situations, the outside organization or department is not willing to allow direct access to its database using SQL, but is willing to provide limited forms of information through predefined interfaces.

When the information is to be used directly by a human, organizations provide web-based forms, where users can input values and get back desired information in

HTML form. However, there are many applications where such information needs to be accessed by software programs rather than by end users. Providing the results of a query in XML form is a clear requirement. In addition, it makes sense to specify the input values to the query also in XML format.

In effect, the provider of the information defines procedures whose input and output are both in XML format. The HTTP protocol is used to communicate the input and output information, since it is widely used and can go through firewalls that institutions use to keep out unwanted traffic from the Internet.

The **Simple Object Access Protocol** (**SOAP**) defines a standard for invoking procedures, using XML for representing the procedure input and output. SOAP defines a standard XML schema for representing the name of the procedure and result status indicators such as failure/error indicators. The procedure parameters and results are application-dependent XML data embedded within the SOAP XML headers.

Typically, HTTP is used as the transport protocol for SOAP, but a message-based protocol (such as email over the SMTP protocol) may also be used. The SOAP standard is widely used today. For example, Amazon and Google provide SOAP-based procedures to carry out search and other activities. These procedures can be invoked by other applications that provide higher-level services to users. The SOAP standard is independent of the underlying programming language, and it is possible for a site running one language, such as C#, to invoke a service that runs on a different language, such as Java.

A site providing such a collection of SOAP procedures is called a **web service**. Several standards have been defined to support web services. The **Web Services Description Language** (**WSDL**) is a language used to describe a web service's capabilities. WSDL provides facilities that interface definitions (or function definitions) provide in a traditional programming language, specifying what functions are available and their input and output types. In addition, WSDL allows specification of the URL and network port number to be used to invoke the web service. There is also a standard called **Universal Description, Discovery, and Integration** (**UDDI**) that defines how a directory of available web services may be created and how a program may search in the directory to find a web service satisfying its requirements.

The following example illustrates the value of web services. An airline may define a web service providing a set of procedures that can be invoked by a travel web site; these may include procedures to find flight schedules and pricing information, as well as to make flight bookings. The travel web site may interact with multiple web services, provided by different airlines, hotels, and other companies, to provide travel information to a customer and to make travel bookings. By supporting web services, the individual companies allow a useful service to be constructed on top, integrating the individual services. Users can interact with a single web site to make their travel bookings without having to contact multiple separate web sites.

To invoke a web service, a client must prepare an appropriate SOAP XML message and send it to the service; when it gets the result encoded in XML, the client must then

extract information from the XML result. There are standard APIs in languages such as Java and C# to create and extract information from SOAP messages.

See the bibliographical notes for references to more information on web services.

30.7.4 Data Mediation

Comparison shopping is an example of a mediation application, in which data about items, inventory, pricing, and shipping costs are extracted from a variety of web sites offering a particular item for sale. The resulting aggregated information is significantly more valuable than the individual information offered by a single site.

A personal financial manager is a similar application in the context of banking. Consider a consumer with a variety of accounts to manage, such as bank accounts, credit-card accounts, and retirement accounts. Suppose that these accounts may be held at different institutions. Providing centralized management for all accounts of a customer is a major challenge. XML-based mediation addresses the problem by extracting an XML representation of account information from the respective web sites of the financial institutions where the individual holds accounts. This information may be extracted easily if the institution exports it in a standard XML format, for example, as a web service. For those that do not, *wrapper* software is used to generate XML data from HTML web pages returned by the web site. Wrapper applications need constant maintenance since they depend on formatting details of web pages, which change often. Nevertheless, the value provided by mediation often justifies the effort required to develop and maintain wrappers.

Once the basic tools are available to extract information from each source, a *mediator* application is used to combine the extracted information under a single schema. This may require further transformation of the XML data from each site, since different sites may structure the same information differently. They may also use different names for the same information (for instance, acct_number and account_id) or may even use the same name for different information. The mediator must decide on a single schema that represents all required information and must provide code to transform data between different representations. XML query languages such as XSLT and XQuery play an important role in the task of transformation between different XML representations.

30.8 Summary

- Like the Hyper-Text Markup Language (HTML) on which the web is based, the Extensible Markup Language (XML) is derived from the Standard Generalized Markup Language (SGML). XML was originally intended for providing functional markup for web documents, but it has now become the de facto standard data format for data exchange between applications.
- XML documents contain elements with matching starting and ending tags indicating the beginning and end of an element. Elements may have subelements nested

within them, to any level of nesting. Elements may also have attributes. The choice between representing information as attributes and subelements is often arbitrary in the context of data representation.

- Elements may have an attribute of type ID that stores a unique identifier for the element. Elements may also store references to other elements by using attributes of type IDREF. Attributes of type IDREFS can store a list of references.
- Documents optionally may have their schema specified by a document type definition (DTD). The DTD of a document specifies what elements may occur, how they may be nested, and what attributes each element may have. Although DTDs are widely used, they have several limitations. For instance, they do not provide a type system.
- XML Schema is now the standard mechanism for specifying the schema of an XML document. It provides a large set of basic types, as well as constructs for creating complex types and specifying integrity constraints, including key constraints and foreign-key (keyref) constraints.
- XML data can be represented as tree structures, with nodes corresponding to elements and attributes. Nesting of elements is reflected by the parent-child structure of the tree representation.
- Path expressions can be used to traverse the XML tree structure and locate data. XPath is a standard language for path expressions. It allows required elements to be specified by a file-system-like path, and it also allows selections and other features. XPath also forms part of other XML query languages.
- The XQuery language is the standard language for querying XML data. It has a structure not unlike SQL, with **for**, **let**, **where**, **order by**, and **return** clauses. However, it supports many extensions to deal with the tree nature of XML and to allow for the transformation of XML documents into other documents with a significantly different structure. XPath path expressions form a part of XQuery. XQuery supports nested queries and user-defined functions.
- The DOM and SAX APIs are widely used for programmatic access to XML data. These APIs are available from a variety of programming languages.
- XML data can be stored in any of several different ways. XML data may also be stored in file systems, or in XML databases, which use XML as their internal representation.
- XML data can be stored as strings in a relational database. Alternatively, relations can represent XML data as trees. As another alternative, XML data can be mapped to relations in the same way that E-R schemas are mapped to relational schemas. Native storage of XML in relational databases is facilitated by adding an **xml** data type to SQL.

- XML is used in a variety of applications, such as storing complex data, exchange of data between organizations in a standardized form, data mediation, and web services. Web services provide a remote-procedure call interface, with XML as the mechanism for encoding parameters as well as results.

Review Terms

- Extensible Markup Language (XML)
- Hyper-Text Markup Language (HTML)
- Standard Generalized Markup Language
- Markup language
- Tags
- Self-documenting
- Element
- Root element
- Nested elements
- Attribute
- Namespace
- Default namespace
- Schema definition
- Document Type Definition (DTD)
  - ID
  - IDREF and IDREFS
- XML Schema
  - Simple and complex types
  - Sequence type
  - Key and keyref
  - Occurrence constraints
- Tree model of XML data
- Nodes
- Querying and transformation
- Path expressions
- XPath
- XQuery
  - FLWOR expressions
    - **for**
    - **let**
    - **where**
    - **order by**
    - **return**
  - Joins
  - Nested FLWOR expression
  - Sorting
- XML API
- Document Object Model (DOM)
- Simple API for XML (SAX)
- Storage of XML data
  - In non-relational data stores
  - In relational databases
    - Store as string
    - Tree representation
    - Map to relations
    - Publish and shred
    - XML-enabled database
    - Native storage
    - SQL/XML
- XML applications

- Storing complex data
- Exchange of data
- Data mediation
- SOAP
- Web services

Practice Exercises

30.1 Write a query in XQuery on the XML representation in Figure 30.11 to find the total salary of all instructors in each department.

30.2 Write a query in XQuery on the XML representation in Figure 30.1 to compute the left outer join of department elements with course elements. (Hint: Use universal quantification.)

30.3 Write queries in XQuery to output course elements with associated instructor elements nested within the course elements, given the university information representation using ID and IDREFS in Figure 30.11.

30.4 Give a relational schema to represent bibliographical information specified according to the DTD fragment shown below:

```
<!DOCTYPE bibliography [
    <!ELEMENT book (title, author+, year, publisher, place?)>
    <!ELEMENT article (title, author+, journal, year, number, volume, pages?)>
    <!ELEMENT author ( last_name, first_name) >
    <!ELEMENT title ( #PCDATA )>
    ··· similar PCDATA declarations for year, publisher, place, journal, year,
        number, volume, pages, last_name and first_name
] >
```

The relational schema must keep track of the order of author elements. You can assume that only books and articles appear as top-level elements in XML documents.

30.5 Show the tree representation of the XML data in Figure 30.1, and the representation of the tree using *nodes* and *child* relations described in Section 30.6.2.

30.6 Consider the following recursive DTD:

```
<!DOCTYPE parts [
    <!ELEMENT part (name, subpartinfo*)>
    <!ELEMENT subpartinfo (part, quantity)>
    <!ELEMENT name ( #PCDATA )>
    <!ELEMENT quantity ( #PCDATA )>
] >
```

a. Show how to map this DTD to a relational schema. You can assume that part names are unique; that is, wherever a part appears, its subpart structure will be the same.

b. Create a schema in XML Schema corresponding to this DTD.

Exercises

30.7 Show, by giving a DTD, how to represent the non-1NF *books* relation from Section 29.1, using XML.

30.8 Consider the schema:

Emp = (ename, ChildrenSet **setof***(Children), SkillsSet* **setof***(Skills))*
Children = (name, Birthday)
Birthday = (day, month, year)
Skills = (type, ExamsSet **setof***(Exams))*
Exams = (year, city)

Write the following queries in XQuery:

a. Find the names of all employees who have a child who has a birthday in March.

b. Find those employees who took an examination for the skill type "typing" in the city "Dayton".

c. List all skill types in *Emp*.

30.9 One way to share an XML document is to use XQuery to convert the schema to an SQL/XML mapping of the corresponding relational schema, and then use the SQL/XML mapping in the backward direction to populate the relation.

As an illustration, give an XQuery query to convert data from the *university-1* XML schema to the SQL/XML schema shown in Figure 30.15.

30.10 Consider the XML data shown in Figure 30.3. Suppose we wish to find purchase orders that ordered two or more copies of the part with identifier 123. Consider the following attempt to solve this problem:

```
for $p in purchaseorder
where $p/part/id = 123 and $p/part/quantity >= 2
return $p
```

Explain why the query may return some purchase orders that order less than two copies of part 123. Give a correct version of the above query.

30.11 Give a query in XQuery to flip the nesting of data from Exercise 30.7. That is, at the outermost level of nesting, the output must have elements corresponding to authors, and each such element must have nested within it items corresponding to all the books written by the author.

30.12 Consider the bibliography DTD fragment:

```
<!DOCTYPE bibliography [
    <!ELEMENT book (title, author+, year, publisher, place?)>
    <!ELEMENT article (title, author+, journal, year, number, volume, pages?)>
    <!ELEMENT author ( last_name, first_name) >
    <!ELEMENT title ( #PCDATA )>
    ··· similar PCDATA declarations for year, publisher, place, journal, year,
        number, volume, pages, last_name and first_name
] >
```

Write the following queries in XQuery:

a. Find all authors who have authored a book and an article in the same year.

b. Display books and articles sorted by year.

c. Display books with more than one author.

d. Find all books that contain the word *database* in their title and the word *Hank* in an author's name (whether first or last).

30.13 Give a relational mapping of the XML purchase order schema illustrated in Figure 30.3, using the approach described in Section 30.6.2.3. Suggest how to remove redundancy in the relational schema if item identifiers functionally determine the description and purchase and supplier names functionally determine the purchase and supplier address, respectively.

30.14 Write queries in SQL/XML to convert university data from the relational schema we have used in earlier chapters to the *university-1* and *university-2* XML schemas.

30.15 Consider the example XML schema from Section 30.3.2, and write XQuery queries to carry out the following tasks:

a. Check if the key constraint shown in Section 30.3.2 holds.

b. Check if the keyref constraint shown in Section 30.3.2 holds.

30.16 Consider Exercise 30.4, and suppose that authors could also appear as top-level elements. What change would have to be done to the relational schema?

30.17 As in Exercise 30.15, write queries to convert university data to the *university-1* and *university-2* XML schemas, but this time by writing XQuery queries on the default SQL/XML database to XML mapping.

Tools

A number of tools to deal with XML are available in the public domain. The W3C web site www.w3.org has pages describing the various XML-related standards, as well as pointers to software tools such as language implementations. An extensive list of XQuery implementations is available at www.w3.org/XML/Query. Saxon D (saxon.sourceforge.net) and Galax (www.galaxquery.org/) are useful as learning tools, although not designed to handle large databases. Exist (exist-db.org) is an open-source XML database, supporting a variety of features. Several commercial databases, including IBM DB2, Oracle, and Microsoft SQL Server, support XML storage, publishing using various SQL extensions, and querying using XPath and XQuery.

Further Reading

The World Wide Web Consortium (W3C) acts as the standards body for web-related standards, including basic XML and all the XML-related languages such as XPath, XSLT, and XQuery. A large number of technical reports defining the XML-related standards are available at www.w3.org. This site also contains tutorials and pointers to software implementing the various standards.

The XQuery language derives from an XML query language called Quilt; Quilt itself included features from earlier languages such as XPath, discussed in Section 30.4.2, and two other XML query languages, XQL and XML-QL. Quilt is described in [Chamberlin et al. (2000)].

Bibliography

[Chamberlin et al. (2000)] D. D. Chamberlin, J. Robie, and D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources", In *Proc. of the International Workshop on the Web and Databases (WebDB)* (2000), pages 53–62.

Credits

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

CHAPTER 31

Information Retrieval

Textual data are unstructured, unlike the rigidly structured data in relational databases. The term **information retrieval** generally refers to the querying of unstructured textual data. Information-retrieval systems have much in common with database systems, in particular, the storage and retrieval of data on secondary storage. However, the emphasis in the field of information systems is different from that in database systems, concentrating on issues such as querying based on keywords; the relevance of documents to the query; and the analysis, classification, and indexing of documents. web search engines today go beyond the paradigm of retrieving documents and address broader issues such as what information to display in response to a keyword query, to satisfy the information needs of a user.

31.1 Overview

The field of **information retrieval** has developed in parallel with the field of databases. In the traditional model used in the field of information retrieval, information is organized into documents, and it is assumed that there is a large number of documents. Data contained in documents are unstructured, without any associated schema. The process of information retrieval consists of locating relevant documents on the basis of user input, such as keywords or example documents.

The web provides a convenient way to get to, and to interact with, information sources across the internet. However, a persistent problem facing the web is the explosion of stored information, with little guidance to help the user to locate what is interesting. Information retrieval has played a critical role in making the web a productive and useful tool, especially for researchers.

Traditional examples of information-retrieval systems are online library catalogs and online document-management systems such as those that store newspaper articles. The data in such systems are organized as a collection of *documents*; a newspaper article and a catalog entry (in a library catalog) are examples of documents. In the context of the web, usually each HTML page is considered to be a document.

A user of such a system may want to retrieve a particular document or a particular class of documents. The intended documents are typically described by a set of **keywords**—for example, the keywords "database system" may be used to locate books on database systems, and the keywords "stock" and "scandal" may be used to locate articles about stock-market scandals. Documents have associated with them a set of keywords, and documents whose keywords contain those supplied by the user are retrieved.

Keyword-based information retrieval can be used not only for retrieving textual data, but also for retrieving other types of data, such as video and audio data, that have descriptive keywords associated with them. For instance, a video movie may have associated with it keywords such as its title, director, actors, and genre, while an image or video clip may have tags, which are keywords describing the image or video clip, associated with it.

There are several differences between this model and the models used in traditional database systems.

- Database systems deal with several operations that are not addressed in information-retrieval systems. For instance, database systems deal with updates and with the associated transactional requirements of concurrency control and durability. These matters are viewed as less important in information systems. Similarly, database systems deal with structured information organized with relatively complex data models (such as the relational model or object-oriented data models), whereas information-retrieval systems traditionally have used a much simpler model, where the information in the database is organized simply as a collection of unstructured documents.
- Information-retrieval systems deal with several issues that have not been addressed adequately in database systems. For instance, the field of information retrieval has dealt with the issue of querying collections of unstructured documents, focusing on issues such as keyword queries, and of ranking of documents on estimated degree of relevance of the documents to the query.

In addition to simple keyword queries that are just sets of words, information-retrieval systems typically allow query expressions formed using keywords and the logical connectives *and, or*, and *not*. For example, a user could ask for all documents that contain the keywords "motorcycle *and* maintenance," or documents that contain the keywords "computer *or* microprocessor," or even documents that contain the keyword "computer *but not* database." A query containing keywords without any of the above connectives is assumed to have *and*s implicitly connecting the keywords.

In **full text** retrieval, all the words in each document are considered to be keywords. For unstructured documents, full text retrieval is essential since there may be no information about what words in the document are keywords. We shall use the word **term** to refer to the words in a document, since all words are keywords.

In its simplest form, an information-retrieval system locates and returns all documents that contain all the keywords in the query, if the query has no connectives; connectives are handled as you would expect. More sophisticated systems estimate relevance of documents to a query so that the documents can be shown in order of estimated relevance. They use information about term occurrences, as well as hyperlink information, to estimate relevance.

Information-retrieval systems, as exemplified by web search engines, have today evolved beyond just retrieving documents based on a ranking scheme. Today, search engines aim to satisfy a user's information needs by judging what topic a query is about and displaying not only web pages judged as relevant, but also displaying other kinds of information about the topic. For example, given a query term *cricket*, a search engine may display scores from ongoing or recent cricket matches, rather than just display top-ranked documents related to cricket. As another example, in response to a query "New York", a search engine may show a map of New York, and images of New York, in addition to web pages related to New York.

31.2 Relevance Ranking Using Terms

The set of all documents that satisfy a query expression may be very large; in particular, there are billions of documents on the web, and most keyword queries on a web search engine find hundreds of thousands of documents containing the keywords. Full text retrieval makes this problem worse: each document may contain many terms, and even terms that are mentioned only in passing are treated equivalently with documents where the term is indeed relevant. Irrelevant documents may be retrieved as a result.

Information-retrieval systems therefore estimate the relevance of documents to a query and return only highly ranked documents as answers. **Relevance ranking** is not an exact science, but there are some well-accepted approaches.

31.2.1 Ranking Using TF-IDF

The first question to address is, given a particular term t, how relevant is a particular document d to the term. One approach is to use the the number of occurrences of the term in the document as a measure of its relevance, on the assumption that relevant terms are likely to be mentioned many times in a document. Just counting the number of occurrences of a term is usually not a good indicator: first, the number of occurrences depends on the length of the document, and second, a document containing 10 occurrences of a term may not be 10 times as relevant as a document containing one occurrence.

One way of measuring $TF(d, t)$, the relevance of a document d to a term t, is

$$TF(d, t) = \log\left(1 + \frac{n(d, t)}{n(d)}\right)$$

where $n(d)$ denotes the number of term occurrences in the document and $n(d, t)$ denotes the number of occurrences of term t in the document d. Observe that this metric takes the length of the document into account. The relevance grows with more occurrences of a term in the document, although it is not directly proportional to the number of occurrences.

Many systems refine the preceding metric by using other information. For instance, if the term occurs in the title, or the author list, or the abstract, the document would be considered more relevant to the term. Similarly, if the first occurrence of a term is late in the document, the document may be considered less relevant than if the first occurrence is early in the document. These notions can be formalized by extensions of the formula we have shown for $TF(d, t)$. In the information retrieval community, the relevance of a document to a term is referred to as **term frequency** (**TF**), regardless of the exact formula used.

A query Q may contain multiple keywords. The relevance of a document to a query with two or more keywords is estimated by combining the relevance measures of the document to each keyword. A simple way of combining the measures is to add them up. However, not all terms used as keywords are equal. Suppose a query uses two terms, one of which occurs frequently, such as *database*, and another that is less frequent, such as *Silberschatz*. A document containing *Silberschatz* but not *database* should be ranked higher than a document containing the term *database* but not *Silberschatz*.

To fix the problem, weights are assigned to terms using the **inverse document frequency** (**IDF**), defined as

$$IDF(t) = \frac{1}{n(t)}$$

where $n(t)$ denotes the number of documents (among those indexed by the system) that contain the term t. The **relevance** of a document d to a set of terms Q is then defined as

$$r(d, Q) = \sum_{t \in Q} TF(d, t) * IDF(t)$$

This measure can be further refined if the user is permitted to specify weights $w(t)$ for terms in the query, in which case the user-specified weights are also taken into account by multiplying $TF(t)$ by $w(t)$ in the above formula.

The above approach of using term frequency and inverse document frequency as a measure of the relevance of a document is called the **TF–IDF** approach.

Almost all text documents (in English) contain words such as *and*, *or*, *a*, and so on, and hence these words are useless for querying purposes since their inverse document frequency is extremely low. Information-retrieval systems define a set of words, called **stop words**, containing 100 or so of the most common words, and ignore these words when indexing a document. Such words are not used as keywords and are discarded if present in the keywords supplied by the user.

Another factor taken into account when a query contains multiple terms is the **proximity** of the terms in the document. If the terms occur close to each other in the document, the document will be ranked higher than if they occur far apart. The formula for $r(d, Q)$ can be modified to take proximity of the terms into account.

Given a query Q, the job of an information-retrieval system is to return documents in descending order of their relevance to Q. Since there may be a very large number of documents that are relevant, information-retrieval systems typically return only the first few documents with the highest degree of estimated relevance, and they permit users to interactively request further documents.

31.2.2 Similarity-Based Retrieval

Certain information-retrieval systems permit **similarity-based retrieval**. Here, the user can give the system document A, and ask the system to retrieve documents that are "similar" to A. The similarity of a document to another may be defined, for example, on the basis of common terms. One approach is to find k terms in A with highest values of $TF(A, t) * IDF(t)$, and to use these k terms as a query to find the relevance of other documents. The terms in the query are themselves weighted by $TF(A, t) * IDF(t)$.

More generally, the similarity of documents is defined by the **cosine similarity** metric. Let the terms occurring in either of the two documents be $t_1, t_2, \ldots, t_n$. Let $r(d, t) = TF(d, t) * IDF(t)$. Then the cosine similarity metric between documents d and e is defined as:

$$\frac{\sum_{i=1}^{n} r(d, t_i) r(e, t_i)}{\sqrt{\sum_{i=1}^{n} r(d, t_i)^2} \sqrt{\sum_{i=1}^{n} r(e, t_i)^2}}$$

You can easily verify that the cosine similarity metric of a document with itself is 1, while that between two documents that do not share any terms is 0.

The name "cosine similarity" comes from the fact that the above formula computes the cosine of the angle between two vectors, one representing each document, defined as follows: Let there be n words overall across all the documents being considered. An n-dimensional space is defined, with each word as one of the dimensions. A document d is represented by a point in this space, with the value of the ith coordinate of the point being $r(d, t_i)$. The vector for document d connects the origin (all coordinates = 0) to the point representing the document. The model of documents as points and vectors in an n-dimensional space is called the **vector space model**.

If the set of documents similar to a query document A is large, the system may present the user a few of the similar documents, allow the user to choose the most relevant few, and start a new search based on similarity to A *and* to the chosen documents. The resultant set of documents is likely to be what the user intended to find. This idea is called **relevance feedback**.

Relevance feedback can also be used to help users find relevant documents from a large set of documents matching the given query keywords. In such a situation, users

may be allowed to identify one or a few of the returned documents as relevant; the system then uses the identified documents to find other similar ones. The resultant set of documents is likely to be what the user intended to find. An alternative to the relevance feedback approach is to require users to modify the query by adding more keywords; relevance feedback can be easier to use, in addition to giving a better final set of documents as the answer.

In order to show the user a representative set of documents when the number of documents is very large, a search system may cluster the documents based on their cosine similarity. Then a few documents from each cluster may be shown, so that more than one cluster is represented in the set of answers.

As a special case of similarity, there are often multiple copies of a document on the web; this could happen, for example, if a web site mirrors the contents of another web site. In this case, it makes no sense to return multiple copies of a highly ranked document as separate answers; duplicates should be detected, and only one copy should be returned as an answer.

31.3 Relevance Using Hyperlinks

Early web-search engines ranked documents by using only TF-IDF–based relevance measures like those described in Section 31.2. However, these techniques had some limitations when used on very large collections of documents, such as the set of all web pages. In particular, many web pages have all the keywords specified in a typical search engine query; further, some of the pages that users want as answers often have just a few occurrences of the query terms and would not get a very high TF-IDF score.

However, researchers soon realized that web pages have very important information that plain text documents do not have, namely hyperlinks. These can be exploited to get better relevance ranking; in particular, the relevance ranking of a page is influenced greatly by hyperlinks that point *to* the page. In this section, we study how hyperlinks are used for ranking of web pages.

31.3.1 Popularity Ranking

The basic idea of **popularity ranking** (also called **prestige ranking**) is to find pages that are popular and to rank them higher than other pages that contain the specified keywords. Since most searches are intended to find information from popular pages, ranking such pages higher is generally a good idea. For instance, the term *google* may occur in vast numbers of pages, but the page google.com is the most popular among the pages that contain the term *google*. The page google.com should therefore be ranked as the most relevant answer to a query consisting of the term *google*.

Traditional measures of relevance of a page such as the TF-IDF-based measures that we saw in Section 31.2 can be combined with the popularity of the page to get an overall measure of the relevance of the page to the query. Pages with the highest overall relevance value are returned as the top answers to a query.

This raises the question of how to define and how to find the popularity of a page. One way would be to find how many times a page is accessed and use the number as a measure of the site's popularity. However, getting such information is impossible without the cooperation of the site, and while a few sites may be persuaded to reveal this information, it is difficult to get it for all sites. Further, sites may lie about their access frequency in order to get ranked higher.

A very effective alternative is to use hyperlinks to a page as a measure of its popularity. Many people have bookmark files that contain links to sites that they use frequently. Sites that appear in a large number of bookmark files can be inferred to be very popular sites. Bookmark files are usually stored privately and are not accessible on the web. However, many users do maintain web pages with links to their favorite web pages. Many web sites also have links to other related sites, which can also be used to infer the popularity of the linked sites. A web search engine can fetch web pages (by a process called crawling, which we describe in Section 31.7) and analyze them to find links between the pages.

A first solution to estimating the popularity of a page is to use the number of pages that link to the page as a measure of its popularity. However, this by itself has the drawback that many sites have a number of useful pages, yet external links often point only to the root page of the site. The root page in turn has links to other pages in the site. These other pages would then be wrongly inferred to be not very popular and would have a low ranking in answering queries.

One alternative is to associate popularity with sites, rather than with pages. All pages at a site then get the popularity of the site, and pages other than the root page of a popular site would also benefit from the site's popularity. However, the question of what constitutes a site then arises. In general, the internet address prefix of a page URL would constitute the site corresponding to the page. However, there are many sites that host a large number of mostly unrelated pages, such as home page servers in universities and web portals such as groups.yahoo.com or groups.google.com. For such sites, the popularity of one part of the site does not imply popularity of another part of the site.

A simpler alternative is to allow **transfer of prestige** from popular pages to pages to which they link. Under this scheme, in contrast to the one-person one-vote principles of democracy, a link from a popular page x to a page y is treated as conferring more prestige to page y than a link from a not-so-popular page z.[1]

This notion of popularity is in fact circular, since the popularity of a page is defined by the popularity of other pages, and there may be cycles of links between pages. However, the popularity of pages can be defined by a system of simultaneous linear equations, which can be solved by matrix manipulation techniques. The linear equations can be defined in such a way that they have a unique and well-defined solution.

[1]This is similar in some sense to giving extra weight to endorsements of products by celebrities (such as film stars), so its significance is open to question, although it is effective and widely used in practice.

It is interesting to note that the basic idea underlying popularity ranking is actually quite old and first appeared in a theory of social networking developed by sociologists in the 1950s. In the social-networking context, the goal was to define the prestige of people. For example, the president of the United States has high prestige since a large number of people know him. If someone is known by multiple prestigious people, then she also has high prestige, even if she is not known by as large a number of people. The use of a set of linear equations to define the popularity measure also dates back to this work.

31.3.2 PageRank

The web search engine Google introduced **PageRank**, which is a measure of popularity of a page based on the popularity of pages that link to the page. Using the PageRank popularity measure to rank answers to a query gave results so much better than previously used ranking techniques that Google became the most widely used search engine in a rather short period of time.

PageRank can be understood intuitively using a **random walk model**. Suppose a person browsing the web performs a random walk (traversal) on web pages as follows: the first step starts at a random web page, and in each step, the random walker does one of the following: With a probability δ the walker jumps to a randomly chosen web page, and with a probability of $1 - \delta$ the walker randomly chooses one of the outlinks from the current web page and follows the link. The PageRank of a page is then the probability that the random walker is visiting the page at any given point in time.

Note that pages that are pointed to from many web pages are more likely to be visited and thus will have a higher PageRank. Similarly, pages pointed to by web pages with a high PageRank will also have a higher probability of being visited, and thus will have a higher PageRank.

PageRank can be defined by a set of linear equations, as follows: First, web pages are given integer identifiers. The jump probability matrix T is defined with $T[i,j]$ set to the probability that a random walker who is following a link out of page i follows the link to page j. Assuming that each link from i has an equal probability of being followed, $T[i,j] = 1/N_i$, where N_i is the number of links out of page i. Most entries of T are 0, and it is best represented as an adjacency list. Then the PageRank $P[j]$ for each page j can be defined as

$$P[j] = \delta/N + (1 - \delta) * \sum_{i=1}^{N} (T[i,j] * P[i])$$

where δ is a constant between 0 and 1 and N the number of pages; δ represents the probability of a step in the random walk being a jump.

The set of equations generated as above are usually solved by an an iterative technique, starting with each $P[i]$ set to $1/N$. Each step of the iteration computes new values for each $P[i]$ using the P values from the previous iteration. Iteration stops when the maximum change in any $P[i]$ value in an iteration goes below some cutoff value.

31.3.3 Other Measures of Popularity

Basic measures of popularity such as PageRank play an important role in ranking of query answers, but they are by no means the only factor. The TF-IDF scores of a page are used to judge its relevance to the query keywords, and they must be combined with the popularity ranking. Other factors must also be taken into account, to handle limitations of PageRank and related popularity measures.

Information about how often a site is visited would be a useful measure of popularity, but as mentioned earlier it is hard to obtain in general. However, search engines do track what fraction of times users click on a page when it is returned as an answer. This fraction can be used as a measure of the site's popularity. To measure the click fraction, instead of providing a direct link to the page, the search engine provides an indirect link through the search engine's site, which records the page click and transparently redirects the browser to the original link.[2]

One drawback of the PageRank algorithm is that it assigns a measure of popularity that does not take query keywords into account. For example, the page google.com is likely to have a very high PageRank because many sites contain a link to it. Suppose it contains a word mentioned in passing, such as *Stanford* (the advanced search page at Google did in fact contain this word at one point several years ago). A search on the keyword Stanford would then return google.com as the highest-ranked answer, ahead of a more relevant answer such as the Stanford University web page.

One widely used solution to this problem is to use keywords in the anchor text of links to a page to judge what topics the page is highly relevant to. The anchor text of a link consists of the text that appears within the HTML a href tag. For example, the anchor text of the link:

```
<a href="http://stanford.edu"> Stanford University</a>
```

is *Stanford University*. If many links to the page stanford.edu have the word *Stanford* in their anchor text, the page can be judged to be very relevant to the in keyword "Stanford." Text near the anchor text may also be taken into account; for example, a web site may contain the text "Stanford's home page is here" but may have used only the word *here* as anchor text in the link to the Stanford web site.

Popularity based on anchor text is combined with other measures of popularity, and with TF-IDF measures, to get an overall ranking for query answers, as we discuss in Section 31.3.5. As an implementation trick, the words in the anchor text are often treated as part of the page, with a term frequency based on the the popularity of the

[2]Sometimes this indirection is hidden from the user. For example, when you point the mouse at a link (such as http://db-book.com) in a Google query result, the link appears to point directly to the site. However, at least as of mid-2009, when you actually click on the link, Javascript code associated with the page actually rewrites the link to go indirectly through Google's site. If you use the back button of the browser to go back to the query result page, and point to the link again, the change in the linked URL becomes visible.

pages where the anchor text appears. Then TF-IDF ranking automatically takes anchor text into account.

An alternative approach to taking keywords into account when defining popularity is to compute a measure of popularity using *only* pages that contain the query keywords, instead of computing popularity using all available web pages. This approach is more expensive, since the computation of popularity ranking has to be done dynamically when a query is received, whereas PageRank is computed statically once and reused for all queries. web search engines handling billions of queries per day cannot afford to spend so much time answering a query. As a result, although this approach can give better answers, it is not very widely used.

The HITS algorithm was based on the above idea of first finding pages that contain the query keywords, and then computing a popularity measure using just this set of related pages. In addition, it introduced a notion of *hubs* and *authorities*. A **hub** is a page that stores links to many related pages; it may not in itself contain actual information on a topic, but it points to pages that contain actual information. In contrast, an **authority** is a page that contains actual information on a topic, although it may not store links to many related pages. Each page then gets a prestige value as a hub (*hub-prestige*) and another prestige value as an authority (*authority-prestige*). The definitions of prestige, as before, are cyclic and are defined by a set of simultaneous linear equations. A page gets higher hub-prestige if it points to many pages with high authority-prestige, while a page gets higher authority-prestige if it is pointed to by many pages with high hub-prestige. Given a query, pages with highest authority-prestige are ranked higher than other pages. See the bibliographical notes for references giving further details.

31.3.4 Search Engine Spamming

Search engine spamming refers to the practice of creating web pages, or sets of web pages, designed to get a high relevance rank for some queries, even though the sites are not actually popular sites. For example, a travel site may want to be ranked high for queries with the keyword "travel". It can get high TF-IDF scores by repeating the word *travel* many times in its page.[3] Even a site unrelated to travel, such as a pornographic site, could do the same thing, and would get highly ranked for a query on the word *travel*. In fact, this sort of spamming of TF-IDF was common in the early days of web search, and there was a constant battle between such sites and search engines that tried to detect spamming and deny them a high ranking.

Popularity ranking schemes such as PageRank make the job of search engine spamming more difficult, since just repeating words to get a high TF-IDF score is no longer sufficient. However, even these techniques can be spammed by creating a collection of web pages that point to each other, increasing their popularity rank. Techniques such as using sites instead of pages as the unit of ranking (with appropriately normalized jump

[3]Repeated words in a web page may confuse users; spammers can tackle this problem by delivering different pages to search engines than to other users, for the same URL, or by making the repeated words invisible, for example, by formatting the words in small white font on a white background.

probabilities) have been proposed to avoid some spamming techniques, but they are not fully effective against other spamming techniques. The war between search engine spammers and the search engines continues even today.

The hubs and authorities approach of the HITS algorithm is more susceptible to spamming. A spammer can create a web page containing links to good authorities on a topic, and gains a high hub score as a result. In addition, the spammer's web page includes links to pages that they wish to popularize, which may not have any relevance to the topic. Because these linked pages are pointed to by a page with a high hub score, they get a high but undeserved authority score.

31.3.5 Combining TF-IDF and Popularity Ranking Measures

We have seen two broad kinds of features used in ranking, TF-IDF and popularity scores such as PageRank. TF-IDF itself reflects a combination of several factors including raw term frequency and inverse document frequency, occurrence of a term in anchor text linking to the page, and a variety of other factors such as occurrence of the term in the title, occurrence of the term early in the document, and larger font size for the term, among other factors.

How to combine the scores of a page on each these factors, to generate an overall page score is a major problem that must be addressed by any information retrieval system. In the early days of search engines, humans created functions to combine scores into an overall score. But today, search engines use machine-learning techniques to decide how to combine scores. Typically, a score-combining formula is fixed, but the formula takes as parameters weights for different scoring factors. By using a training set of query results ranked by humans, a machine-learning algorithm can come up with an assignment of weights for each scoring factor that results in the best ranking performance across multiple queries.

We note that most search engines do not reveal how they compute relevance rankings; they believe that revealing their ranking techniques would allow competitors to catch up and would make the job of search engine spamming easier, resulting in poorer quality results.

31.4 Synonyms, Homonyms, and Ontologies

Consider the problem of locating documents about motorcycle maintenance, using the query "motorcycle maintenance". Suppose that the keywords for each document are the words in the title and the names of the authors. The document titled *Motorcycle Repair* would not be retrieved, since the word *maintenance* does not occur in its title.

We can solve that problem by making use of **synonyms**. Each word can have a set of synonyms defined, and the occurrence of a word can be replaced by the *or* of all its synonyms (including the word itself). Thus, the query "motorcycle *and* repair" can be replaced by "motorcycle *and* (repair *or* maintenance)." This query would find the desired document.

Keyword-based queries also suffer from the opposite problem, of **homonyms**, that is, single words with multiple meanings. For instance, the word *object* has different meanings as a noun and as a verb. The word *table* may refer to a dinner table or to a table in a relational database.

In fact, a danger even with using synonyms to extend queries is that the synonyms may themselves have different meanings. For example, "allowance" is a synonym for one meaning of the word *maintenance* but has a different meaning than what the user intended in the query "motorcycle maintenance". Documents that use the synonyms with an alternative intended meaning would be retrieved. The user is then left wondering why the system thought that a particular retrieved document (e.g., using the word *allowance*) is relevant, if it contains neither the keywords the user specified, nor words whose intended meaning in the document is synonymous with specified keywords! It is therefore a bad idea to use synonyms to extend a query without first verifying the synonyms with the user.

A better approach to this problem is for the system to understand what **concept** each word in a document represents, and similarly to understand what concepts a user is looking for, and to return documents that address the concepts that the user is interested in. A system that supports **concept-based querying** has to analyze each document to disambiguate each word in the document and replace it with the concept that it represents; disambiguation is usually done by looking at other surrounding words in the document. For example, if a document contains words such as *database* or *query*, the word *table* probably should be replaced by the concept "table: data," whereas if the document contains words such as *furniture*, *chair*, or *wood* near the word *table*, the word *table* should be replaced by the concept "table: furniture." Disambiguation based on nearby words is usually harder for user queries, since queries contain very few words, so concept-based query systems would offer several alternative concepts to the user, who picks one or more before the search continues.

Concept-based querying has several advantages; for example, a query in one language can retrieve documents in other languages, so long as they relate to the same concept. Automated translation mechanisms can be used subsequently if the user does not understand the language in which the document is written. However, the overhead of processing documents to disambiguate words is very high when billions of documents are being handled. Internet search engines therefore generally did not support concept-based querying initially, but interest in concept-based approaches is growing rapidly. However, concept-based querying systems have been built and used for other large collections of documents.

Querying based on concepts can be extended further by exploiting concept hierarchies. For example, suppose a person issues a query "flying animals"; a document containing information about "flying mammals" is certainly relevant, since a mammal is an animal. However, the two concepts are not the same, and just matching concepts would not allow the document to be returned as an answer. Concept-based querying systems can support retrieval of documents based on concept hierarchies.

Ontologies are hierarchical structures that reflect relationships between concepts. The most common relationship is the **is-a** relationship; for example, a leopard *is-a* mammal, and a mammal *is-a* animal. Other relationships, such as *part-of*, are also possible; for example, an airplane wing is *part-of* an airplane.

The WordNet system defines a large variety of concepts with associated words (called a *synset* in WordNet terminology). The words associated with a synset are synonyms for the concept; a word may be a synonym for several different concepts. In addition to synonyms, WordNet defines homonyms and other relationships. In particular, the *is-a* and *part-of* relationships that it defines connect concepts, and in effect define an ontology. The Cyc project is another effort to create an ontology.

In addition to language-wide ontologies, ontologies have been defined for specific areas to deal with terminology relevant to those areas. For example, ontologies have been created to standardize terms used in businesses; this is an important step in building a standard infrastructure for handling order processing and other interorganization flow of data. As another example, consider a medical insurance company that needs to get reports from hospitals containing diagnosis and treatment information. An ontology that standardizes the terms helps hospital staff to understand the reports unambiguously. This can greatly help in analysis of the reports—for example, to track how many cases of a particular disease occurred in a particular time frame.

It is also possible to build ontologies that link multiple languages. For example, WordNets have been built for different languages, and common concepts between languages can be linked to each other. Such a system can be used for translation of text. In the context of information retrieval, a multilingual ontology can be used to implement a concept-based search across documents in multiple languages.

The largest effort in using ontologies for concept-based queries is the **Semantic web**. The Semantic web is led by the World Wide web Consortium and consists of a collection of tools, standards, and languages that permit data on the web to be connected based on their semantics, or meaning. Instead of being a centralized repository, the Semantic web is designed to permit the same kind of decentralized, distributed growth that has made the World Wide web so successful. Key to this is the ability to integrate multiple, distributed ontologies. As a result, anyone with access to the internet can add to the Semantic web.

31.5 Indexing of Documents

An effective index structure is important for efficient processing of queries in an information-retrieval system. Documents that contain a specified keyword can be located efficiently by using an **inverted index** that maps each keyword K_i to a list S_i of (identifiers of) the documents that contain K_i. For example, if documents d_1, d_9, and d_{21} contain the term *Silberschatz*, the inverted list for the keyword "Silberschatz" would be "$d_1; d_9; d_{21}$". To support relevance ranking based on proximity of keywords, such an index may provide not just identifiers of documents, but also a list of locations within

the document where the keyword appears. For example, if "Silberschatz" appeared at position 21 in d_1, positions 1 and 19 in d_2, and positions 4, 29, and 46 in d_3, the inverted list with positions would be "$d_1/21; d_9/1, 19; d_{21}/4, 29, 46$". The inverted lists may also include with each document the term frequency of the term.

Such indices must be stored on disk, and each list S_i can span multiple disk pages. To minimize the number of I/O operations to retrieve each list S_i, the system would attempt to keep each list S_i in a set of consecutive disk pages, so the entire list can be retrieved with just one disk seek. A B^+-tree index can be used to map each keyword K_i to its associated inverted list S_i.

The *and* operation finds documents that contain all of a specified set of keywords $K_1, K_2, \ldots, K_n$. We implement the *and* operation by first retrieving the sets of document identifiers $S_1, S_2, \ldots, S_n$ of all documents that contain the respective keywords. The intersection, $S_1 \cap S_2 \cap \cdots \cap S_n$, of the sets gives the document identifiers of the desired set of documents. The *or* operation gives the set of all documents that contain at least one of the keywords $K_1, K_2, \ldots, K_n$. We implement the *or* operation by computing the union, $S_1 \cup S_2 \cup \cdots \cup S_n$, of the sets. The *not* operation finds documents that do not contain a specified keyword K_i. Given a set of document identifiers S, we can eliminate documents that contain the specified keyword K_i by taking the difference $S - S_i$, where S_i is the set of identifiers of documents that contain the keyword K_i.

Given a set of keywords in a query, many information-retrieval systems do not insist that the retrieved documents contain all the keywords (unless an *and* operation is used explicitly). In this case, all documents containing at least one of the words are retrieved (as in the *or* operation) but are ranked by their relevance measure.

To use term frequency for ranking, the index structure should additionally maintain the number of times terms occur in each document. To reduce this effort, they may use a compressed representation with only a few bits that approximates the term frequency. The index should also store the document frequency of each term (i.e., the number of documents in which the term appears).

If the popularity ranking is independent of the index term (as is the case for Page Rank), the list S_i can be sorted on the popularity ranking (and secondarily, for documents with the same popularity ranking, on document-id). Then a simple merge can be used to compute *and* and *or* operations. For the case of the *and* operation, if we ignore the TF-IDF contribution to the relevance score and merely require that the document should contain the given keywords, merging can stop once K answers have been obtained, if the user requires only the top K answers. In general, the results with the highest final score (after including TF-IDF scores) are likely to have high popularity scores and would appear near the front of the lists. Techniques have been developed to estimate the best possible scores of remaining results, and these can be used to recognize that answers not yet seen cannot be part of the top K answers. Processing of the lists can then terminate early.

However, sorting on popularity score is not fully effective in avoiding long inverted list scans, since it ignores the contribution of the TF-IDF scores. An alternative in such cases is to break up the inverted list for each term into two parts. The first part contains

documents that have a high TF-IDF score for that term (e.g., documents where the term occurs in the document title, or in anchor text referencing the document). The second part contains all documents. Each part of the list can be sorted in order of (popularity, document-id). Given a query, merging the first parts of the list for each term is likely to give several answers with an overall high score. If sufficient high-scoring answers are not found using the first parts of the lists, the second parts of the lists are used to find all remaining answers. If a document scores high on TF-IDF, it is likely to be found when merging the first parts of the lists. See the bibliographical notes for related references.

31.6 Measuring Retrieval Effectiveness

Each keyword may be contained in a large number of documents; hence, a compact representation is critical to keep space usage of the index low. Thus, the sets of documents for a keyword are maintained in a compressed form. So that storage space is saved, the index is sometimes stored such that the retrieval is approximate; a few relevant documents may not be retrieved (called a **false drop** or **false negative**), or a few irrelevant documents may be retrieved (called a **false positive**). A good index structure will not have *any* false drops, but it may permit a few false positives; the system can filter them away later by looking at the keywords that they actually contain. In web indexing, false positives are not desirable either, since the actual document may not be quickly accessible for filtering.

Two metrics are used to measure how well an information-retrieval system is able to answer queries. The first, **precision**, measures what percentage of the retrieved documents are actually relevant to the query. The second, **recall**, measures what percentage of the documents relevant to the query were retrieved. Ideally both should be 100 percent.

Precision and recall are also important measures for understanding how well a particular document-ranking strategy performs. Ranking strategies can result in false negatives and false positives, but in a more subtle sense.

- False negatives may occur when documents are ranked, as a result of relevant documents receiving a low ranking. If the system fetched all documents down to those with very low ranking there would be very few false negatives. However, humans would rarely look beyond the first few tens of returned documents, and they may thus miss relevant documents because they are not ranked highly. Exactly what is a false negative depends on how many documents are examined. Therefore, instead of having a single number as the measure of recall, we can measure the recall as a function of the number of documents fetched.
- False positives may occur because irrelevant documents get higher rankings than relevant documents. This too depends on how many documents are examined. One option is to measure precision as a function of number of documents fetched.

A better and more intuitive alternative for measuring precision is to measure it as a function of recall. With this combined measure, both precision and recall can be computed as a function of number of documents, if required.

For instance, we can say that with a recall of 50 percent the precision was 75 percent, whereas at a recall of 75 percent the precision dropped to 60 percent. In general, we can draw a graph relating precision to recall. These measures can be computed for individual queries, then averaged out across a suite of queries in a query benchmark.

Yet another problem with measuring precision and recall lies in how to define which documents are really relevant and which are not. In fact, it requires an understanding of natural language, and understanding of the intent of the query, to decide if a document is relevant or not. Researchers therefore have created collections of documents and queries and have manually tagged documents as relevant or irrelevant to the queries. Different ranking systems can be run on these collections to measure their average precision and recall across multiple queries.

31.7 Crawling and Indexing the web

web crawlers are programs that locate and gather information on the web. They recursively follow hyperlinks present in known documents to find other documents. Crawlers start from an initial set of URLs, which may be created manually. Each of the pages identified by these URLs is fetched from the web. The web crawler then locates all URL links in these pages and adds them to the set of URLs to be crawled, if they have not already been fetched, or added to the to-be-crawled set. This process is repeated by fetching all pages in the to-be-crawled set and processing the links in these pages in the same fashion. By repeating the process, all pages that are reachable by any sequence of links from the initial set of URLs would be eventually fetched.

Since the number of documents on the web is very large, it is not possible to crawl the whole web in a short period of time; and in fact, all search engines cover only some portions of the web, not all of it, and their crawlers may take weeks or months to perform a single crawl of all the pages they cover. There are usually many processes, running on multiple machines, involved in crawling. A database stores a set of links (or sites) to be crawled; it assigns links from this set to each crawler process. New links found during a crawl are added to the database and may be crawled later if they are not crawled immediately. Pages have to be refetched (i.e., links recrawled) periodically to obtain updated information and to discard sites that no longer exist, so that the information in the search index is kept reasonably up-to-date.

See the references in the bibliography for a number of practical details in performing a web crawl, such as infinite sequences of links created by dynamically generated pages (called a **spider trap**), prioritization of page fetches, and ensuring that web sites are not flooded by a burst of requests from a crawler.

Pages fetched during a crawl are handed over to a prestige computation and indexing system, which may be running on a different machine. The prestige computation

and indexing systems themselves run on multiple machines in parallel. Pages can be discarded after they are used for prestige computation and added to the index; however, they are usually cached by the search engine to give search engine users fast access to a cached copy of a page, even if the original web site containing the page is not accessible.

It is not a good idea to add pages to the same index that is being used for queries, since doing so would require concurrency control on the index and would affect query and update performance. Instead, one copy of the index is used to answer queries while another copy is updated with newly crawled pages. At periodic intervals the copies switch over, with the old one being updated while the new copy is being used for queries.

To support very high query rates, the indices may be kept in main memory, and there are multiple machines; the system selectively routes queries to the machines to balance the load among them. Popular search engines often have tens of thousands of machines carrying out the various tasks of crawling, indexing, and answering user queries.

web crawlers depend on all relevant pages being reachable through hyperlinks. However, many sites containing large collections of data may not make all the data available as hyperlinked pages. Instead, they provide search interfaces, where users can enter terms or select menu options and get results. As an example, a database of flight information is usually made available using such a search interface, without any hyperlinks to the pages containing flight information. As a result, the information inside such sites is not accessible to a normal web crawler. The information in such sites is often referred to as **deep web** information.

Deep web crawlers extract some such information by guessing what terms would make sense to enter, or what menu options to choose, in such search interfaces. By entering each possible term/option and executing the search interface, they are able to extract pages with data that they would not have been able to find otherwise. The pages extracted by a deep web crawl may be indexed just like regular web pages. The Google search engine, for example, includes results from deep web crawls.

31.8 Information Retrieval: Beyond Ranking of Pages

Information-retrieval systems were originally designed to find textual documents related to a query, and they were later extended to finding pages on the web that are related to a query. People use search engines for many different tasks, from simple tasks such as locating a web site that they want to use, to a broader goal of finding information on a topic of interest. web search engines have become extremely good at the task of locating web sites that a user wants to visit. The task of providing information on a topic of interest is much harder, and we study some approaches in this section.

There is also an increasing need for systems that try to understand documents (to a limited extent) and answer questions based on that (limited) understanding. One approach is to create structured information from unstructured documents and to answer questions based on the structured information. Another approach applies natural lan-

guage techniques to find documents related to a question (phrased in natural language) and return relevant segments of the documents as an answer to the question.

31.8.1 Diversity of Query Results

Today, search engines do not just return a ranked list of web pages relevant to a query. They also return image and video results relevant to a query. Further, there are a variety of sites providing dynamically changing content such as sports scores, or stock market tickers. To get current information from such sites, users would have to first click on the query result. Instead, search engines have created "gadgets," which take data from a particular domain, such as sports updates, stock prices, or weather conditions, and format them in a nice graphical manner, to be displayed as results for a query. Search engines have to rank the set of gadgets available in terms of relevance to a query and display the most relevant gadgets, along with web pages, images, videos, and other types of results. Thus, a query result has a diverse set of result types.

Search terms are often ambiguous. For example, a query "eclipse" may be referring to a solar or lunar eclipse, or to the integrated development environment (IDE) called Eclipse. If all the highly ranked pages for the term *eclipse* are about the IDE, a user looking for information about solar or lunar eclipses may be very dissatisfied. Search engines therefore attempt to provide a set of results that are *diverse* in terms of their topics, to minimize the chance that a user would be dissatisfied. To do so, at indexing time the search engine must disambiguate the sense in which a word is used in a page; for example, it must decide whether the use of the word *eclipse* in a page refers to the IDE or the astronomical phenomenon. Then, given a query, the search engine attempts to provide results that are relevant to the most common senses in which the query words are used.

The results obtained from a web page need to be summarized as a **snippet** in a query result. Traditionally, search engines have provided a few words surrounding the query keywords as a snippet that helps indicate what the page contains. However, there are many domains where the snippet can be generated in a much more meaningful manner. For example, if a user queries about a restaurant, a search engine can generate a snippet containing the restaurant's rating, a phone number, and a link to a map, in addition to providing a link to the restaurant's home page. Such specialized snippets are often generated for results retrieved from a database, for example, a database of restaurants.

31.8.2 Information Extraction

Information-extraction systems convert information from textual form to a more structured form. For example, a real-estate advertisement may describe attributes of a home in textual form, such as "two-bedroom, three-bath house in Queens, $1 million," from which an information extraction system may extract attributes such as number of bedrooms, number of bathrooms, cost, and neighborhood. The original advertisement could have used various terms, such as *2BR*, or two *BR*, or *two bed*, to denote two bedrooms. The extracted information can be used to structure the data in a standard

way. Thus, a user could specify that he is interested in two-bedroom houses, and a search system would be able to return all relevant houses based on the structured data, regardless of the terms used in the advertisement.

An organization that maintains a database of company information may use an information-extraction system to extract information automatically from newspaper articles; the information extracted would relate to changes in attributes of interest, such as resignations, dismissals, or appointments of company officers.

As another example, search engines designed for finding scholarly research articles, such as Citeseer and Google Scholar, crawl the web to retrieve documents that are likely to be research articles. They examine some features of each retrieved document, such as the presence of words such as *bibliography*, *references*, and *abstract*, to judge if a document is in fact a scholarly research article. They then extract the title, list of authors, and the citations at the end of the article by using information extraction techniques. The extracted citation information can be used to link each article to articles that it cites or to articles that cite it; such citation links between articles can be very useful for a researcher.

Several systems have been built for information extraction for specialized applications. They use linguistic techniques, page structure, and user-defined rules for specific domains such as real estate advertisements or scholarly publications. For limited domains, such as a specific web site, it is possible for a human to specify patterns that can be used to extract information. For example, on a particular web site, a pattern such as "Price: <number> $", where <number> indicates any number, may match locations where the price is specified. Such patterns can be created manually for a limited number of web sites.

However, on the web scale with millions of web sites, manual creation of such patterns is not feasible. Machine-learning techniques, which can learn such patterns given a set of training examples, are widely used to automate the process of information extraction.

Information extraction usually has errors in some fraction of the extracted information; typically this is because some page had information in a format that syntactically matched a pattern but did not actually specify a value (such as the price). Information extraction using simple patterns, which separately match parts of a page, is relatively error prone. Machine-learning techniques can perform much more sophisticated analysis, based on interactions between patterns, to minimize errors in the information extracted while maximizing the amount of information extracted. See the references in the bibliographical notes for more information.

31.8.3 Question Answering

Information retrieval systems focus on finding documents relevant to a given query. However, the answer to a query may lie in just one part of a document, or in small parts of several documents. **Question answering** systems attempt to provide direct answers to questions posed by users. For example, a question of the form "Who killed Lincoln?"

may best be answered by a line that says "Abraham Lincoln was shot by John Wilkes Booth in 1865." Note that the answer does not actually contain the words *killed* or *who,* but the system infers that "who" can be answered by a name, and "killed" is related to "shot."

Question answering systems targeted at information on the web typically generate one or more keyword queries from a submitted question, execute the keyword queries against web search engines, and parse returned documents to find segments of the documents that answer the question. A number of linguistic techniques and heuristics are used to generate keyword queries and to find relevant segments from the document.

An issue in answering questions is that different documents may indicate different answers to a question. For example, if the question is "How tall is a giraffe?" different documents may give different numbers as an answer. These answers form a distribution of values, and a question answering system may choose the average, or median value of the distribution as the answer to be returned; to reflect the fact that the answer is not expected to be precise, the system may return the average along with the standard deviation (e.g., average of 16 feet, with a standard deviation of 2 feet), or a range based on the average and the standard deviation (e.g., between 14 and 18 feet).

Current-generation question answering systems are limited in power, since they do not really understand either the question or the documents used to answer the question. However, they are useful for a number of simple question answering tasks.

31.8.4 Querying Structured Data

Structured data are primarily represented in either relational or XML form. Several systems have been built to support keyword querying on relational and XML data (see Chapter 30). A common theme between these systems lies in finding nodes (tuples or XML elements) containing the specified keywords and finding connecting paths (or common ancestors, in the case of XML data) between them.

For example, a query "Zhang Katz" on a university database may find the *name* "Zhang" occurring in a *student* tuple, and the *name* "Katz" in an *instructor* tuple, and a path through the *advisor* relation connecting the two tuples. Other paths, such as student "Zhang" taking a course taught by "Katz," may also be found in response to this query. Such queries may be used for ad hoc browsing and querying of data when the user does not know the exact schema and does not wish to take the effort to write an SQL query defining what she is searching for. Indeed, it is unreasonable to expect lay users to write queries in a structured query language, whereas keyword querying is quite natural.

Since queries are not fully defined, they may have many different types of answers, which must be ranked. A number of techniques have been proposed to rank answers in such a setting, based on the lengths of connecting paths, and on techniques for assigning directions and weights to edges. Techniques have also been proposed for assigning popularity ranks to tuples and XML elements, based on links such as foreign

key and IDREF links. See the bibliographical notes for more information on keyword searching of relational and XML data.

31.9 Directories and Categories

A typical library user may use a catalog to locate a book for which she is looking. When she retrieves the book from the shelf, however, she is likely to *browse* through other books that are located nearby. Libraries organize books in such a way that related books are kept close together. Hence, a book that is physically near the desired book may be of interest as well, making it worthwhile for users to browse through such books.

To keep related books close together, libraries use a **classification hierarchy**. Books on science are classified together. Within this set of books, there is a finer classification, with computer-science books organized together, mathematics books organized together, and so on. Since there is a relation between mathematics and computer science, relevant sets of books are stored close to each other physically. At yet another level in the classification hierarchy, computer-science books are broken down into subareas, such as operating systems, languages, and algorithms. Figure 31.1 illustrates a classification hierarchy that may be used by a library. Because books can be kept at only one place, each book in a library is classified into exactly one spot in the classification hierarchy.

In an information-retrieval system, there is no need to store related documents close together. However, such systems need to *organize documents logically* to permit browsing. Thus, such a system could use a classification hierarchy similar to one that

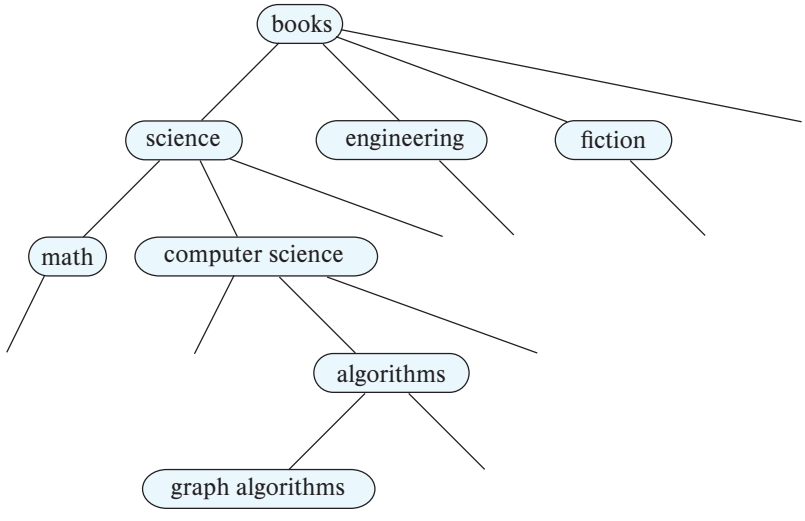

Figure 31.1 A classification hierarchy for a library system.

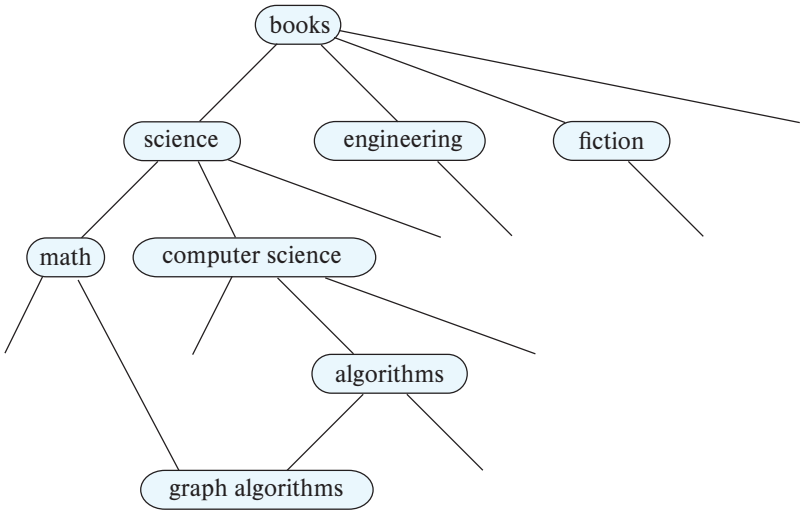

Figure 31.2 A classification DAG for a library information-retrieval system.

libraries use, and, when it displays a particular document, it can also display a brief description of documents that are close in the hierarchy.

In an information-retrieval system, there is no need to keep a document in a single spot in the hierarchy. A document that talks of mathematics for computer scientists could be classified under mathematics as well as under computer science. All that is stored at each spot is an identifier of the document (i.e., a pointer to the document), and it is easy to fetch the contents of the document by using the identifier.

As a result of this flexibility, not only can a document be classified under two locations, but also a subarea in the classification hierarchy can itself occur under two areas. The class of "graph algorithm" documents can appear both under mathematics and under computer science. Thus, the classification hierarchy is now a directed acyclic graph (DAG), as shown in Figure 31.2. A graph-algorithm document may appear in a single location in the DAG but can be reached via multiple paths.

A **directory** is simply a classification DAG structure. Each leaf of the directory stores links to documents on the topic represented by the leaf. Internal nodes may also contain links, for example, to documents that cannot be classified under any of the child nodes.

To find information on a topic, a user would start at the root of the directory and follow paths down the DAG until reaching a node representing the desired topic. While browsing down the directory, the user can find not only documents on the topic he is interested in, but also find related documents and related classes in the classification hierarchy. The user may learn new information by browsing through documents (or subclasses) within the related classes.

Organizing the enormous amount of information available on the web into a directory structure is a daunting task.

- The first problem is determining what exactly the directory hierarchy should be.
- The second problem is, given a document, deciding which nodes of the directory are categories relevant to the document.

To tackle the first problem, portals such as Yahoo! have teams of "Internet librarians" who come up with the classification hierarchy and continually refine it.

The second problem can also be tackled manually by librarians, or web site maintainers may be responsible for deciding where their sites should lie in the hierarchy. There are also techniques for deciding automatically the location of documents based on computing their similarity to documents that have already been classified.

Wikipedia, the online encyclopedia, addresses the classification problem in the reverse direction. Each page in Wikipedia has a list of **categories** to which it belongs. For example, as of 2009, the Wikipedia page on giraffes had several categories including "Mammals of Africa." In turn, the "Mammals of Africa" category itself belongs to the category "Mammals by geography," which in turn belongs to the category "Mammals", which in turn has a category "Vertebrates," and so on. The category structure is useful for browsing other instances of the same category, for example, to find other mammals of Africa, or other mammals. Conversely, a query that looks for mammals can use the category information to infer that a giraffe is a mammal. The Wikipedia category structure is not a tree but is almost a DAG; it is not actually a DAG since it has a few instances of loops, which probably reflect categorization errors.

31.10 Summary

- Information-retrieval systems are used to store and query textual data such as documents. They use a simpler data model than do database systems but provide more powerful querying capabilities within the restricted model.
- Queries attempt to locate documents that are of interest by specifying, for example, sets of keywords. The query that a user has in mind usually cannot be stated precisely; hence, information-retrieval systems order answers on the basis of potential relevance.
- Relevance ranking makes use of several types of information, such as:
  - Term frequency: how important each term is to each document.
  - Inverse document frequency.
  - Popularity ranking.
- Similarity of documents is used to retrieve documents similar to an example document. The cosine metric is used to define similarity and is based on the vector space model.

- PageRank and hub/authority rank are two ways to assign prestige to pages on the basis of links to the page. The PageRank measure can be understood intuitively using a random-walk model. Anchor text information is also used to compute a per-keyword notion of popularity. Information-retrieval systems need to combine scores on multiple factors, such as TF-IDF and PageRank, to get an overall score for a page.
- Search engine spamming attempts to get (an undeserved) high ranking for a page.
- Synonyms and homonyms complicate the task of information retrieval. Concept-based querying aims at finding documents that contain specified concepts, regardless of the exact words (or language) in which the concept is specified. Ontologies are used to relate concepts using relationships such as is-a or part-of.
- Inverted indices are used to answer keyword queries.
- Precision and recall are two measures of the effectiveness of an information-retrieval system.
- web search engines crawl the web to find pages, analyze them to compute prestige measures, and index them.
- Techniques have been developed to extract structured information from textual data, to perform keyword querying on structured data, and to give direct answers to simple questions posed in natural language.
- Directory structures and categories are used to classify documents with other similar documents.

Review Terms

- Information-retrieval systems
- Keyword search
- Full text retrieval
- Term
- Relevance ranking
  - Term frequency
  - Inverse document frequency
  - Relevance
  - Proximity
- Similarity-based retrieval
  - Vector space model
  - Cosine similarity metric
  - Relevance feedback
- Stop words
- Relevance using hyperlinks
  - Popularity/prestige
  - Transfer of prestige
- PageRank
  - Random walk model
- Anchor-text – based relevance
- Hub/authority ranking

- Search engine spamming
- Synonyms
- Homonyms
- Concepts
- Concept-based querying
- Ontologies
- Semantic web
- Inverted index
- False drop
- False negative
- False positive
- Precision
- Recall
- web crawlers
- Deep web
- Query result diversity
- Information extraction
- Question answering
- Querying structured data
- Directories
- Classification hierarchy
- Categories

Practice Exercises

31.1 Compute the relevance (using appropriate definitions of term frequency and inverse document frequency) of each of the Practice Exercises in this chapter to the query "SQL relation".

31.2 Suppose you want to find documents that contain at least k of a given set of n keywords. Suppose also you have a keyword index that gives you a (sorted) list of identifiers of documents that contain a specified keyword. Give an efficient algorithm to find the desired set of documents.

31.3 Suggest how to implement the iterative technique for computing PageRank given that the T matrix (even in adjacency list representation) does not fit in memory.

31.4 Suggest how a document containing a word (such as *leopard*) can be indexed such that it is efficiently retrieved by queries using a more general concept (such as "carnivore" or "mammal"). You can assume that the concept hierarchy is not very deep, so each concept has only a few generalizations (a concept can, however, have a large number of specializations). You can also assume that you are provided with a function that returns the concept for each word in a document. Also suggest how a query using a specialized concept can retrieve documents using a more general concept.

31.5 Suppose inverted lists are maintained in blocks, with each block noting the largest popularity rank and TF-IDF scores of documents in the remaining blocks in the list. Suggest how merging of inverted lists can stop early if the user wants only the top K answers.

Exercises

31.6 Using a simple definition of term frequency as the number of occurrences of the term in a document, give the TF-IDF scores of each term in the set of documents consisting of this and the next exercise.

31.7 Create a small example of four small documents, each with a PageRank, and create inverted lists for the documents sorted by the PageRank. You do not need to compute PageRank, just assume some values for each page.

31.8 Suppose you wish to perform keyword querying on a set of tuples in a database, where each tuple has only a few attributes, each containing only a few words. Does the concept of term frequency make sense in this context? And that of inverse document frequency? Explain your answer. Also suggest how you can define the similarity of two tuples using TF-IDF concepts.

31.9 Web sites that want to get some publicity can join a web ring, where they create links to other sites in the ring in exchange for other sites in the ring creating links to their site. What is the effect of such rings on popularity ranking techniques such as PageRank?

31.10 The Google search engine provides a feature whereby Web sites can display advertisements supplied by Google. The advertisements supplied are based on the contents of the page. Suggest how Google might choose which advertisements to supply for a page, given the page contents.

31.11 One way to create a keyword-specific version of PageRank is to modify the random jump such that a jump is only possible to pages containing the keyword. Thus, pages that do not contain the keyword but are close (in terms of links) to pages that contain the keyword also get a nonzero rank for that keyword.

a. Give equations defining such a keyword-specific version of PageRank.

b. Give a formula for computing the relevance of a page to a query containing multiple keywords.

31.12 The idea of popularity ranking using hyperlinks can be extended to relational and XML data, using foreign key and IDREF edges in place of hyperlinks. Suggest how such a ranking scheme may be of value in the following applications:

a. A bibliographic database that has links from articles to authors of the articles and links from each article to every article that it references.

b. A sales database that has links from each sales record to the items that were sold.

Also suggest why prestige ranking can give less than meaningful results in a movie database that records which actor has acted in which movies.

31.13 What is the difference between a false positive and a false drop? If it is essential that no relevant information be missed by an information-retrieval query, is it acceptable to have either false positives or false drops? Why?

Tools

Google (http://www.google.com) is currently the most popular search engine, but there are a number of other search engines, such as Microsoft Bing (http://www.bing.com) and Yahoo! search (search.yahoo.com). The site searchenginewatch.com provides a wealth of information about search engines. Yahoo! (dir.yahoo.com) and the Open Directory Project (dmoz.org) provide classification hierarchies for web sites.

Further Reading

[Manning et al. (2008)], [Chakrabarti (2002)], [Grossman and Frieder (2004)], [Witten et al. (1999)], and [Baeza-Yates and Ribeiro-Neto (1999)] provide textbook descriptions of information retrieval. In particular, [Chakrabarti (2002)] and [Manning et al. (2008)] provide detailed coverage of web crawling, ranking techniques, and mining techniques related to information retrieval such as text classification and clustering.

Bibliography

[Baeza-Yates and Ribeiro-Neto (1999)] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, Addison Wesley (1999).

[Chakrabarti (2002)] S. Chakrabarti, *Mining the Web: Discovering Knowledge from HyperText Data*, Morgan Kaufmann (2002).

[Grossman and Frieder (2004)] D. A. Grossman and O. Frieder, *Information Retrieval: Algorithms and Heuristics*, 2nd edition, Springer Verlag (2004).

[Manning et al. (2008)] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*, Cambridge University Press (2008).

[Witten et al. (1999)] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd edition, Morgan Kaufmann (1999).

Credits

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

CHAPTER 32

PostgreSQL

Ioannis Alagiannis (Swisscom AG)
Renata Borovica-Gajic (University of Melbourne, AU)[1]

PostgreSQL is an open-source object-relational database management system. It is a descendant of one of the earliest such systems, the POSTGRES system developed under Professor Michael Stonebraker at the University of California, Berkeley. The name "Postgres" is derived from the name of a pioneering relational database system, Ingres. Currently, PostgreSQL offers features such as complex queries, foreign keys, triggers, views, transactional integrity, full-text searching, and limited data replication. Users can extend PostgreSQL with new data types, functions, operators, or index methods. PostgreSQL supports a variety of programming languages (including C, C++, Java, Perl, Tcl, and Python), as well as the database interfaces JDBC and ODBC.

PostgreSQL runs under virtually all Unix-like operating systems, including Linux, Microsoft Windows and Apple Macintosh OS X. PostgreSQL has been released under the BSD license, which grants permission to anyone for the use, modification, and distribution of the PostgreSQL code and documentation for any purpose without any fee. PostgreSQL has been used to implement several different research and production applications (such as the PostGIS system for geographic information) and is used an educational tool at several universities. PostgreSQL continues to evolve through the contributions of a large community of developers.

In this chapter, we explain how PostgreSQL works, starting from user interfaces and languages, and continuing into the internals of the system. The chapter would be useful for application developers who use PostgreSQL, and desire to understand and

[1]Both authors equally contributed to this work.
This chapter is an online resource of Database System Concepts, 7th edition, by Silberschatz, Korth and Sudarshan, McGraw-Hill, 2019.

This chapter is released under the Creative Commons Attribution-NonCommercial-ShareAlike (CC BY-NC-SA) 4.0 International License. To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-sa/4.0. For any use beyond those covered by this license, obtain permission by emailing *db-book-authors@cs.yale.edu*.

make better use of its features. It would also be particularly useful for students and developers who wish to add functionality to the PostgreSQL system, by modifying its source code.

32.1 Interacting with PostgreSQL

The standard distribution of PostgreSQL comes with command-line tools for administering the database. However, there is a wide range of commercial and open-source graphical administration and design tools that support PostgreSQL. Software developers may also access PostgreSQL through a comprehensive set of programming interfaces.

32.1.1 Interactive Terminal Interfaces

Like most database systems, PostgreSQL offers command-line tools for database administration. The psql interactive terminal client supports execution of SQL commands on the server, and viewing of the results. Additionally, it provides the user with meta-commands and shell-like features to facilitate a wide variety of operations. Some of its features are:

- **Variables.** psql provides variable substitution features, similar to common Unix command shells.
- **SQL interpolation.** The user can substitute ("interpolate") psql variables into SQL statements by placing a colon in front of the variable name.
- **Command-line editing.** psql uses the GNU readline library for convenient line editing, with tab-completion support.

32.1.2 Graphical Interfaces

The standard distribution of PostgreSQL does not contain any graphical tools. However, there are several open source as well as commercial graphical user interface tools for tasks such as SQL development, database administration, database modeling/design and report generation. Graphical tools for SQL development and database administration include pgAdmin, PgManager, and RazorSQL. Tools for database design include TOra, Power*Architect, and PostgreSQL Maestro. Moreover, PostgreSQL works with several commercial forms-design and report-generation tools such as Reportizer, dbForge, and Database Tour.

32.1.3 Programming Language Interfaces

PostgreSQL standard distribution includes two client interfaces libpq and ECPG. The libpq library provides the C API for PostgreSQL and is the underlying engine for most programming-language bindings. The libpq library supports both synchronous

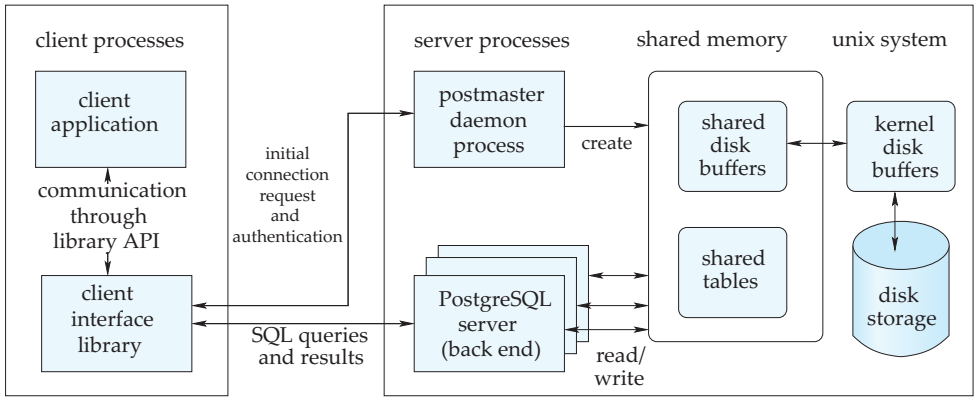

Figure 32.1 The PostgreSQL system architecture.

and asynchronous execution of SQL commands and prepared statements, through a re-entrant and thread-safe interface. The connection parameters of libpq may be configured in several flexible ways, such as setting environment variables, placing settings in a local file, or creating entries on an LDAP server. ECPG (Embedded SQL in C) is an embedded SQL preprocessor for the C language and PostgreSQL. It allows for accessing PostgreSQL using C programs with embedded SQL code in the following form: *EXEC SQL* sql-statements. ECPG provides a flexible interface for connecting to the database server, executing SQL statements, and retrieving query results, among other features.

Apart from the client interfaces included in the standard distribution of PostgreSQL, there are also client interfaces provided from external projects. These include native interfaces for ODBC and JDBC, as well as bindings for most programming languages, including C, C++, PHP, Perl, Tcl/Tk, JavaScript, Python, and Ruby.

32.2 System Architecture

The PostgreSQL system is based on a multi-process architecture, as shown in Figure 32.1. A set of one or more of databases is managed by a collection of processes. The **postmaster** is the central coordinating process and is responsible for system initialization (including allocation of shared memory and starting of background processes), and for shutting down the server. Additionally, the **postmaster** manages connections with client applications and assigns each new connecting client to a backend server process for executing the queries on behalf of the client and for returning the results to the client.

Client applications can connect to the PostgreSQL server and submit queries through one of the many database application program interfaces supported by PostgreSQL (libpq, JDBC, ODBC) that are provided as client-side libraries. An example client application is the command-line psql program, included in the standard

PostgreSQL distribution. The postmaster is responsible for handling the initial client connections. For this, it constantly listens for new connections on a known port. When it receives a connection request, the postmaster first performs initialization steps such as user authentication, and then assigns an idle backend server process (or spawns a new one if required) to handle the new client. After this initial connection, the client interacts only with the backend server process, submitting queries and receiving query results. As long as the client connection is active, the assigned backend server process is dedicated to only that client connection. Thus, PostgreSQL uses a **process-per-connection model** for the backend server.

The backend server process is responsible for executing the queries submitted by the client by performing the necessary query-execution steps, including parsing, optimization, and execution. Each backend server process can handle only a single query at a time. An application that desires to execute more than one query in parallel must maintain multiple connections to the server. At any given time there may be multiple clients connected to the system, and thus multiple backend server processes may be executing concurrently.

In addition to the *postmaster* and the *backend* server processes PostgreSQL utilizes several background worker processes to perform data management tasks, including the background writer, the statistics collector, the write-ahead log (WAL) writer and the checkpointer processes. The background writer process is responsible for periodically writing the dirty pages from the shared buffers to persistent storage. The statistics collector process continuously collects statistics information about the table accesses and the number of rows in tables. The WAL writer process periodically flushes the WAL data to persistent storage while the checkpointer process performs database checkpoints to speed up recovery. These background processes are initiated by the *postmaster* process.

When it comes to memory management in PostgreSQL, we can identify two different categories a) local memory and b) shared memory. Each backend process allocates local memory for its own tasks such as query processing (e.g., internal sort operations hash tables and temporary tables) and maintenance operations (e.g., *vacuum*, *create index*).

On the other hand, the in-memory buffer pool is placed in shared memory, so that all the processes, including backend server processes and background processes can access it. Shared memory is also used to store lock tables and other data that must be shared by server and background processes.

Due to the use of shared memory as the inter-process communication medium, a PostgreSQL server should run on a single shared-memory machine; a single-server site cannot be executed across multiple machines that do not share memory, without the assistance of third-party packages. However, it is possible to build a shared-nothing parallel database system with an instance of PostgreSQL running on each node; in fact, several commercial parallel database systems have been built with exactly this architecture.

32.3 Storage and Indexing

PostgreSQL's approach to data layout and storage has the goals of (1) a simple and clean implementation and (2) ease of administration. As a step toward these goals, PostgreSQL relies on file-system files for data storage (also referred to as "*cooked*" files), instead of handling the physical layout of data on raw disk partitions by itself. PostgreSQL maintains a list of directories in the file hierarchy to use for storage; these directories are referred to as **tablespaces**. Each PostgreSQL installation is initialized with a default tablespace, and additional tablespaces may be added at any time. When creating a table, index, or entire database, the user may specify a tablespace in which to store the related files. It is particularly useful to create multiple tablespaces if they reside on different physical devices, so that tablespaces on the faster devices may be dedicated to data that are accessed more frequently. Moreover, data that are stored on separate disks may be accessed in parallel more efficiently.

The design of the PostgreSQL storage system potentially leads to some performance limitations, due to clashes between PostgreSQL and the file system. The use of cooked file systems results in double buffering, where blocks are first fetched from disk into the file system's cache (in kernel space), and are then copied to PostgreSQL's buffer pool. Performance can also be limited by the fact that PostgreSQL stores data in 8-KB blocks, which may not match the block size used by the kernel. It is possible to change the PostgreSQL block size when the server is installed, but this may have undesired consequences: small blocks limit the ability of PostgreSQL to store large tuples efficiently, while large blocks are wasteful when a small region of a file is accessed.

On the other hand, modern enterprises increasingly use external storage systems, such as network-attached storage and storage-area networks, instead of disks attached to servers. Such storage systems are administered and tuned for performance separately from the database. PostgreSQL may directly leverage these technologies because of its reliance on "cooked" file systems. For most applications, the performance reduction due to the use of "cooked" file systems is minimal, and is justified by the ease of administration and management, and the simplicity of implementation.

32.3.1 Tables

The primary unit of storage in PostgreSQL is a table. In PostgreSQL, tuples in a table are stored in *heap files.* These files use a form of the standard *slotted-page* format (Section 13.2.2). The PostgreSQL slotted-page format is shown in Figure 32.2. In each page, the page header is followed by an array of *line pointers* (also referred to as *item identifiers*). A line pointer contains the offset (relative to the start of the page) and length of a specific tuple in the page. The actual tuples are stored from the end of the page to simplify insertions. When a new item is added in the page, if all line pointers are in use, a new line pointer is allocated at the beginning of the unallocated space (pd_lower) while the actual item is stored from the end of the unallocated space (pd_upper).

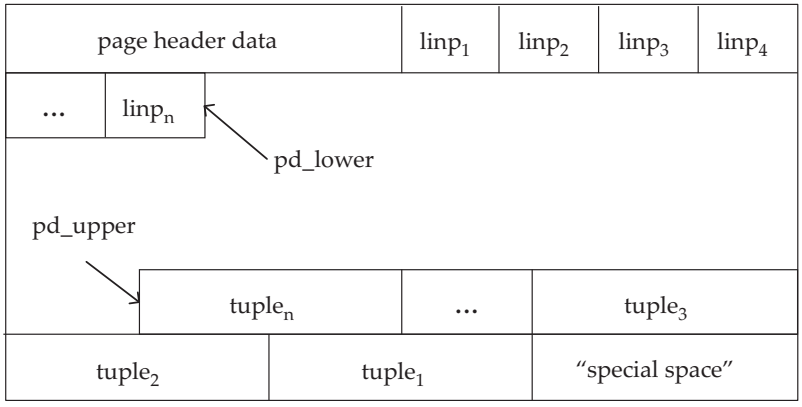

Figure 32.2 Slotted-page format for PostgreSQL tables.

A record in a heap file is identified by its **tuple ID** (**TID**). The TID consists of a 4-byte block ID which specifies the page in the file containing the tuple and a 2-byte slot ID. The slot ID is an index into the line pointer array that in turn is used to access the tuple.

Due to the multi-version concurrency control (MVCC) technique used by PostgreSQL, there may be multiple versions of a tuple, each with an associated start and end time for validity. Delete operations do not immediately delete tuples, and update operations do not directly update tuples. Instead, deletion of a tuple initially just updates the end-time for validity, while an update of a tuple create a new version of the existing tuple; the old version has its validity end-time set to just before the validity start-time of the new version.

Old versions of tuples that are no longer visible to any transaction are physically deleted later; deletion causes holes to be formed in a page. The indirection of accessing tuples through the line pointer array permits the compaction of such holes by moving the tuples, without affecting the TID of the tuple.

The length of a physical tuple is limited by the size of a data page, which makes it difficult to store very long tuples. When PostgreSQL encounters a large tuple that cannot fit in a page, it tries to "TOAST" individual large attributes, that is, compress the attribute or break it up into smaller pieces. In some cases, "toasting" an attribute may be accomplished by compressing the value. If compression does not shrink the tuple enough to fit in the page (as is often the case), the data in the toasted attribute is replaced by a reference to the attribute value; the attribute value is stored outside the page in an associated TOAST table. Large attribute values are split into smaller chunks; the chunk size is chosen such that four chunks can fit in a page. Each chunk is stored as a separate row in the associated TOAST table. An index on the combination of the identifier of a toasted attribute with the sequence number of each chunk allows efficient retrieval of the values. Only the data types with variable-length representation support

TOAST, to avoid imposing the overhead on data types that cannot produce large field values. The toasted attribute size is limited to 1 GB.

32.3.2 Indices

A PostgreSQL index is a data structure that provides a dynamic mapping from search predicates to sequences of tuple IDs from a particular table. The returned tuples are intended to match the search predicate, although in some cases the predicate must be rechecked on the actual tuples, since the index may return a superset of matching tuples. PostgreSQL supports several different index types, including indices that are based on user-extensible access methods. All the index types in PostgreSQL currently use the slotted-page format described in Section 32.3.1 to organize the data within an index page.

32.3.2.1 Index Types

PostgreSQL supports several types of indices that target different categories of workloads. In addition to the conventional B-tree,[2] PostgreSQL supports hash indices and several specialized indexing methods: the Generalized Search Tree (GiST), the Space-partitioned Generalized Search Tree (SP-GiST), the Generalized Inverted Index (GIN) and the Block Range Index (BRIN), which can be beneficial for workloads requiring full-text indexing, querying multi-value elements or being naturally clustered on some specific attribute(s).

B-tree: In PostgreSQL, the B-tree is the default index type, and the implementation is based on Lehman and Yao's B-link tree (B-link trees, described briefly in Section 18.10.2, are a variant of B^+-trees that support high concurrency of operations). B-trees can efficiently support equality and range queries on sortable data, as also certain pattern-matching operations such as some cases of **like** expressions.

Hash: PostgreSQL's hash indices are an implementation of linear hashing. Such indices are useful only for simple equality operations. The hash indices used by PostgreSQL had been shown to have lookup performance no better than that of B-trees while having considerably larger size and maintenance costs. The use of hash indices was generally discouraged due to the lack of write-ahead logging. However, in PostgreSQL 10 and 11, the hash index implementation has been significantly improved: hash indices now support write-ahead logging, can be replicated, and performance has improved as well.

Generalized Search Tree (GiST): The **Generalized Search Tree** (**GiST**) is an extensible indexing structure supported by PostgreSQL. The GiST index is based on a balanced tree-structure similar to a B-tree, but where several of the operations on the tree are not predefined, but instead must be specified by an access method implementor. GiST allows creation of specialized index types on top of the basic GiST template, without having to deal with the numerous internal details of a complete index implementation.

[2] As is conventional in the industry, the term B-tree is used in place of B^+-tree, and should be interpreted as referring to the B^+-tree data structure.

Examples of some indices built using GiST index include R-trees, as well as less conventional indices for multidimensional cubes and full-text search.

The GiST interface requires the access-method implementer to only implement certain operations on the data type being indexed, and specify how to arrange those data values in the GiST tree. New GiST access methods can be implemented by creating an operator class. Each GiST operator class may have a different set of strategies based on the search predicates implemented by the index. There are five support functions that an index operator class for GiST must provide, such as for testing set membership, for splitting sets of entries on page overflows, and for computing cost of inserting a new entry. GiST also allows four support functions that are optional, such as for supporting ordered scans, or to allow the index to contain a different type than the data type on which it is build. An index built on the GiST interface may be lossy, meaning that such an index might produce false matches; in that case, records fetched by the index need to have the index predicate rechecked, and some of the fetched records may fail the predicate.

PostgreSQL provides several index methods implemented using GiST such as indices for multidimensional cubes, and for storing key-value pairs. The original PostgreSQL implementation of R-trees was replaced by GiST operator classes which allowed R-trees to take advantage of the write-ahead logging and concurrency capabilities provided by the GiST index. The original R-tree implementation did not have these features, illustrating the benefits of using the GiST index template to implement specific indices.

Space-partitioned GiST (SP-GiST): Space-partitioned GiST indices leverage balanced search trees to facilitate disk-based implementations of a wide range of non-balanced data structures, such as quad-trees, k-d trees, and radix trees (tries). These data structures are designed for in-memory usage, with small node sizes and long paths in the tree, and thus cannot directly be used to implement disk-based indices. SP-GiST maps search tree nodes to disk pages in such a way that a search requires accessing only a few disk pages, even if it traverses a larger number of tree nodes. Thus, SP-GiST permits the efficient disk-based implementation of index structures originally designed for in-memory use. Similar to GiST, SP-GiST provides an interface with a high level of abstraction that allows the development of custom indices by providing appropriate access methods.

Generalized Inverted Index (GIN): The GIN index is designed for speeding up queries on multi-valued elements, such as text documents, JSON structures and arrays. A GIN stores a set of (key, posting list) pairs, where a posting list is a set of row IDs in which the key occurs. The same row ID might appear in multiple posting lists. Queries can specify multiple keys, for example with keys as words, GIN can be used to implement keyword indices.

GIN, like GiST, provides extensibility by allowing an index implementor to specify custom “strategies” for specific data types; the strategies specify, for example, how keys are extracted from indexed items and from query conditions, and how to determine whether a row that contains some of the key values in a query actually satisfies the query.

During query execution, GIN efficiently identifies index keys that overlap the search key, and computes a bitmap indicating which searched-for elements are members of the index key. To do so, GIN uses support function that extract members from a set, support functions that compare individual members. Another support function decides if the search predicate is satisfied, based on the bitmap and the original predicate. If the search predicate cannot be resolved without the full indexed attribute, the decision function must report a possible match and the predicate must be rechecked after retrieving the data item.

Each key is stored only once in a GIN, making GIN suitable for situations where many indexed items contain each key. However, updates are slower on GIN making them better for querying relatively static data, while GiST indices are preferred for workloads with frequent updates.

Block Range Index (BRIN): BRIN indices are designed for indexing such very large datasets that are naturally clustered on some specific attribute(s), with timestamps being a natural example. Many modern applications generate datasets with such characteristics. For example, an application collecting sensor measurements like temperature or humidity might have a timestamp column based on the time when each measurement was collected. In most cases, tuples for older measurements will appear earlier in the table storing the data. BRIN indices can speed up lookups and analytical queries with aggregates while requiring significantly less storage space than a typical B-tree index.

BRIN indices store some summary information for a group of pages that are physically adjacent in a table (block range). The summary data that a BRIN index will store depends on the operator class selected for each column of the index. For example, data types having a linear sort order can have operator classes that store the minimum and maximum value within each block range.

A BRIN index exploits the summary information stored for each block range to return tuples from only within the qualifying block ranges based on the query conditions. For example, in case of a table with a BRIN index on a timestamp, if the table is filtered by the timestamp, the BRIN is scanned to identify the block ranges that might have qualifying values and a list of block ranges is returned. The decision of whether a block range is to be selected or not is based on the summary information that the BRIN maintains for the given block range, such as the min and max value for timestamps. The selected block ranges might have false matches, that is the block range may contain items that do not satisfy the query condition. Therefore, the query executor re-evaluates the predicates on tuples in the selected block ranges and discards tuples that do not satisfy the predicate.

A BRIN index is typically very small, and scanning the index thus adds a very small overhead compared to a sequential scan, but may help in avoiding scanning large parts of a table that are found to not contain any matching tuples. The size of a BRIN index is determined by the size of the relation divided by the size of the block range. The smaller the block range, the larger the index, but at the same time the summary data stored can

be more precise, and thus more data blocks can potentially be skipped during an index scan.

32.3.2.2 Other Index Variations

For some of the index types described above, PostgreSQL supports more complex variations such as:

- **Multicolumn indices**: These are useful for conjuncts of predicates over multiple columns of a table. Multicolumn indices are supported for B-tree, GiST, GIN, and BRIN indices, and up to 32 columns can be specified in the index.
- **Unique indices**: Unique and primary-key constraints can be enforced by using unique indices in PostgreSQL. Only B-tree indices may be defined as being unique. PostgreSQL automatically creates a unique index when a unique constraint or primary key is defined for a table.
- **Covering indices**: A covering index is an index that includes additional attributes that are not part of the search key (as described earlier in Section 14.6). Such extra attributes can be added to allow a frequently used query to be answered using only the index, without accessing the underlying relation. Plans that use an index without accessing the underlying relation are called index-only plans, the implementation of index-only plans in PostgreSQL is described in Section 32.3.2.4. PostgreSQL uses the include clause to specify the extra attributes to be included in the index. A covering index can enhance query performance; however, the index build time increases since more data should be written and the index size becomes larger since the non-search attributes are duplicating data from the original table. Currently, only B-tree indices support covering indices.
- **Indices on expressions**: In PostgreSQL, it is possible to create indices on arbitrary scalar expressions of columns, and not just specific columns, of a table. An example is to support case-insensitive comparisons by defining an index on the expression *lower*(*column*). A query with the predicate *lower*(*column*) = 'value' cannot be efficiently evaluated using a regular B-tree index since matching records may appear at multiple locations in the index; on the other hand, an index on the expression *lower*(*column*) can be used to efficiently evaluate the query since all such records would map to the same index key.
  Indices on expressions have a higher maintenance cost (i.e., insert and update speed) but they can be useful when retrieval speed is more important.
- **Operator classes**: The specific comparison functions used to build, maintain, and use an index on a column are tied to the data type of that column. Each data type has associated with it a default *operator class* that identifies the actual operators that would normally be used for the data type. While the default operator class is sufficient for most uses, some data types might possess multiple “meaningful”

classes. For instance, in dealing with complex numbers, it might be desirable to sort a complex-number data type either by absolute value or by real part. In this case, two operator classes can be defined for the data type, and one of the operator classes can be chosen when creating the index.

- **Partial indices**: These are indices built over a subset of a table defined by a predicate. The index contains only entries for tuples that satisfy the predicate. An example of the use of a partial indices would be a case where a column contains a large number of occurrences of some of values. Such common values are not worth indexing, since index scans are not beneficial for queries that retrieve a large subset of the base table. A partial index can be built using a predicate that excludes the common values. Such an index would be smaller and would incur less storage and I/O cost than a full index. Such partial indices are also less expensive to maintain, since a large fraction of inserts and deletes will not affect the the index.

32.3.2.3 Index Construction

An index can be created using the **create index** command. For example, the following DDL statement creates a B-tree index on instructor salaries.

create index *inst_sal_idx* **on** *instructor* (*salary*);

This statement is executed by scanning the *instructor* relation to find row versions that might be visible to a future transaction, then sorting their index attributes and building the index structure. During this process, the building transaction holds a lock on the *instructor* relation that prevents concurrent **insert**, **delete**, and **update** statements. Once the process is finished, the index is ready to use and the table lock is released.

The lock acquired by the **create index** command may present a major inconvenience for some applications where it is difficult to suspend updates while the index is built. For these cases, PostgreSQL provides the **create index concurrently** variant, which allows concurrent updates during index construction. This is achieved by a more complex construction algorithm that scans the base table twice. The first table scan builds an initial version of the index, in a way similar to normal index construction described above. This index may be missing tuples if the table was concurrently updated; however, the index is well formed, so it is flagged as being ready for insertions. Finally, the algorithm scans the table a second time and inserts all tuples it finds that still need to be indexed. This scan may also miss concurrently updated tuples, but the algorithm synchronizes with other transactions to guarantee that tuples that are updated during the second scan will be added to the index by the updating transaction. Hence, the index is ready to use after the second table scan. Since this two-pass approach can be expensive, the plain **create index** command is preferred if it is easy to suspend table updates temporarily.

32.3.2.4 Index-Only Scans

An index-only scan allows for processing a query using only an index, without access the table records. Traditionally, indices in PostgreSQL are secondary indices, meaning that each index is stored separately from the table's main data (heap). In the normal index scan scenario, PostgreSQL initially uses the index to locate qualifying tuples and then accesses the heap to retrieve the tuples. Accessing the heap via an index scan might result in a large number of random accesses, since the tuples to be accessed might be anywhere in the heap; index scans can potentially have a high query execution time if a large number of records are retrieved. In contrast, an index-only scan can allow the query to be executed without fetching the tuples, provided the query accesses only index attributes. This can significantly decrease the number I/O operations required, and improve performance correspondingly.

To apply an index-only scan plan during query execution, the query must reference only attributes already stored in the index and the index should support index-only scans. As of PostgreSQL 11, index-only scans are supported with B-trees and some operator classes of GiST and SP-GiST indices. In practice, the index must physically store, or else be able to reconstruct, the original data value for each index entry. Thus, for example, a GIN index cannot support index-only scan since each index entry typically holds only part of the original data value.

In PostgreSQL, using an index-only scan plan does not guarantee that no heap pages will be accessed, due to the presence of multiple tuple versions in PostgreSQL; an index may contain entries for tuple versions that should not be visible to the transaction performing the scan. To check if the tuple is visible, PostgreSQL normally performs a heap page access, to find timestamp information for the tuple. However, PostgreSQL provides a clever optimization that can avoid heap page access in many cases. For each heap relation PostgreSQL maintains a visibility map. The visibility map tracks the pages that contain only tuples that are visible to all active transactions and therefore they do not contain any tuples that need to be vacuumed. Visibility information is stored only in heap entries and not in index entries. As a result, accessing a tuple using an index-only scan will require a heap access if the tuple has been recently modified. Otherwise the heap access can be avoided by checking the visibility map bit for the corresponding heap page. If the bit is set, the tuple is visible to all transactions, and so the data can be returned from the index. On the other hand, if the bit is not set, the heap will be accessed and the performance for this retrieval will be similar to a traditional index access.

The visibility map bit is set during vacuum, and reset whenever a tuple in the heap page is updated. Overall, the more the heap pages that have their all-visible map bits set the higher the performance benefit from an index-only scan. The visibility map is much smaller than the heap file since it requires only two bits per page; thus, very little I/O is required to access it.

In PostgreSQL index-only scans can also be performed on covering indices, which can store attributes other than the index key. Index-only scans on the covering index can

allow efficient sequential access to tuples in the key order, avoiding expensive random access that would be otherwise required by a secondary-index based access. An index-only scan can be used provided all attributes required by the query are contained in the index key or in the covering attributes.

32.3.3 Partitioning

Table partitioning in PostgreSQL allows a table to be split into smaller physical pieces, based on the value of partitioning attributes. Partitioning can be quite beneficial in certain scenarios; for example, it can improve query performance when the query includes predicates on the partitioning attributes, and the matching tuples are in a single partition or a small number of partitions. Table partitioning can also reduce the overhead of bulk loading and deletion in some cases by adding or removing partitions without modifying existing partitions. Partitioning can also make maintenance operations such as VACUUM and REINDEX faster. Further, indices on the partitions are smaller than a index on the whole table, thus it is more likely to fit into memory. Partitioning a relation is a good idea as long as most queries that access the relation include predicates on the partitioning attributes. Otherwise, the overhead of accessing multiple partitions can slow down query processing to some extent.

As of version 11, PostgreSQL comes with three types of partitioning tables:

1. **Range Partitioning**: The table is partitioned into ranges (e.g., date ranges) defined by a key column or set of columns. The range of values in each partition is assigned based on some partitioning expression. The ranges should be contiguous and non-overlapping.
2. **List Partitioning**: The table is partitioned by explicitly listing the set of discrete values that should appear in each partition.
3. **Hash Partitioning**: The tuples are distributed across different partitions according to a hash partition key. Hash partitioning is ideal for scenarios in which there is no natural partitioning key or details about data distribution.

Partitioning in PostgreSQL can be implemented manually using table inheritance. However, a simpler way of implementing partitioning is through the declarative partitioning feature of PostgreSQL. Declarative partitioning allows a user to create a partitioned table by specifying the partitioning type and the list of columns or expressions to be used as the partition key. For example, consider the *takes* relation; a clause

partition by range(*year*)

can be added at the end of the **create table** specification for the *takes* relation to specify that the relation should be partitioned by the *year* attribute.

Then, one or more partitions must be created, with each partition specifying the bounds that correspond to the partitioning method and partition key of the parent, as illustrated below:

create table *takes_till_2017* **partition of** *takes* **for values from** (1900) to (2017);
create table *takes_2017* **partition of** *takes* **for values from** (2017) to (2018);
create table *takes_2018* **partition of** *takes* **for values from** (2018) to (2019);
create table *takes_2019* **partition of** *takes* **for values from** (2019) to (2020);
create table *takes_from_2020* **partition of** *takes* **for values from** (2020) to (2100);

New tuples are routed to the proper partitions according to the selected partition key. Partition key ranges must not overlap, and there must be a partition defined for each valid key value. The query planner of PostgreSQL can exploit the partitioning information to eliminate unnecessary partition accesses during query processing.

Each partition as above is a normal PostgreSQL table, and it is possible to specify a tablespace and storage parameters for each partition separately. Partitions may have their own indexes, constraints and default values, distinct from those of other partitions of the same table. However, there is no support for foreign keys referencing partitioned tables, or for exclusion constraints[3] spanning all partitions.

Turning a table into a partitioned table or vice versa is not supported; however, it is possible to add a regular or partitioned table containing data as a partition of a partitioned table, or remove a partition from a partitioned table turning it into a standalone table.

32.4 Query Processing and Optimization

When PostgreSQL receives a query, the query is first parsed into an internal representation, which goes through a series of transformations, resulting in a query plan that is used by the **executor** to process the query.

32.4.1 Query Rewrite

The first stage of a query's transformation is the **rewrite** stage, which is responsible for implementing the PostgreSQL **rules** system. In PostgreSQL, users can create **rules** that are fired on different query structures such as **update**, **delete**, **insert**, and **select** statements. A view is implemented by the system by converting a view definition into a **select** rule. When a query involving a **select** statement on the view is received, the **select** rule for the view is fired, and the query is rewritten using the definition of the view.

A rule is registered in the system using the **create rule** command, at which point information on the rule is stored in the catalog. This catalog is then used during query rewrite to uncover all candidate rules for a given query.

[3] **Exclusion constraints** in PostgreSQL allow a constraint on each row that can involve other rows; for example, such a constraint can specify that there is no other row with the same key value, or there is no other row with an overlapping range. Efficient implementation of exclusion constraints requires the availability of appropriate indices. See the PostgreSQL manuals for more details.

The rewrite phase first deals with all **update**, **delete**, and **insert** statements by firing all appropriate rules. Such statements might be complicated and contain **select** clauses. Subsequently, all the remaining rules involving only **select** statements are fired. Since the firing of a rule may cause the query to be rewritten to a form that may require another rule to be fired, the rules are repeatedly checked on each form of the rewritten query until a fixed point is reached and no more rules need to be fired.

32.4.2 Query Planning and Optimization

Once the query has been rewritten, it is subject to the planning and optimization phase. Here, each query block is treated in isolation and a plan is generated for it. This planning begins bottom-up from the rewritten query's innermost subquery, proceeding to its outermost query block.

The optimizer in PostgreSQL is, for the most part, cost based. The idea is to generate an access plan whose estimated cost is minimal. The cost model includes as parameters the I/O cost of sequential and random page fetches, as well as the CPU costs of processing heap tuples, index tuples, and simple predicates.

Optimization of a query can be done using one of two approaches:

- **Standard planner**: The standard planner uses the the bottom-up dynamic programming algorithm for join order optimization, which we saw earlier in Section 16.4.1, which is often referred to as the System R optimization algorithm.
- **Genetic query optimizer**: When the number of tables in a query block is very large, System R's dynamic programming algorithm becomes very expensive. Unlike other commercial systems that default to greedy or rule-based techniques, PostgreSQL uses a more radical approach: a genetic algorithm that was developed initially to solve traveling-salesman problems. There exists anecdotal evidence of the successful use of genetic query optimization in production systems for queries with around 45 tables.

Since the planner operates in a bottom-up fashion on query blocks, it is able to perform certain transformations on the query plan as it is being built. One example is the common subquery-to-join transformation that is present in many commercial systems (usually implemented in the rewrite phase). When PostgreSQL encounters a noncorrelated subquery (such as one caused by a query on a view), it is generally possible to pull up the planned subquery and merge it into the upper-level query block. PostgreSQL is able to decorrelate many classes of correlated subqueries, but there are other classes of queries that it is not able to decorrelate. (Decorrelation is described in more detail in Section 16.4.4.)

The query optimization phase results in a *query plan*, which is a tree of relational operators. Each operator represents a specific operation on one or more sets of tuples. The operators can be unary (for example, sort, aggregation), binary (for example, nested-loop join), or *n*-ary (for example, set union).

Crucial to the cost model is an accurate estimate of the total number of tuples that will be processed at each operator in the plan. These estimates are inferred by the optimizer on the basis of statistics that are maintained on each relation in the system. These statistics include the total number of tuples for each relation and average tuple size. PostgreSQL also maintains statistics about each column of a relation, such as the column cardinality (that is, the number of distinct values in the column), a list of most common values in the table and the number of occurrences of each common value, and a histogram that divides the column's values into groups of equal population. In addition, PostgreSQL also maintains a statistical correlation between the physical and logical row orderings of a column's values—this indicates the cost of an index scan to retrieve tuples that pass predicates on the column. The DBA must ensure that these statistics are current by running the **analyze** command periodically.

32.4.3 Query Executor

The executor module is responsible for processing a query plan produced by the optimizer. The executor is based on the demand-driven pipeline model (described in Section 15.7.2.1), where each operator implements the *iterator* interface with a set of four functions: `open()`, `next()`, `rescan()`, and `close()`). PostgreSQL iterators have an extra function, `rescan()`, which is used to reset a subplan (say for an inner loop of a join) with new values for parameters such as index keys. Some of the important categories of operators are as follows:

1. **Access methods**: Access methods are operators that are used to retrieve data from disk, and include sequential scans of heap files, index scans, and bitmap index scans.
    - **Sequential scans**: The tuples of a relation are scanned sequentially from the first to last blocks of the file. Each tuple is returned to the caller only if it is "visible" according to the transaction isolation rules (Section 32.5.1.1).
    - **Index scans**: Given a search condition such as a range or equality predicate, an index scan returns a set of matching tuples from the associated heap file. In a typical case, the operator processes one tuple at a time, starting by reading an entry from the index and then fetching the corresponding tuple from the heap file. This can result in a random page fetch for each tuple in the worst case. The cost of accessing the heap file can be alleviated if an index-only scan is used that allows for retrieving data directly from the index (see Section 32.3.2.4 for more details).
    - **Bitmap index scans**: A bitmap index scan reduces the danger of excessive random page fetches in index scans. To do so, processing of tuples is done in two phases.

a. The first phase reads all index entries and populates a bitmap that contains one bit per heap page; the tuple ID retrieved from the index scan is used to set the bit of the corresponding page.

b. The second phase fetches heap pages whose bit is set, scanning the bitmap in sequential order. This guarantees that each heap page is accessed only once, and increases the chance of sequential page fetches. Once a heap page is fetched, the index predicate is rechecked on all the tuples in the page, since a page whose bit is set may well contain tuples that do not satisfy the index predicate.

Moreover, bitmaps from multiple indexes can be merged and intersected to evaluate complex Boolean predicates before accessing the heap.

2. **Join methods**: PostgreSQL supports three join methods: sorted merge joins, nested-loop joins (including index-nested loop variants for accessing the inner relation using an index), and a hybrid hash join.

3. **Sort**: Small relations are sorted in-memory using quicksort, while larger are sorted using an external sort algorithm. Initially, the input tuples are stored in an unsorted array as long as there is available working memory for the sort operation. If all the tuples fit in memory, the array is sorted using quicksort, and the sorted tuples can be accessed by sequentially scanning the array. Otherwise, the input is divided into sorted runs by using replacement selection; replacement selection uses a priority tree implemented as a heap, and can generate sorted runs that are bigger than the available memory. The sorted runs are stored in temporary files and then merged using a polyphase merge.

4. **Aggregation**: Grouped aggregation in PostgreSQL can be either sort-based or hash-based. When the estimated number of distinct groups is very large, the former is used; otherwise, an in-memory hash-based approach is preferred.

32.4.4 Parallel Query Support

PostgreSQL can generate parallel query plans[4] to leverage multiple CPUs/cores, which can significantly improve query execution performance. Nevertheless, not all queries can benefit from parallel plans, either due to implementation limitations or because the serial query plan is still a better option. The optimizer is responsible for determining whether a parallel plan is the faster execution strategy or not. As of PostgreSQL version 11, only read-only queries can exploit parallel plans. A parallel query plan will not be generated if the query might be suspended during processing (e.g., a PL/pgSQL loop of the form **for** target **in** query **loop** .. **end loop**), if the query uses any function marked as **parallel unsafe**, if the query runs in another parallel query. Further, as of

[4] See https://www.postgresql.org/docs/current/parallel-query.html.

PostgreSQL 11, parallel query plans will not be used if the transaction isolation level is set to serializable, although this may be fixed in future versions.

When a parallel query operator is used, the master backend process coordinates the parallel execution. It is responsible for spawning the required number of workers, executing the non-parallel activity while contributing to parallel execution as one of the workers. The planner determines the number of the background workers that will be used to process the child plan of the *Gather* node.

A parallel query plan includes a *Gather* or *Gather Merge* node which has exactly one child plan. This child plan is the part of the plan that will be executed in parallel. If the root node is *Gather* or *Gather Merge*, then the whole query can be executed in parallel. The master backend executes the *Gather* or *Gather Merge* node. The *Gather* node is responsible for retrieving the tuples generated by the background workers. A *Gather Merge* node is used when the parallel part of the plan produces tuples in sorted order. The background workers and the master backend process communicate through the shared memory area.

PostgreSQL has parallel-aware flavors for the basic query operations. It supports three types of parallel scans; namely, parallel sequential scan, parallel bitmap heap scan and parallel index/index-only scan (only for B-tree indexes). PostgreSQL also supports parallel versions of nested loop, hash and merge joins. In a join operation, at least one of the tables is scanned by multiple background workers. Each background worker additionally scans the inner table of the join and then forwards the computed tuples to the master backend coordinator process. For nested-loop join and merge join the inner side of the join is always non-parallel.

PostgreSQL can also generate parallel plans for aggregation operations. In this case, the aggregation happens in two steps: (a) each background worker produces a partial result for a subset of the data, and (b) the partial results are collected to the master backend process which computes the final result using the partial results generated by the workers.

32.4.5 Triggers and Constraints

In PostgreSQL (unlike some commercial systems) active-database features such as triggers and constraints are not implemented in the rewrite phase. Instead they are implemented as part of the query executor. When the triggers and constraints are registered by the user, the details are associated with the catalog information for each appropriate relation and index. The executor processes an **update**, **delete**, and **insert** statement by repeatedly generating tuple changes for a relation. For each row modification, the executor explicitly identifies, fires, and enforces candidate triggers and constraints, before or after the change as required.

32.4.6 Just-in-Time (JIT) Compilation

The Just-in-Time compilation (JIT) functionality was introduced in PostgreSQL version 11. The physical data independence abstraction supported by relational databases pro-

vides significant flexibility by hiding many low-level details. However, that flexibility can incur a performance hit due to the interpretation overhead (e.g., function calls, unpredictable branches, high number of instructions). For example, computing the qualifying tuples for any arbitrary SQL expression requires evaluating predicates that can use any of the supported SQL datatypes (e.g.,integer,double); the predicate evaluation function must be able to handle all these data types, and also handle sub-expressions containing other operators. The evaluation function is, in effect, an interpreter that processes the execution plan. With JIT compilation, a generic interpreted program can be compiled at query execution time into a native-code program that is tailored for the specific data types used in a particular expression. The resultant compiled code can execute significantly faster than the original interpreted function.

As of PostgreSQL 11, PostgreSQL exploits JIT compilation to accelerate expression evaluation and tuple deforming (which is explained shortly). Those operations were chosen because they are executed very frequently (per tuple) and therefore have a high cost for analytics queries that process large amounts of data. PostgreSQL accelerates expression evaluation (i.e., the code path used to evaluate WHERE clause predicates, expressions in target lists, aggregates and projections) by generating tailored code to each case, depending on the data types of the attributes. Tuple deforming is the process of transforming an on-disk tuple into its in-memory representation. JIT compilation in PostgreSQL creates a transformation function specific to the table layout and the columns to be extracted. JIT compilation may be added for other operations in future releases.

JIT compilation is primarily beneficial for long-running CPU-bound queries. For short-running queries the overhead of performing JIT compilation and optimizations can be higher than the savings in execution time. PostgreSQL selects whether JIT optimizations will be applied during planning, based on whether the estimated query cost is above some threshold.

PostgreSQL uses LLVM to perform JIT compilation; LLVM allows systems to generate a device independent assembly language code, which is then optimized and compiled to machine code specific to the hardware platform. As of PostgreSQL 11, JIT compilation support is not enabled by default, and is used only when PostgreSQL is built using the `–with-llvm` option. The LLVM dependent code is loaded on-demand from a shared library.

32.5 Transaction Management in PostgreSQL

Transaction management in PostgreSQL uses both snapshot isolation (described earlier in Section 18.8), and two-phase locking. Which one of the two protocols is used depends on the type of statement being executed. For DML statements the snapshot isolation technique is used; the snapshot isolation scheme is referred to as the multiversion concurrency control (MVCC) scheme in PostgreSQL. Concurrency control for DDL statements, on the other hand, is based on standard two-phase locking.

32.5.1 Concurrency Control

Since the concurrency control protocol used by PostgreSQL depends on the *isolation level* requested by the application, we begin with an overview of the isolation levels offered by PostgreSQL. We then describe the key ideas behind the MVCC scheme, followed by a discussion of their implementation in PostgreSQL and some of the implications of MVCC. We conclude this section with an overview of locking for DDL statements and a discussion of concurrency control for indices.

32.5.1.1 Isolation Levels

The SQL standard defines three weak levels of consistency, in addition to the serializable level of consistency. The purpose of providing the weak consistency levels is to allow a higher degree of concurrency for applications that do not require the strong guarantees that serializability provides. Examples of such applications include long-running transactions that collect statistics over the database and whose results do not need to be precise.

The SQL standard defines the different isolation levels in terms of phenomena that violate serializability. The phenomena are called *dirty read*, *nonrepeatable read*, *phantom read*, and *serialization anomaly* and are defined as follows:

- **Dirty read.** The transaction reads values written by another transaction that has not committed yet.
- **Nonrepeatable read.** A transaction reads the same object twice during execution and finds a different value the second time, although the transaction has not changed the value in the meantime.
- **Phantom read.** A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed as a result of another recently committed transaction.
- **Serialization anomaly.** A successfully committed group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

Each of the above phenomena violates transaction isolation, and hence violates serializability. Figure 32.3 shows the definition of the four SQL isolation levels specified in the SQL standard—read uncommitted, read committed, repeatable read, and serializable—in terms of these phenomena. In PostgreSQL the user can select any of the four transaction isolation levels (using the command **set transaction**); however, PostgreSQL implements only three distinct isolation levels. A request to set transaction isolation level to read uncommitted is treated the same as a request to set the isolation level to read committed. The default isolation level is read committed.

Isolated level	*Dirty Read*	*Non repeatable Read*	*Phantom Read*
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeated Read	No	No	Maybe
Serializable	No	No	No

Figure 32.3 Definition of the four standard SQL isolation levels.

32.5.1.2 Concurrency Control for DML Commands

The MVCC scheme used in PostgreSQL is an implementation of the snapshot isolation protocol, which was described earlier in Section 18.8. The key idea behind MVCC is to maintain different versions of each row that correspond to instances of the row at different points in time. This allows a transaction to see a consistent **snapshot** of the data, by selecting the most recent version of each row that was committed before taking the snapshot. The MVCC protocol uses snapshots to ensure that every transaction sees a consistent view of the database: before executing a command, the transaction chooses a snapshot of the data and processes the row versions that are either in the snapshot or created by earlier commands of the same transaction. This view of the data is transaction-consistent, since the snapshot includes only committed transactions, but the snapshot is not necessarily equal to the current state of the data.

The motivation for using MVCC is that, unlike with locking, readers never block writers, and writers never block readers. Readers access the version of a row that is part of the transaction's snapshot. Writers create their own separate copy of the row to be updated. Section 32.5.1.3 shows that the only conflict that causes a transaction to be blocked arises if two writers try to update the same row. In contrast, under the standard two-phase locking approach, both readers and writers might be blocked, since there is only one version of each data object and both read and write operations are required to obtain a lock before accessing any data.

The basic snapshot isolation protocol has several benefits over locking, but unfortunately does not guarantee serializability, as we saw earlier in Section 18.8.3. The Serializable Snapshot Isolation (SSI), which provides the benefits of snapshot isolation, while also guaranteeing serializability, was introduced in PostgreSQL 9.1. The key ideas behind SSI are summarized in Section 18.8.3, and more details may be found in the references in bibliographic notes at the end of the chapter.

SSI retains many of the performance benefits of snapshot isolation and at the same time guarantees true serializability. SSI runs transactions using snapshot isolation, checks for conflicts between concurrent transactions at runtime, and aborts transactions when anomalies might happen.

32.5.1.3 Implementation of MVCC

At the core of PostgreSQL MVCC is the notion of *tuple visibility*. A PostgreSQL tuple refers to a version of a row. Tuple visibility defines which of the potentially many ver-

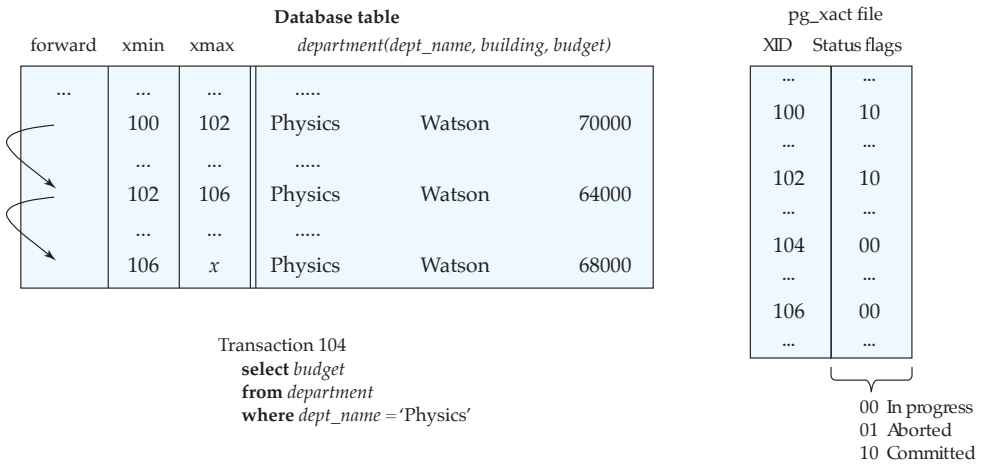

Figure 32.4 The PostgreSQL data structures used for MVCC.

sions of a row in a table is valid within the context of a given statement or transaction. A transaction determines tuple visibility based on a database snapshot that is chosen before executing a command.

A tuple is visible for a transaction T if the following two conditions hold:

1. The tuple was created by a transaction that committed before transaction T took its snapshot.
2. Updates to the tuple (if any) were executed by a transaction that is either
   - aborted, *or*
   - started running after T took its snapshot, *or*
   - was active when T took its snapshot.

To be precise, a tuple is also visible to T if it was created by T and not subsequently updated by T. We omit the details of this special case for simplicity.

The goal of the above conditions is to ensure that each transaction sees a consistent view of the data. PostgreSQL maintains the following state information to check these conditions efficiently:

- A *transaction ID*, which serves as a start timestamp, is assigned to every transaction at transaction start time. PostgreSQL uses a logical counter for assigning transaction IDs.
- A log file called *pg_xact* contains the current status of each transaction. The status can be either in progress, committed, or aborted.

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

Figure 32.5 The *department* relation.

- Each tuple in a table has a header with three fields: *xmin*, which contains the transaction ID of the transaction that created the tuple and which is therefore also called the *creation-transaction ID*; *xmax*, which contains the transaction ID of the replacing/deleting transaction (or *null* if not deleted/replaced) and which is also referred to as the *expire-transaction ID*; and a forward link to new versions of the same logical row, if there are any.
- A *SnapshotData* data structure is created either at transaction start time or at query start time, depending on the isolation level (described in more detail below). Its main purpose is to decide whether a tuple is visible to the current command. The *SnapshotData* stores information about the state of transactions at the time it is created, which includes a list of active transactions and *xmax*, a value equal to 1 + the highest ID of any transaction that has started so far. The value *xmax* serves as a "cutoff" for transactions that may be considered visible.

Figure 32.4 illustrates some of this state information through a simple example involving a database with only one table, the *department* table from Figure 32.5. The *department* table has three columns, the name of the department, the building where the department is located, and the budget of the department. Figure 32.4 shows a fragment of the *department* table containing only the (versions of) the row corresponding to the Physics department. The tuple headers indicate that the row was originally created by transaction 100, and later updated by transaction 102 and transaction 106. Figure 32.4 also shows a fragment of the corresponding *pg_xact* file. On the basis of the *pg_xact* file, transactions 100 and 102 are committed, while transactions 104 and 106 are in progress.

Given the above state information, the two conditions that need to be satisfied for a tuple to be visible can be rewritten as follows:

1. The creation-transaction ID in the tuple header
    a. is a committed transaction according to the *pg_xact* file, *and*

   b. is less than the cutoff transaction ID *xmax* recorded by *SnapshotData*, *and*
   c. is not one of the active transactions stored in *SnapshotData*.

2. The expire-transaction ID, if it exists,
   a. is an aborted transaction according to the *pg_xact* file, *or*
   b. is greater than or equal to the cutoff transaction ID *xmax* recorded by *SnapshotData*, *or*
   c. is one of the active transactions stored in *SnapshotData*.

Consider the example database in Figure 32.4 and assume that the *SnapshotData* used by transaction 104 simply uses 103 as the cutoff transaction ID *xmax* and does not show any earlier transactions to be active. In this case, the only version of the row corresponding to the Physics department that is visible to transaction 104 is the second version in the table, created by transaction 102. The first version, created by transaction 100, is not visible, since it violates condition 2: The expire-transaction ID of this tuple is 102, which corresponds to a transaction that is not aborted and that has a transaction ID less than or equal to 103. The third version of the Physics tuple is not visible, since it was created by transaction 106, which has a transaction ID larger than transaction 103, implying that this version had not been committed at the time *SnapshotData* was created. Moreover, transaction 106 is still in progress, which violates another one of the conditions. The second version of the row meets all the conditions for tuple visibility.

The details of how PostgreSQL MVCC interacts with the execution of SQL statements depends on whether the statement is an **insert**, **select**, **update**, or **delete** statement. The simplest case is an **insert** statement, which may simply create a new tuple based on the data in the statement, initialize the tuple header (the creation ID), and insert the new tuple into the table. Unlike two-phase locking, this does not require any interaction with the concurrency-control protocol unless the insertion needs to be checked for integrity conditions, such as uniqueness or foreign key constraints.

When the system executes a **select**, **update**, or **delete** statement the interaction with the MVCC protocol depends on the isolation level specified by the application. If the isolation level is read committed, the processing of a new statement begins with creating a new *SnapshotData* data structure (independent of whether the statement starts a new transaction or is part of an existing transaction). Next, the system identifies *target tuples*, that is, the tuples that are visible with respect to the *SnapshotData* and that match the search criteria of the statement. In the case of a **select** statement, the set of target tuples make up the result of the query.

In the case of an **update** or **delete** statement in read committed mode, the snapshot isolation protocol used by PostgreSQL requires an extra step after identifying the target tuples and before the actual update or delete operation can take place. The reason is that visibility of a tuple ensures only that the tuple has been created by a transaction that committed before the **update**/**delete** statement in question started. However, it is possi-

ble that, since query start, this tuple has been updated or deleted by another concurrently executing transaction. This can be detected by looking at the expire-transaction ID of the tuple. If the expire-transaction ID corresponds to a transaction that is still in progress, it is necessary to wait for the completion of this transaction first. If the transaction aborts, the **update** or **delete** statement can proceed and perform the actual modification. If the transaction commits, the search criteria of the statement need to be evaluated again, and only if the tuple still meets these criteria can the row be modified. If the row is to be deleted, the main step is to update the expire-transaction ID of the old tuple. A row update also performs this step, and additionally creates a new version of the row, sets its creation-transaction ID, and sets the forward link of the old tuple to reference the new tuple.

Going back to the example from Figure 32.4, transaction 104, which consists of a **select** statement only, identifies the second version of the Physics row as a target tuple and returns it immediately. If transaction 104 were an update statement instead, for example, trying to increment the budget of the Physics department by some amount, it would have to wait for transaction 106 to complete. It would then re-evaluate the search condition and, only if it is still met, proceed with its update.

Using the protocol described above for **update** and **delete** statements provides only the read-committed isolation level. Serializability can be violated in several ways. First, nonrepeatable reads are possible. Since each query within a transaction may see a different snapshot of the database, a query in a transaction might see the effects of an **update** command completed in the meantime that were not visible to earlier queries within the same transaction. Following the same line of thought, phantom reads are possible when a relation is modified between queries.

In order to provide the PostgreSQL serializable isolation level, PostgreSQL MVCC eliminates violations of serializability in two ways: First, when it is determining tuple visibility, all queries within a transaction use a snapshot as of the start of the transaction, rather than the start of the individual query. This way successive queries within a transaction always see the same data.

Second, the way updates and deletes are processed is different in serializable mode compared to read-committed mode. As in read-committed mode, transactions wait after identifying a visible target row that meets the search condition and is currently updated or deleted by another concurrent transaction. If the concurrent transaction that executes the update or delete aborts, the waiting transaction can proceed with its own update. However, if the concurrent transaction commits, there is no way for PostgreSQL to ensure serializability for the waiting transaction. Therefore, the waiting transaction is rolled back and returns the following error message: “could not serialize access due to read/write dependencies among transactions“. It is up to the application to handle an error message like the above appropriately, by aborting the current transaction and restarting the entire transaction from the beginning.

Further, to ensure serializability, the serializable snapshot-isolation technique (which is used when the isolation level is set to serializable) tracks read-write conflicts

between transactions, and forces rollback of transactions whenever certain patterns of conflicts are detected.

Further, to guarantee serializability, the phantom-phenomenon (Section 18.4.3) must be avoided; the problem occurs when a transaction reads a set of tuples satisfying a predicate, and a concurrent transaction performs an update that creates a new tuple satisfying the predicate, or updates a tuple in a way that results in the tuple satisfying the predicate, when it did not do so earlier.

To avoid the phantom phenomenon, the SSI implementation in PostgreSQL uses predicate locking, using ideas from the index locking technique described in Section 18.4.3, but with modifications to work correctly with SSI. Predicate locking helps to detect when a write might have an impact on the result of a predicate read by a concurrent transaction. These locks do not cause any blocking and therefore can not cause a deadlock. They are used to identify and flag dependencies among concurrent serializable transactions which in certain combinations can lead to serialization anomalies. Predicate locks show up in the pg_locks system view with a mode of SIReadLock. The particular locks acquired during execution of a query depend on the plan used by the query. A read-only transaction may be able to release its SIRead lock before completion, if it detects that no conflicts can still occur which could lead to a serialization anomaly. On the other hand, SIRead locks may need to be kept past transaction commit, until overlapping read write transactions complete.

32.5.1.4 Implications of Using MVCC

Using the PostgreSQL MVCC scheme has implications in three different areas:

1. An extra burden is placed on the storage system, since it needs to maintain different versions of tuples.
2. The development of concurrent applications takes some extra care, since PostgreSQL MVCC can lead to subtle, but important, differences in how concurrent transactions behave, compared to systems where standard two-phase locking is used.
3. The performance of MVCC depends on the characteristics of the workload running on it.

The implications of MVCC in PostgreSQL are described in more detail below.

Creating and storing multiple versions of every row can lead to excessive storage consumption. The space occupied by a version cannot be freed by the transaction that deletes the tuple or creates a new version. Instead, a tuple version can be freed only later, after checking that the tuple version cannot be visible to any active or future transactions. The task of freeing space is nontrivial, because indices may refer to the location of an old tuple version, so these references need to be deleted before reusing the space.

In general, versions of tuples are freed up by the vacuum process of PostgreSQL. The vacuum process can be initiated by a command, but PostgreSQL employs a background process to **vacuum** tables automatically. The vacuum process first scans the heap, and whenever it finds a tuple version that cannot be accessed by any current/future transaction, it marks the tuple as "dead". The vacuum process then scans all indices of the relation, and removes any entries that point to dead tuples. Finally, it rescans the heap, physically deleting tuple versions that were marked as dead earlier.

PostgreSQL also supports a more aggressive form of tuple reclaiming in cases where the creation of a version does not affect the attributes used in indices, and further the old and new tuple versions are on the same page. In this case no index entry is created for the new tuple version, but instead a link is added from the old tuple version in the heap page to the new tuple version (which is also on the same heap page). An index lookup will first find the old version, and if it determines that the version is not visible to the transaction, the version chain is followed to find the appropriate version. When the old version is no longer visible to any transaction, the space for the old version can be reclaimed in the heap page by some clever data structure tricks within the page, without touching the index.

The **vacuum** command offers two modes of operation: Plain **vacuum** simply identifies tuples that are not needed, and makes their space available for reuse. This form of the command executes as described above, and can operate in parallel with normal reading and writing of the table. **Vacuum full** does more extensive processing, including moving of tuples across blocks to try to compact the table to the minimum number of disk blocks. This form is much slower and requires an exclusive lock on each table while it is being processed.

PostgreSQL's approach to concurrency control performs best for workloads containing many more reads than updates, since in this case there is a very low chance that two updates will conflict and force a transaction to roll back. Two-phase locking may be more efficient for some update-intensive workloads, but this depends on many factors, such as the length of transactions and the frequency of deadlocks.

32.5.1.5 DDL Concurrency Control

The MVCC mechanisms described in the previous section do not protect transactions against operations that affect entire tables, for example, transactions that drop a table or change the schema of a table. Toward this end, PostgreSQL provides explicit locks that DDL commands are required to acquire before starting their execution. These locks are always table based (rather than row based) and are acquired and released in accordance with the strict two-phase locking protocol.

Figure 32.6 lists all types of locks offered by PostgreSQL, which locks they conflict with, and some commands that use them (the **create index concurrently** command is covered in Section 32.3.2.3). The names of the lock types are often historical and do not necessarily reflect the use of the lock. For example, all the locks are table-level locks, although some contain the word "row" in the name. DML commands acquire

Lock name	*Conflicts with*	*Acquired by*
ACCESS SHARE	ACCESS EXCLUSIVE	**select** query
ROW SHARE	EXCLUSIVE ACCESS EXCLUSIVE	**select for update** query **select for share** query
ROW EXCLUSIVE	SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**update** **delete** **insert** queries
SHARE UPDATE EXCLUSIVE	SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**vacuum** **analyze** **create index concurrently**
SHARE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	**create index**
SHARE ROW EXCLUSIVE	ROW EXCLUSIVE SHARE UPDATE EXCLUSIVE SHARE SHARE ROW EXCLUSIVE EXCLUSIVE ACCESS EXCLUSIVE	---
EXCLUSIVE	All except ACCESS SHARE	---
ACCESS EXCLUSIVE	All modes	**drop table** **alter table** **vaccum full**

Figure 32.6 Table-level lock modes.

only locks of the first three types. These three lock types are compatible with each other, since MVCC takes care of protecting these operations against each other. DML commands acquire these locks only for protection against DDL commands.

While their main purpose is providing PostgreSQL internal concurrency control for DDL commands, all locks in Figure 32.6 can also be acquired explicitly by PostgreSQL applications through the **lock table** command.

Locks are recorded in a lock table that is implemented as a shared-memory hash table keyed by a signature that identifies the object being locked. If a transaction wants to acquire a lock on an object that is held by another transaction in a conflicting mode, it needs to wait until the lock is released. Lock waits are implemented through semaphores, each of which is associated with a unique transaction. When waiting for a lock, a transaction actually waits on the semaphore associated with the transaction holding the lock. Once the lock holder releases the lock, it will signal the waiting trans-

action(s) through the semaphore. By implementing lock waits on a per-lock-holder basis, rather than on a per-object basis, PostgreSQL requires at most one semaphore per concurrent transaction, rather than one semaphore per lockable object.

Deadlock detection is triggered if a transaction has been waiting for a lock for more than 1 second. The deadlock-detection algorithm constructs a wait-for graph based on the information in the lock table and searches this graph for circular dependencies. If it finds any, meaning a deadlock was detected, the transaction that triggered the deadlock detection aborts and returns an error to the user. If no cycle is detected, the transaction continues waiting on the lock. Unlike some commercial systems, PostgreSQL does not tune the lock time-out parameter dynamically, but it allows the administrator to tune it manually. Ideally, this parameter should be chosen on the order of a transaction lifetime, in order to optimize the trade-off between the time it takes to detect a deadlock and the work wasted for running the deadlock detection algorithm when there is no deadlock.

32.5.1.6 Locking and Indices

Indices in PostgreSQL allow for concurrent access by multiple transactions. B-tree, GIN, GiST and SP-GiST indices use short-term share/exclusive page-level locks for read/write access which are released immediately after each index row is fetched or inserted. On the other hand, hash indices use share/exclusive hash-bucket-level locks for read/write access, and the locks are released after the whole bucket is processed. This might cause deadlock since the locks are held longer than one index operation. Indices that support predicate locking acquire SIRead locks on index pages that are accessed when searching for tuples that satisfy the predicate used in the predicate read.

32.5.2 Recovery

PostgreSQL employs write-ahead log (WAL) based recovery to ensure atomicity and durability. The approach is similar to the standard recovery techniques; however, recovery in PostgreSQL is simplified in some ways because of the MVCC protocol.

Under PostgreSQL, recovery does not have to undo the effects of aborted transactions: an aborting transaction makes an entry in the *pg_xact* file, recording the fact that it is aborting. Consequently, all versions of rows it leaves behind will never be visible to any other transactions. The only case where this approach could potentially lead to problems is when a transaction aborts because of a crash of the corresponding PostgreSQL process and the PostgreSQL process does not have a chance to create the *pg_xact* entry before the crash. PostgreSQL handles this as follows: Before checking the status of a transaction in the *pg_xact* file, PostgreSQL checks whether the transaction is running on any of the PostgreSQL processes. If no PostgreSQL process is currently running the transaction, but the *pg_xact* file shows the transaction as still running, it is safe to assume that the transaction crashed and the transaction's *pg_xact* entry is updated to "aborted".

Additionally, recovery is simplified by the fact that PostgreSQL MVCC already keeps track of some of the information required by write-ahead logging. More precisely, there is no need for logging the start, commit, and abort of transactions, since MVCC logs the status of every transaction in the *pg_xact*.

PostgreSQL provides support for two-phase commit; two-phase commit is described in more detail in Section 23.2.1. The **prepare transaction** command brings a transaction to the prepared-state of two-phase commit by persisting its state on disk. When the coordinator decides on whether the transaction should be committed or aborted, PostgreSQL can do so, even if there is a system crash between reaching the prepared state and the commit/abort decision. Leaving transactions in the prepared state for a long time is not recommended since locks continue to be held by the prepared transaction, affecting concurrency and interfering with the ability of VACUUM to reclaim storage.

32.5.3 High Availability and Replication

PostgreSQL provides support for high availability (HA). To achieve HA, all updates performed on the primary database system must be replicated to a secondary (backup/standby) database system, which is referred to as a replica. In case the primary database fails, the secondary can take over transaction processing.

Updates performed at a primary database can be replicated to more than one secondary database. Reads can then be performed at the replicas, as long as the reader does not mind reading a potentially slightly outdated state of the database.

Replication in a HA cluster is done by log shipping. The primary severs (servers that can modify data) operate in continuous archiving mode while the secondary servers operate in recovery mode, continuously reading the WAL records from the primary. By default, the log is created in units of segments, which are files that are (by default) 16MB in size. PostgreSQL implements file-based log shipping by transferring WAL records one file (WAL segment) at a time. Log shipping comes with low performance overhead on master servers; however, there is a potential windows for data loss. Log shipping is asynchronous, which means that WAL records are shipped after transaction commit. Thus, if the primary server crashes then the transactions not yet shipped will be lost. PostgreSQL has a configuration parameter (archive_timeout) that can help to limit the potential window of data lost in file-based log shipping by forcing the primary server to switch to new WAL segment file periodically.

Streaming replication solution allows for secondary servers to be more up-to-date than the file-based log shipping approach, thereby decreasing the potential window of data loss. In this case, the primary servers stream WAL records to the secondary servers as the WAL records are generated. without having to wait for the WAL file to be filled. Streaming replication is asynchronous by default and thus, it is still possible to experience a delay between the time a transaction is committed on the primary server and the moment is becomes visible in the secondary server. Nevertheless, this window

of potential data loss is shorter (typically below a second) than the one in file-based log shipping.

PostgreSQL can also operate using synchronous replication. In this case, each commit of a write transaction waits until confirmation is received that the commit has been written to the WAL on disk of both the primary and secondary server. Even though this approach increases the confidence that the data of a transaction commit will be available, commit processing is slower. Further, data loss is still possible if both primary and secondary servers crash at the same time. For read-only transactions and transaction rollbacks, there is no need to wait for the response from the secondary servers.

In addition to physical replication, PostgreSQL also supports **logical replication**. Logical replication allows for fine-grained control over data replication by replicating logical data modifications from the WAL based on a replication identity (usually a primary key). Physical replication, on the other hand, is based on exact block addresses and byte-by-byte replication. The logical replication can be enabled by setting the *wal_level* configuration parameter to *logical*.

Logical replication is implemented using a publish and subscribe model in which one or more subscribers subscribing to one or more publications (changes generated from a table or a group of tables). The server responsible for sending the changes is called a publisher while the server that subscribes to the changes is called a subscriber. When logical replication is enabled, the subscriber receives a snapshot of the data on the publisher database. Then, each change that happens on the publisher is identified and sent to the subscriber using streaming replication. The subscriber is responsible for applying the change in the same order as the publisher, to guarantee consistency. Typical use-cases for logical replication include replicating data between different platforms or different major versions of PostgreSQL, sharing a subset of the database between different groups of users, sending incremental changes in a single database, consolidating multiple databases into a single one, among other use-cases.

32.6 SQL Variations and Extensions

The current version of PostgreSQL supports almost all entry-level SQL-92 features, as well as many of the intermediate- and full-level features. It also supports many SQL:1999 and SQL:2003 features, including most object-relational features and the SQL/XML features for parsed XML. In fact, some features of the current SQL standard (such as arrays, functions, and inheritance) were pioneered by PostgreSQL.

32.6.1 PostgreSQL Types

PostgreSQL has support for several nonstandard types, useful for specific application domains. Furthermore, users can define new types with the **create type** command. This includes new low-level base types, typically written in C (see Section 32.6.2.1).

32.6.1.1 The PostgreSQL Type System

PostgreSQL types fall into the following categories:

- **Base types**: Base types are also known as **abstract data types**; that is, modules that encapsulate both state and a set of operations. These are implemented below the SQL level, typically in a language such as C (see Section 32.6.2.1). Examples are **int4** (already included in PostgreSQL) or **complex** (included as an optional extension type). A base type may represent either an individual scalar value or a variable-length array of values. For each scalar type that exists in a database, PostgreSQL automatically creates an array type that holds values of the same scalar type.
- **Composite types**: These correspond to table rows; that is, they are a list of field names and their respective types. A composite type is created implicitly whenever a table is created, but users may also construct them explicitly.
- **Domains**: A domain type is defined by coupling a base type with a constraint that values of the type must satisfy. Values of the domain type and the associated base type may be used interchangeably, provided that the constraint is satisfied. A domain may also have an optional default value, whose meaning is similar to the default value of a table column.
- **Enumerated types**: These are similar to enum types used in programming languages such as C and Java. An enumerated type is essentially a fixed list of named values. In PostgreSQL, enumerated types may be converted to the textual representation of their name, but this conversion must be specified explicitly in some cases to ensure type safety. For instance, values of different enumerated types may not be compared without explicit conversion to compatible types.
- **Pseudotypes**: Currently, PostgreSQL supports the following pseudotypes: *any*, *anyarray*, *anyelement*, *anyenum*, *anynonarray cstring*, *internal*, *opaque*, *language_handler*, *record*, *trigger*, and *void*. These cannot be used in composite types (and thus cannot be used for table columns), but can be used as argument and return types of user-defined functions.
- **Polymorphic types.** Four of the pseudotypes *anyelement*, *anyarray*, *anynonarray*, and *anyenum* are collectively known as **polymorphic**. Functions with arguments of these types (correspondingly called **polymorphic functions**) may operate on any actual type. PostgreSQL has a simple type-resolution scheme that requires that: (1) in any particular invocation of a polymorphic function, all occurrences of a polymorphic type must be bound to the same actual type (that is, a function defined as *f*(*anyelement*, *anyelement*) may operate only on pairs of the same actual type), and (2) if the return type is polymorphic, then at least one of the arguments must be of the same polymorphic type.

32.6.1.2 Nonstandard Types

The types described in this section are included in the standard distribution. Furthermore, thanks to the open nature of PostgreSQL, there are several contributed extension types, such as complex numbers, and ISBN/ISSNs (see Section 32.6.2).

Geometric data types (*point*, *line*, *lseg*, *box*, *polygon*, *path*, *circle*) are used in geographic information systems to represent two-dimensional spatial objects such as points, line segments, polygons, paths, and circles. Numerous functions and operators are available in PostgreSQL to perform various geometric operations such as scaling, translation, rotation, and determining intersections.

Full-text searching is performed in PostgreSQL using the *tsvector* type that represents a document and the *tsquery* type that represents a full-text query. A *tsvector* stores the distinct words in a document, after converting variants of each word to a common normal form (for example, removing word stems). PostgreSQL provides functions to convert raw text to a *tsvector* and concatenate documents. A *tsquery* specifies words to search for in candidate documents, with multiple words connected by Boolean operators. For example, the query `'index & !(tree | hash)'` finds documents that contain "index" without using the words "tree" or "hash." PostgreSQL natively supports operations on full-text types, including language features and indexed search.

PostgreSQL offers data types to store network addresses. These data types allow network-management applications to use a PostgreSQL database as their data store. For those familiar with computer networking, we provide a brief summary of this feature here. Separate types exist for IPv4, IPv6, and Media Access Control (MAC) addresses (*cidr*, *inet* and *macaddr*, respectively). Both *inet* and *cidr* types can store IPv4 and IPv6 addresses, with optional subnet masks. Their main difference is in input/output formatting, as well as the restriction that classless Internet domain routing (CIDR) addresses do not accept values with nonzero bits to the right of the netmask. The *macaddr* type is used to store MAC addresses (typically, Ethernet card hardware addresses). PostgreSQL supports indexing and sorting on these types, as well as a set of operations (including subnet testing, and mapping MAC addresses to hardware manufacturer names). Furthermore, these types offer input-error checking. Thus, they are preferable over plain text fields.

The PostgreSQL *bit* type can store both fixed- and variable-length strings of 1s and 0s. PostgreSQL supports bit-logical operators and string-manipulation functions for these values.

PostgreSQL offers data types to store XML and JSON data. Both XML and JSON data can be stored as text. However, the specialized data types offer additional functionality. For example, the XML data type allows for checking the input values for well-formedness while there are support functions to perform type-safe operations on it. Similarly, the JSON type has the advantage of enforcing that each stored value is valid according to the JSON rules. There are two JSON data types: *json* and *jsonb* and both accept almost identical sets of values as input. However, the *json* data type stores an exact copy of the input text, which processing functions must re-parse on each execu-

tion; while *jsonb* data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process while supporting indexing capabilities.

32.6.2 Extensibility

Like most relational database systems, PostgreSQL stores information about databases, tables, columns, and so forth, in what are commonly known as **system catalogs**, which appear to the user as normal tables. Other relational database systems are typically extended by changing hard-coded procedures in the source code or by loading special extension modules written by the vendor.

Unlike most relational database systems, PostgreSQL goes one step further and stores much more information in its catalogs: not only information about tables and columns, but also information about data types, functions, access methods, and so on. Therefore, PostgreSQL makes it easy for users to extend and facilitates rapid prototyping of new applications and storage structures. PostgreSQL can also incorporate user-written code into the server, through dynamic loading of shared objects. This provides an alternative approach to writing extensions that can be used when catalog-based extensions are not sufficient.

Furthermore, the contrib module of the PostgreSQL distribution includes numerous user functions (for example, array iterators, fuzzy string matching, cryptographic functions), base types (for example, encrypted passwords, ISBN/ISSNs, *n*-dimensional cubes) and index extensions (for example, RD-trees,[5] indexing for hierarchical labels). Thanks to the open nature of PostgreSQL, there is a large community of PostgreSQL professionals and enthusiasts who also actively extend PostgreSQL. Extension types are identical in functionality to the built-in types; the latter are simply already linked into the server and preregistered in the system catalog. Similarly, this is the only difference between built-in and extension functions.

32.6.2.1 Types

PostgreSQL allows users to define composite types, enumeration types, and even new base types. A composite-type definition is similar to a table definition (in fact, the latter implicitly does the former). Stand-alone composite types are typically useful for function arguments. For example, the definition:

create type *city_t* **as** (*name* **varchar**(80), *state* **char**(2));

allows functions to accept and return *city_t* tuples, even if there is no table that explicitly contains rows of this type.

[5] RD-trees are designed to index sets of items, and support set containment queries such as finding all sets that contain a given query set.

Adding base types to PostgreSQL is straightforward; an example can be found in complex.sql and complex.c in the tutorials of the PostgreSQL distribution. The base type can be declared in C, for example:

```
typedef struct Complex {
    double x;
    double y;
} Complex;
```

The next step is to define functions to read and write values of the new type in text format. Subsequently, the new type can be registered using the statement:

```
create type complex (
    internallength = 16,
    input = complex_in,
    output = complex_out,
    alignment = double
);
```

assuming the text I/O functions have been registered as *complex_in* and *complex_out*. The user has the option of defining binary I/O functions as well (for more efficient data dumping). Extension types can be used like the existing base types of PostgreSQL. In fact, their only difference is that the extension types are dynamically loaded and linked into the server. Furthermore, indices may be extended easily to handle new base types; see Section 32.6.2.3.

32.6.2.2 Functions

PostgreSQL allows users to define functions that are stored and executed on the server. PostgreSQL also supports function overloading (that is, functions may be declared by using the same name but with arguments of different types). Functions can be written as plain SQL statements, or in several procedural languages (covered in Section 32.6.2.4). Finally, PostgreSQL has an application programmer interface for adding functions written in C (explained in this section).

User-defined functions can be written in C (or a language with compatible calling conventions, such as C++). The actual coding conventions are essentially the same for dynamically loaded, user-defined functions, as well as for internal functions (which are statically linked into the server). Hence, the standard internal function library is a rich source of coding examples for user-defined C functions. Once the shared library containing the function has been created, a declaration such as the following registers it on the server:

```
create function complex_out(complex)
    returns cstring
    as 'shared_object_filename'
    language C immutable strict;
```

The entry point to the shared object file is assumed to be the same as the SQL function name (here, *complex_out*), unless otherwise specified.

The example here continues the one from Section 32.6.2.1. The application program interface hides most of PostgreSQL's internal details. Hence, the actual C code for the above text output function of *complex* values is quite simple:

```
PG_FUNCTION_INFO_V1(complex_out);
Datum complex_out(PG_FUNCTION_ARGS) {
    Complex *complex = (Complex *) PG_GETARG_POINTER(0);
    char *result;
    result = (char *) palloc(100);
    snprintf(result, 100, "(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

The first line declares the function complex_out, and the following lines implement the output function. The code uses several PostgreSQL-specific constructs, such as the palloc() function, which dynamically allocates memory controlled by PostgreSQL's memory manager.

Aggregate functions in PostgreSQL operate by updating a **state value** via a **state transition** function that is called for each tuple value in the aggregation group. For example, the state for the **avg** operator consists of the running sum and the count of values. As each tuple arrives, the transition function simply add its value to the running sum and increment the count by one. Optionally, a *final* function may be called to compute the return value based on the state information. For example, the final function for **avg** would simply divide the running sum by the count and return it.

Thus, defining a new aggregate function (referred to a **user-defined aggregate function**) is a simple as defining these functions. For the *complex* type example, if *complex_add* is a user-defined function that takes two complex arguments and returns their sum, then the **sum** aggregate operator can be extended to complex numbers using the simple declaration:

```
create aggregate sum (complex) (
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);
```

Note the use of function overloading: PostgreSQL will call the appropriate *sum* aggregate function, on the basis of the actual type of its argument during invocation. The *stype* is the state value type. In this case, a final function is unnecessary, since the return value is the state value itself (that is, the running sum in both cases).

User-defined functions can also be invoked by using operator syntax. Beyond simple "syntactic sugar" for function invocation, operator declarations can also provide hints to the query optimizer in order to improve performance. These hints may include information about commutativity, restriction and join selectivity estimation, and various other properties related to join algorithms.

32.6.2.3 Index Extensions

The indices supported by PostgreSQL can be extended to accommodate new base types. Adding index extensions for a type requires the definition of an **operator class**, which encapsulates the following:

- **Index-method strategies**: These are a set of operators that can be used as qualifiers in **where** clauses. The particular set depends on the index type. For example, B-tree indices can retrieve ranges of objects, so the set consists of five operators (<, <=, =, >=, and >), all of which can appear in a **where** clause involving a B-tree index while a hash index allows only equality testing.
- **Index-method support routines**: The above set of operators is typically not sufficient for the operation of the index. For example, a hash index requires a function to compute the hash value for each object.

For example, if the following functions and operators are defined to compare the magnitude of *complex* numbers (see Section 32.6.2.1), then we can make such objects indexable by the following declaration:

```
create operator class complex_abs_ops
    default for type complex using btree as
            operator 1 < (complex, complex),
            operator 2 <= (complex, complex),
            operator 3 = (complex, complex),
            operator 4 >= (complex, complex),
            operator 5 > (complex, complex),
            function 1 complex_abs_cmp(complex, complex);
```

The **operator** statements define the strategy methods and the **function** statements define the support methods.

32.6.2.4 Procedural Languages

Stored functions and procedures can be written in a number of procedural languages. Furthermore, PostgreSQL defines an application programmer interface for hooking up any programming language for this purpose. Programming languages can be registered on demand and are either **trusted** or **untrusted**. The latter allow unlimited access to the DBMS and the file system, and writing stored functions in them requires superuser privileges.

- **PL/pgSQL**. This is a trusted language that adds procedural programming capabilities (for example, variables and control flow) to SQL. It is very similar to Oracle's PL/SQL. Although code cannot be transferred verbatim from one to the other, porting is usually simple.
- **PL/Tcl**, **PL/Perl**, and **PL/Python**. These leverage the power of Tcl, Perl, and Python to write stored functions and procedures on the server. The first two come in both trusted and untrusted versions (PL/Tcl, PL/Perl and PL/TclU, PL/PerlU, respectively), while PL/Python is untrusted at the time of this writing. Each of these has bindings that allow access to the database system via a language-specific interface.

32.7 Foreign Data Wrappers

Foreign data wrappers (FDW) allow a user to connect with external data sources to transparently query data that reside outside of PostgreSQL, as if the data were part of an existing table in a database. PostgreSQL implements FDWs to provide SQL/MED ("Management of External Data") functionality. SQL/MED is an extension of the ANSI SQL standard specification that defines types that allow a database management system to access external data. FDWs can be a powerful tool both for data migration and data analysis scenarios.

Today, there are a number of FDWs that enable PostgreSQL to access different remote stores, such as other relational databases supporting SQL, key-value (NoSQL) sources, and flat files; however, most of them are implemented as PostgreSQL extensions and are not officially supported. PostgreSQL provides two FDWs modules:

- file_fdw: The *file_fdw* module allow users to create foreign tables for data files in the server's file system, or to specify commands to be executed on the server and read their output. Access is read-only and the data file or command output should be in a format compatible to the **copy from** command. These include csv files, text files with one row per line, with columns separated by user-specified delimiter character, and a PostgreSQL specific binary format.
- postgres_fdw: The *postgres_fdw* module is used to access remote tables stored in external PostgreSQL servers. Using *postgres_fdw*, foreign tables are updatable as long as the the required privileges are set. When a query references a remote table,

postgres_fdw opens a transaction on the remote server that is committed or aborted when the local transaction commits or aborts. The remote transaction uses *serializable* isolation level when the local transaction has *serializable* isolation level; otherwise it uses *repeatable read* isolation level. This ensures that a query performing multiple table scans on the remote server it will get snapshot-consistent results for all the scans.

Instead of fetching all the required data from the remote database and compute the query locally, *postgres_fdw* tries to reduce the amount of data transferred from foreign servers. Queries are optimized to send the query **where** clauses that use data types, operators, and built-in functions to the remote server for execution and by retrieving only the table columns that are needed for the correct query execution. Similarly, when a join operation is performed between foreign tables on the same foreign server, *postgres_fdw* pushes down the join operation to the remote server and retrieves only the results, unless the optimizer estimates that it will be more efficient to fetch rows from each table individually.

In a typical usage scenario, the user should go through a number of prerequisites steps before accessing foreign data using the FDWs provided by PostgreSQL. Specifically, the user should a) install the desired FDW extension, b) create a foreign server object to specify connection information for the particular external data source, c) specify the credentials a user should use to access an external data resource, and d) create one or more foreign tables specifying the schema of the external data source to be accessed.

A FDW handles all the operations performed on a foreign table and is responsible for accessing the remote data source and returning it to the PostgreSQL executor. PostgreSQL internally provides a list of APIs that a FDW can implement, depending on the desired functionality. For example, if a FDW intends to support remote foreign joins instead of fetching the tables and performing the join locally, it should implement the `GetForeignJoinPaths()` callback function.

32.8 PostgreSQL Internals for Developers

This section is targeted for developers and researchers who plan to extend the PostgreSQL source code to implement any desired functionality. The section provides pointers on how to install PostgreSQL from source code, navigate the source code, and understand some basic PostgreSQL data structures and concepts, as a first step toward adding new functionality to PostgreSQL. The section pays particular focus on the region-based memory manager of PostgreSQL and the structure of nodes in a query plan, and the key functions that are invoked during the processing of a query. It also explains the organization of tuples and their internal representation in the form of `Datum` data structures used to represent values, and various data structures used to represent tuples. Finally, the section describes error handling mechanisms in PostgreSQL, and

offer advice on steps required when adding a new functionality. This section can also serve as a reference source for key concepts whose understanding is necessary when changing the source code of PostgreSQL. For more development information we encourage the readers to refer to the PostgreSQL development wiki.[6]

32.8.1 Installation From Source Code

We start with a brief walk-through of the steps required to install PostgreSQL from source code, which is a first step for the development of any new functionality. The source code can be obtained from the version control repository at: git.postgresql.org, or downloaded from: https://www.postgresql.org/download/. We describe the basic steps required for installation in this section; detailed installation instructions can be found at: https://www.postgresql.org/docs/current/installation.html.

32.8.1.1 Requirements

The following software packages are required for a successful build:

- An ISO/ANSI C compiler is required to compile the source code. Recent versions of GCC are recommended, but PostgreSQL can be built using a wide variety of compilers from different vendors.
- tar is required to unpack the source distribution, in addition to either gzip or bzip2.
- The GNU Readline library is used by default. An alternative is to disable it by specifying —without-readline option when invoking configure (discussed next). When installing PostgreSQL under a Linux distribution both readline and readline-devel packages are needed.
- The zlib compression library is used by default. This library is used by pg_dump and pg_restore.
- GNU Flex (2.5.31 or later) and Bison(1.875 or later) are needed to build from a Git checkout (which is necessary when doing any development on the server).
- Perl (5.8.3 or later) is required when doing a build from the Git checkout.

32.8.1.2 Installation Steps

Once all required software packages are installed, we can proceed with the PostgreSQL installation. The installation consists of a couple of steps discussed next:

[6] https://wiki.postgresql.org/wiki/Development_information

1. **Configuration**: The first step of the installation procedure is to configure the source tree for a given system, and specify installation options (e.g. the directory of the build). A default configuration can be invoked with:

```
./configure
```

The configure script sets up files for building the server, utilities and all clients applications, by default under /usr/local/pgsql; to specify an alternative directory you should run configure with the command line option —prefix=PATH, where PATH is the directory where you wish to install PostgreSQL.[7]).
In addition to the –prefix option, other frequently used options include –enable-debug, –enable-depend, and –enable-cassert, which enable debugging; it is important to use these options to help you debug code that you create in PostgreSQL. The –enable-debug option enables build with debugging symbols (-g), the –enable-depend option turns on automatic dependency tracking, while the –enable-cassert option enables assertion checks (used for debugging).
Further, it is recommended that you set the environment variable CFLAGS to the value -O0 (the letter "O" followed by a zero) to turn off compiler optimization entirely. This option reduces compilation time and improves debugging information. Thus, the following commands can be used to configure PostgreSQL to support debugging:

```
export CFLAGS=-O0
./configure –prefix=PATH –enable-debug –enable-depend –enable-cassert
```

where PATH is the path for installing the files. The CFLAGS variable can alternatively be set in the command line by adding the option CFLAGS='-O0' to the configure command above.

2. **Build**: To start the build, type either of the following commands:

```
make
make all
```

This will build the core of PostgreSQL. For a complete build that includes the documentation as well as all additional modules (the contrib directory) type:

```
make world
```

3. **Regression Tests**: This is an optional step to verify whether PostgreSQL runs on the current machine in the expected way. To run the regression tests type:

```
make check
```

[7] More details about the command line options of configure can be found at: https://www.postgresql.org/docs/current/install-procedure.html.

4. **Installation**: To install PostgreSQL enter:

```
make install
```

This step will install files into the default directory or the directory specified with the –prefix command line option provided in Step 1.
For a full build (including the documentation and the contribution modules) type:

```
make install-world
```

32.8.1.3 Creating a Database

Once PostgreSQL is installed, we will create our first database, using the following steps; for a more detailed list of steps we encourage the reader to refer to: https://www.postgresql.org/docs/current/creating-cluster.html.

1. Create a directory to hold the PostgreSQL data tree by executing the following commands in the bash console:

```
mkdir DATA_PATH
```

where DATA_PATH is a directory on disk where PostgreSQL will hold data.

2. Create a PostgreSQL cluster by executing:

```
PATH/bin/initdb -D DATA_PATH
```

where PATH is the installation directory (specified in the ./configure call), and DATA_PATH is the data directory path.
A database cluster is a collection of databases that are managed by a single server instance. The initdb function creates the directories in which the database data will be stored, generates the shared catalog tables (which are the tables that belong to the whole cluster rather than to any particular database), and creates template1 (a template for generating new databases) and postgres databases. The postgres database is a default database available for use by all users, and any third party applications.

3. Start up the PostgreSQL server by executing:

```
PATH/bin/postgres -D DATA_PATH >logfile 2>&1 &
```

where PATH and DATA_PATH are as described earlier.
By default, PostgreSQL uses the TCP/IP port 5432 for the postgres server to listen for connections. Since default installations frequently exist side by side with the installations from source code, not all PostgreSQL servers can be simultaneously listening on the same TCP port. The postgres server can also run on a

different port, specified by using the flag -p. This port should then be specified by all client applications (e.g. createdb, psql discussed next).
To run postgres on a different port type:

```
PATH/bin/postgres -D DATA_PATH -p PORT >logfile 2>&1 &
```

where PORT is an alternative port number between 1024 and 65535, that is not currently used by any application on your computer.
The postgres command can also be called in *single-user mode*. This mode is particularly useful for debugging or disaster recovery. When invoked in single-user mode from the shell, the user can enter queries and the results will be printed to the screen, but in a form that is more useful for developers than end users. In the single-user mode, the session user will be set to the user with ID 1, and implicit superuser powers are granted to this user. This user does not actually have to exist, so the single-user mode can be used to manually recover from certain kinds of accidental damage to the system catalogs. To run the postgres server in the single-user mode type:

```
PATH/bin/postgres –single -D DATA_PATH DBNAME
```

4. Create a new PostgreSQL database in the cluster by executing:

```
PATH/bin/createdb -p PORT test
```

where PORT is the port on which the server is running; the port specification can be omitted if the default port (5432) is being used.
After this step, in addition to template1 and postgres databases, the database named test will be placed in the cluster as well. You can use any other name in place of test.

5. Log-in to the database using the psql command:

```
PATH/bin/psql -p PORT test
```

Now you can create tables, insert some data and run queries over this database. When debugging, it is frequently useful to run SQL commands directly from the command line or read them from a file. This can be achieved by specifying the options -c or -f. To execute a specific command you can use:

```
PATH/bin/psql -p PORT -c COMMAND test
```

where COMMAND is the command you wish to run, which is typically enclosed in double quotes.
To read and execute SQL statements from a file you can use:

```
PATH/bin/psql -p PORT -f FILENAME test
```

Directory	Description
config	Configuration system for driving the build
contrib	Source code for contribution modules (extensions)
doc	Documentation
src/backend	PostgreSQL Server (backend)
src/bin	psql, pg_dump, initdb, pg_upgrade and other front-end utilities
src/common	Code common to the front- and backends
src/fe_utils	Code useful for multiple front-end utilities
src/include	Header files for PostgreSQL
src/include/catalog	Definition of the PostgreSQL catalog tables
src/interfaces	Interfaces to PostgreSQL including libpq, ecpg
src/pl	Core Procedural Languages (plpgsql, plperl, plpython, tcl)
src/port	Platform-specific hacks
src/test	Regression tests
src/timezone	Timezone code from IANA
src/tools	Developer tools (including pgindent[8])
src/tutorial	SQL tutorial scripts

Figure 32.7 Top-level source code organization

where FILENAME is the name of the file containing SQL commands. If a file has multiple statements they need to be separated by semicolon.
When either -c or -f is specified, psql does not read commands from standard input; instead it terminates after processing all the -c and -f options in sequence.

32.8.2 Code Organization

Prior to adding any new functionality it is necessary to get familiar with the source code organization. Figure 32.7 presents the organization of PostgreSQL top level directory.

The PostgreSQL contrib tree contains porting tools, analysis utilities, and plug-in features that are not part of the core PostgreSQL system. Client (front-end) programs are placed in src/bin. The directory src/pl contains support for procedural languages (e.g. perl and python), which allows for writing PostgreSQL functions and procedures in these languages. Some of these libraries are not part of the generic build and need to be explicitly enabled (e.g. use ./configure –with-perl for perl support). The src/test directory contains a variety of regression tests, e.g. for testing authentication, concurrent behavior, locality and encodings, and recovery and replication.

Directory	Description
access	Methods for accessing different types of data (e.g., heap, hash, btree, gist/gin)
bootstrap	Routines for running PostgreSQL in a âŁžbootstrapâŁž mode (by initdb)
catalog	Routines used for modifying objects in the PostgreSQL Catalog
commands	User-level DDL/SQL commands (e.g., create, alter, vacuum, analyze, copy)
executor	Executor runs queries after they have been planned and optimized
foreign	Handles foreign data wrappers, user mappings, etc
jit	Provides independent Just-In-Time Compilation infrastructure
lib	Contains general purpose data structures used in the backend (e.g., binary heap, bloom filters, etc)
libpq	Code for the wire protocol (e.g., authentication, and encryption)
main	The main() routine determines how the PostgreSQL backend process will start and starts the right subsystem
nodes	Generalized Node structures in PostgreSQL. Contains functions to manipulate with nodes (e.g., copy, compare, print, etc)
optimizer	Optimizer implements the costing system and generates a plan for the executor
parser	Parser parses the sent queries
partitioning	Common code for declarative partitioning in PostgreSQL
po	Translations of backend messages to other languages
port	Backend-specific platform-specific hacks
postmaster	The main PostgreSQL process that always runs, answers requests, and hands off connections
regex	Henry Spencer's regex library
replication	Backend components to support replication, shipping of WAL logs, and reading them
rewrite	Query rewrite engine used with RULEs
snowball	Snowball stemming used with full-text search
statistics	Extended statistics system (CREATE STATISTICS)
storage	Storage layer handles file I/O, deals with pages and buffers
tcop	Traffic Cop gets the actual queries, and runs them
tsearch	Full-Text Search engine
utils	Various backend utility components, caching system, memory manager, etc

Figure 32.8 Source code organization of PostgreSQL backend

Since typically new functionality is added in the PostgreSQL backend directory, we further dive into the organization of this directory, which is presented in Figure 32.8.

Parser: The parser of PostgreSQL consists of two major components - the lexer and grammar. The lexer determines how the input will be tokenized. The grammar defines the grammar of the SQL and other commands that are processed by PostgreSQL, and is used for parsing commands.

The corresponding files of interest in the /backend/parser directory are: i) scan.l, which is the lexer that handles tokenization, ii) gram.y, which is the definition of the grammar, iii) parse_*.c, which contains specialized routines for parsing, and iv) analyze.c, which contains routines to transform a raw parse tree into a query tree representation.

Optimizer: The optimizer takes the query structure returned by the parser as input and produces a plan to be used by the executor as output. The /path directory contains code for exploring possible ways to join the tables (using dynamic programming), while the /plan subdirectory contains code for generating the actual execution plan. The /prep directory contains code for handling preprocessing steps for special cases. The /geqo directory contains code for a planner that uses genetic optimization algorithm to handle queries with a large number of joins; the genetic optimizer performs a semi-random search through the join tree space. The primary entry point for the optimizer is the planner() function.

Executor: The executor processes a query plan, which is a tree of plan nodes. The plan tree nodes are operators that implement a demand/pull driven pipeline of tuple processing operations (following the Volcano model). When the next() function is called on a node, it produces the next tuple in its output sequence, or NULL if no more tuples are available. If the node is not a relation-scan or index-scan, it will have 1 or more children nodes. the code implementing the operation calls next() on its children to obtain input tuples.

32.8.3 System Catalogs

PostgreSQL is an entirely catalog driven DBMS. Not only are system catalogs used to store metadata information about tables, their columns, and indices, but they are also used to store metadata information about data types, function definitions, operators and access methods. Such an approach provides an elegant way of offering extensibility: new data types or access methods can be added by simply inserting a record in the corresponding pg_* table. System catalogs are thus used to a much greater extent in PostgreSQL, compared to other relational database systems.

The list of catalog tables with their descriptions can be found at: https://www.postgresql.org/docs/current/catalogs.html. For a graphical representation of the relationships between different catalog tables, we encourage the reader to refer to: https://www.postgrescompare.com/pg_catalog/constrained_pg_catalog_organic.pdf.

Some of the widely used system catalog tables are the following:

- **pg_class**: contains one row for each table in the database.
- **pg_attribute**: contains one row for each column of each table in the database.
- **pg_index**: contains one row for each index with a reference to the table it belongs to (described in the pg_class table).
- **pg_proc**: contains one row for each defined function. Each function is described through its name, input argument types, result type, implementation language, and definition (either the text, if it is in an interpreted language, or a reference to its executable code, if it is compiled). Compiled functions can be statically linked into the server, or stored in shared libraries that are dynamically loaded on the first use.
- **pg_operator**: contains operators that can be used in expressions.
- **pg_type**: contains information about basic data types that columns in the table can have and that are accepted as inputs and outputs to functions. New data types are added by making new entries in the pg_type table.

In addition to the system catalogs, PostgreSQL provides a number of built-in system views. System views typically provide convenient access to commonly used queries on the system catalogs. For instance:

select *tablename* **from** *pg_catalog.pg_tables*;

will print the list of table names of all tables in the database. System views can be recognized in the pg_catalog schema by the plural suffix (e.g., pg_tables, or pg_indexes).

32.8.4 The Region-Based Memory Manager

PostgreSQL uses a region-based memory manager, which implements of a hierarchy of different memory contexts. Objects are created in specific memory contexts, and a single function call frees up all objects that are in a particular memory context. The memory manager routines can be found in the /backend/utils/mmgr directory. An important difference between PostgreSQL and other applications written in C is that in PostgreSQL, memory is allocated via a special routine called palloc(), as opposed to the traditional malloc() routine of C. The allocations can be freed individually with pfree() function calls, as opposed to calling the free() routine; but the main advantage of memory contexts over plain use of malloc()/free() is that the entire content of a memory context can be efficiently freed using a single call, without having to request freeing of each individual chunk within it. This is both faster and less error prone compared to per-object bookkeeping, since it prevents memory leakage and reduces chances of use-after-free bugs.

All allocations occur inside a corresponding memory context. Examples of contexts are: CurrentMemoryContext (the current context), TopMemoryContext (the backend lifetime), CurTransactionContext (the transaction lifetime), PerQueryContext (the query lifetime), PerTupleContext (the per-result-tuple lifetime). When unspecified, the default memory context is CurrentMemoryContext, which is stored as a global variable. Contexts are arranged in a tree hierarchy, expressed through a parent-child relationship. Whenever a new context gets created, the context from which the creation context routine gets invoked becomes the parent of the newly created context.

The basic operations on a memory context are:

- **MemoryContextCreate()**: to create a new context.
- **MemoryContextSwitchTo()**: to switch from the CurrentMemoryContext into a new one.
- **MemoryContextAlloc()**: to allocate a chunk of memory within a context.
- **MemoryContextDelete()**: to delete a context, which includes freeing all the memory allocated within the context.
- **MemoryContextReset()**: to reset a context, which frees all memory allocated in the context, but not the context object itself.

The MemoryContextSwitchTo() operation selects a new current context and returns the previous context, so that the caller can restore the previous context before exiting. When a memory context is reset, all allocations within the context are automatically released. Reset or deletion of a context will automatically invoke the call to reset/delete all children contexts. As a consequence, memory leaks are rare in the backend, since all memory will eventually be released. Nonetheless, a memory leak could happen when memory is allocated in a too-long-lived context, i.e., a context that lives longer than supposed. An example of such a case would be allocating resources that are needed per tuple into CurTransactionContext that lives much longer than the tuple.

32.8.5 Node Structure and Node Functions

Each query in PostgreSQL is represented as a tree of nodes; the actual type of nodes differs depending on the query stage (e.g., the parsing, optimization, or execution stage). Since C does not support inheritance at the language level (unlike C++), to represent node type inheritance, PostgreSQL uses a node structuring convention that in effect implements a simple object system with support for a single inheritance. The root of the class hierarchy is Node, presented below:

```
typedef struct {
    NodeTag type;
} Node;
```

The first field of any Node is NodeTag, which is used to determine a Node's specific type at run-time. Each node consists of a type, plus appropriate data. It is particularly important to understand the node type system when adding new features, such as new access path, or new execution operator. Important functions related to nodes are: makeNode() for creating a new node, IsA() which is a macro for run-time type testing, equal() for testing the equality of two nodes, copyObject() for a deep copy of a node (which should make a copy of the tree rooted at that node), nodeToString() to serialize a node to text (which is useful for printing the node and tree structure), and stringToNode() for deserializing a node from text.

An important thing to remember when modifying or creating a new node type is to update these functions (especially equal() and copy() that can be found in equalfuncs.c and copyfuncs.c in the /CODE/nodes/ directory). For serialization and deserialization, /nodes/outfuncs.c need to be modified as well.

32.8.5.1 Steps For Adding a New Node Type

To illustrate, suppose that we want to introduce a new node type called TestNode, the following are the steps required:

1. Add a tag (T_TestNode) to the enum NodeTag in include/nodes/nodes.h.
2. Add the structure definition to the appropriate include/nodes/*.h file. The nodes.h is used for defining node tags (NodeTag), primnodes.h for primitive nodes, parsenodes.h for parse tree nodes, pathnodes.h for path tree nodes and planner internal structures, plannodes.h for plan tree nodes, execnodes.h for executor nodes, and memnodes.h for memory nodes.
   For example, new node type can be defined as follows:

```
typedef struct TestNode
{
    NodeTag type;
    /* a list of other attributes */
} TestNode;
```

3. If we intend to use copyObject(), equal(), nodeToString() or stringToNode(), we need to add an appropriate function to copyfuncs.c, equalfuncs.c, outfuncs.c, and readfuncs.c respectively. For example:

```
static TestNode *
_copyTestNode(const TestNode *from)
{
      TestNode *newnode = makeNode(TestNode);
      /* copy remainder of node fields (if any) */
      newnode= COPY_*(from);
      return newnode;
}
```

where COPY_* is a routine to copy individual fields. Alternatively, each attribute of the TestNode can be copied individually by calling the existing copy routines, such as COPY_NODE_FIELD, COPY_SCALAR_FIELD, and COPY_POINTER_FIELD.

4. We also need to modify the functions in nodeFuncs.c to add code for handling the new node type; which of the functions needs to be modified depends on the node type added. Examples of functions in this file includes *_tree_walker() functions to traverse various types of trees, and *_tree_mutator() functions to traverse various types of trees and return a copy with specified changes to the tree.

As a general note, there may be other places in the code where we might need to inform PostgreSQL about our new node type. The safest way to make sure no place in the code has been overlooked is to search (e.g., using grep) for references to one or two similar existing node types to find all the places where they appear in the code.

32.8.5.2 Casting Pointers to Subtypes and Supertypes

To support inheritance, PostgreSQL uses a simple convention where the first field of any subtype is its parent, i.e., its supertype. Hence, casting a subtype into a supertype is trivial. Since the first field of a node of any type is guaranteed to be the NodeTag, any node can be cast into Node *. Declaring a variable to be of Node * (instead of void *) can facilitate debugging.

In the following we show examples of casting a subtype into a supertype and vice versa, by using the example of SeqScanState and PlanState. PlanState is the common abstract superclass for all PlanState-type nodes, including ScanState node. ScanState extends PlanState for node types that represent scans of an underlying relation. Its subtypes include SeqScanState, IndexScanState, and IndexOnlyScanState among others. To cast SeqScanState into PlanState we can use direct casting such as (PlanState *) SeqScanState *.

Casting a supertype into a subtype on the other hand requires a run-time call to a castNode macro, which will check whether the NodeTag of the given pointer is of the given subtype. Similarly, a supertype can be cast directly into a subtype after invoking the IsA macro at run-time, which will check whether the given node is of the requested

subtype (again by checking the NodeTag value). The following code snippet shows an example of casting a supertype into a subtype by using the castNode macro:

```
static TupleTableSlot *
ExecSeqScan(PlanState *pstate)
{
    /* Cast a PlanState (supertype) into a SeqScanState (subtype) */
    SeqScanState *node = castNode(SeqScanState, pstate);
    ...
}
```

32.8.6 Datum

Datum is a generic data type used to store the internal representation of a single value of any SQL data type that can be stored in a PostgreSQL table. It is defined in postgres.h. A Datum contains either a value of a pass-by-value type or a pointer to a value of a pass-by-reference type. The code using the Datum has to know which type it is, since the Datum itself does not contain that information. Usually, C code will work with a value in a native representation, and then convert to or from a Datum in order to pass the value through data-type-independent interfaces.

There are a number of macros to cast a Datum to and from one of the specific data types. For instance:

- **Int32GetDatum(int)**: will return a Datum representation of an Int32.
- **DatumGetInt32(Datum)**: will return Int32 from a Datum.

Similar macros exist for all other data types such as Bool (boolean), and Char (character) data types.

32.8.7 Tuple

Datums are used to extensively to represent values in tuples. A tuple comprises of a sequence of Datums. HeapTupleData (defined in include/access/htup.h) is an in-memory data structure that points to a tuple. It contains the length of a tuple, and a pointer to the tuple header. The structure definition is as follows:

```
typedef struct HeapTupleData
{
    uint32 t_len; /* length of *t_data */
    ItemPointerData t_self; /* SelfItemPointer */
    Oid t_tableOid; /* table the tuple came from */
    HeapTupleHeader t_data; /* pointer to tuple header and data */
} HeapTupleData;
```

The t_len field contains the tuple length; the value of this field should always be valid, except in the pointer-to-nothing case. The t_self pointer is a pointer to an item within a disk page of a known file. It consists of a block ID (which is a unique identifier of a block), and an offset within the block. The t_self and t_tableOid (the ID of the table the tuple belongs to) values should be valid if the HeapTupleData points to a disk buffer, or if it represents a copy of a tuple on disk. They should be explicitly set invalid in tuples that do not correspond to tables in the database.

There are several ways in which a pointer t_data can point to a tuple:

- **Pointer to a tuple stored in a disk buffer**: which is a pointer directly to a pinned buffer page (when the page is stored in the memory buffer pool).
- **Pointer to nothing**: which points to NULL, and is frequently used as a failure indicator in functions.
- **Part of a palloc'd tuple**: the HeapTupleData struct itself and the tuple form a single palloc'd chunk. t_data points to the memory location immediately following the HeapTupleData struct.
- **Separately allocated tuple**: t_data points to a palloc'd chunk that is not adjacent to the HeapTupleData struct.
- **Separately allocated minimal tuple**: t_data points minimal tuple offset bytes before the start of a MinimalTuple.

MinimalTuple (defined in htup_details.h) is an alternative representation to HeapTuple that is used for transient tuples inside the executor, in places where transaction status information is not required, and the tuple length is known. The purpose of MinimalTuple is to save a few bytes per each tuple, which may be a worthwhile effort over a number of tuples. This representation is chosen so that tuple access routines can work with either full or minimal tuples via a HeapTupleData pointer structure introduced above. The access routines see no difference, except that they must not access fields that are not part of the MinimalTuple (such as the tuple length for instance).

PostgreSQL developers recommend that tuples (both MinimalTuple and HeapTuple) should be accessed via TupleTableSlot routines. TupleTableSlot is an abstraction used in the executor to hide details to where tuple pointers point to (e.g., buffer page, heap allocated memory, etc). The executor stores tuples in a "tuple table", which is a list of independent TupleTableSlot(s), which enables cursor-like behavior. The TupleTableSlot routines will prevent access to the system columns and thereby prevent accidental use of the nonexistent fields. Examples of TupleTableSlot routines include the following:

```
PostgresMain()
    exec_simple_query()
        pg_parse_query()
            raw_parser() – calling the parser
        pg_analyze_rewrite()
            parse_analyze() – calling the parser (analyzer)
            pg_rewrite_query()
                QueryRewrite() – calling the rewriter
                    RewriteQuery()
        pg_plan_queries()
            pg_plan_query()
                planner() – calling the optimizer
                    create_plan()
        PortalRun()
            PortalRunSelect()
                ExecutorRun()
                    ExecutePlan() – calling the executor
                        ExecProcNode()
                                – uses the demand-driven pipeline execution model
            or
            ProcessUtility() – calling utilities
```

Figure 32.9 PostgreSQL Query Execution Stack

```
void (*copyslot) (TupleTableSlot *dstslot, TupleTableSlot *srcslot);
HeapTuple (*get_heap_tuple)(TupleTableSlot *slot);
MinimalTuple (*get_minimal_tuple)(TupleTableSlot *slot);
HeapTuple (*copy_heap_tuple)(TupleTableSlot *slot);
MinimalTuple (*copy_minimal_tuple)(TupleTableSlot *slot);
```

These function pointers are redefined for different types of tuples, such as HeapTuple, MinimalTuple, BufferHeapTuple, and VirtualTuple.

32.8.8 Query Execution Stack

Figure 32.9 depicts the query execution stack through important function calls. The execution starts with the PostgresMain() routine, which is the main module of the PostgreSQL backend and can be found in /tcop/postgres.c. Execution then proceeds through the parser, rewriter, optimizer, and executor.

Parser: When a query is received, the PostgresMain() routine calls exec_simple_query(), which in turn calls first pg_parse_query() to perform the parsing of the query. This function in turn calls the function raw_parser() (which is located in /parser/parser.c). The pg_parse_query() routine returns a list of raw parse

trees – each parse tree representing a different command, since a query may contain multiple select statements separated by semicolons.

Each parse tree is then individually analyzed and rewritten. This is achieved by calling pg_analyze_rewrite() from the exec_simple_query() routine. For a given raw parse tree, the pg_analyze_rewrite() routine performs parse analysis and applies rule rewriting (combining parsing and rewriting), returning a list of Query nodes as a result (since one query can be expanded into several ones as a result of this process). The first routine that pg_analyze_rewrite() invokes is parse_analyze() (located in /parser/analyze.c) to obtain a Query node of the given raw parse tree.

Rewriter: The rule rewrite system is triggered after parser. It takes the output of the parser, one Query tree, and defined rewrite rules, and creates zero or more Query trees as result. Typical examples of rewrite rules are replacing the use of a view with its definition, or populating procedural fields. The parse_analyze() call from the parser is thus followed by pg_rewrite_query() to perform rewriting. The pg_rewrite_query() invokes the QueryRewrite() routine (located in /rewrite/rewriteHandler.c), which is the primary module of the query rewriter. This method in turn makes a recursive call of RewriteQuery() where rewrite rules are repeatedly applied, as long as some rule is applicable.

Optimizer: After pg_analyze_rewrite() finishes, producing a list of Query nodes as output, the pg_plan_queries() routine is invoked to generate plans for all the nodes from the Query list. Each Query node is optimized by calling pg_plan_query(), which in turn invokes planner() (located in /plan/planner.c), which is the main entry point for the optimizer. The planner() routine invokes the create_plan() routine to create the best query plan for a given path, returning a Plan as output. Finally, the planner routine creates a PlannedStmt node to be fed to the executor.

Executor: Once the best plan is found for each Query node, the exec_simple_query() routine calls PortalRun(). A portal, previously created in the initialization step (discussed in the next section), represents the execution state of query. PortalRun() in turn invokes ExecutorRun() through PortalRunSelect() in the case of queries, or ProcessUtility() in the case of utility functions for each individual statement. Both ExecutorRun() and ProcessUtility() accept a PlannedStmt node; the only difference is that the utility call has the commandType attribute of the node set to CMD_UTILITY.

The ExecutorRun() defined in execMain.c, which is the main routine of the executor module, invokes ExecutePlan() which processes the query plan by calling ExecProcNode() for each individual node in the plan, applying the demand-driven pipelining (iterator) model (see Section 15.7.2.1 for more details).

32.8.8.1 Memory Management and Contexts Switches

Before making any changes to PostgreSQL code, it is important to understand how different contexts get switched during the lifetime of a query. Figure 32.10 shows a sketch of the query processing control flow, with key context switch points annotated with comments.

```
CreateQueryDesc()
ExecutorStart()
    CreateExecutorState() — creates per-query context
    switch to per-query context
    InitPlan()
        ExecInitNode() — recursively scans plan tree
            CreateExprContext() — creates per-tuple context
            ExecInitExpr()
ExecutorRun()
    ExecutePlan()
        ExecProcNode() — recursively called in per-query context
            ExecEvalExpr() — called in per-tuple context
            ResetExprContext() — to free per-tuple memory
ExecutorFinish()
    ExecPostprocessPlan()
    AfterTriggerEndQuery()
ExecutorEnd()
    ExecEndPlan()
        ExecEndNode() — recursively releases resources
    FreeExecutorState() — frees per-query context and child contexts
FreeQueryDesc()
```

Figure 32.10 PostgreSQL Query Processing Control Flow

Initialization: The CreateQueryDesc() routine (defined in tcop/pquery.c), is invoked through PortalStart(), i.e. in the initialization step before calling the ExecutorRun() routine. This function allocates the query descriptor, which is then used in all key executor routines (ExecutorStart, ExecutorRun, ExecutorFinish, and ExecutorEnd). The QueryDesc encapsulates everything that the executor needs to execute the query (e.g., a PlannedStmt of the chosen plan, source text of the query, the destination for tuple output, parameter values that are passed in, and a PlanState, among other information).

ExecutorStart() (defined in executor/ExecMain.c) is invoked through PortalStart(), when starting a new portal in which a query is going to be executed. This function must be called at the beginning of execution of any query plan. ExecutorStart() in turn calls CreateExecutorState() (defined in executor/utils.c) to create a per-query context. After creating the per-query context, InitPlan() (defined in executor/execMain.c) allocates necessary memory, and calls the ExecInitNode() routine (defined in executor/execProcnode.c); this function recursively initializes all the nodes in the plan tree. Query plan nodes may invoke CreateExprContext() (defined in execUtils.c) to create a per tuple context, and ExecInitExpr() to initialize it.

Execution: After the initialization is finalized, the execution starts by invoking ExecutorRun(), which calls ExecutePlan(), which in turn invokes ExecProcNode() for each

node in the plan tree. The context is switched from the per-query context into the per-tuple context for each invocation of the ExecEvalExpr() routine.

Upon the exit from this routine, ResetExprContext() is invoked. This is a macro that invokes the MemoryContextReset() routine to release all the space allocated within the per-tuple context.

Cleanup: The ExecutorFinish() routine must be called after ExecutorRun(), and before ExecutorEnd(). This routine performs cleanup actions such as calling ExecPostprocessPlan() to allow plan nodes to execute required actions before the shutdown, and AfterTriggerEndQuery() to invoke all AFTER IMMEDIATE trigger events.

The ExecutorEnd() routine must be called at the end of execution. This routine invokes ExecEndPlan() which in turn calls ExecEndNode() to recursively release all resources. FreeExecutorState() frees up the per-query context and consequently all of its child contexts (e.g., per-tuple contexts) if they have not been released already. Finally, FreeQueryDesc() from tcop/pquery.c frees the query descriptor created by CreateQueryDesc().

This fine level of control through different contexts coupled with palloc() and pfree() routines ensures that memory leaks rarely happen in the backend.

32.8.9 Error Handling

PostgreSQL has a rich error handling mechanism, with most prominent being ereport() and elog() macros. The ereport() macro is used for user-visible errors and allows for a detailed specification through a number of fields (e.g. SQLSTATE, detail, hint, etc.). The treatment of exceptions is conceptually similar to the treatment of exceptions in other languages such as C++, but is implemented using a set of macros since C does not have an exception mechanism defined as part of the language.

The elog(ERROR) function traces up the stack to the closest error handling block, which can then either handle the error or re-throw it. The top-level error handler (if reached) aborts the current transaction and resets the transaction's memory context. Reset in turn frees all resources held by the transaction, including files, locks, allocated memory and pinned buffer pages. Assertions (i.e., the Assert function) help detect programming errors, and are frequently used in PostgreSQL code.

32.8.10 Tips For Adding New Functionality

Adding a new feature for the first time in PostgreSQL can be an overwhelming experience even for researchers and experienced developers. This section provides some guidelines and general advice on how to minimize the risk of failures.

First, one needs to isolate in which subsystem the desired feature should be implemented, for example the backend, PostgreSQL contributions (contrib), etc. Once the subsystem is identified, a general strategy is to explore how similar parts of the system function. Developers are strongly advised against copying code directly since this is a common source of errors. Instead, the developers should focus on understanding of the exposed APIs, existing function calls, and the way they are used.

Another common source of errors is re-implementing existing functionality, often due to insufficient familiarity with the code base. A general guideline to avoid potential errors and code duplication is to use existing APIs and add only necessary additions to the existing code base. For instance, PostgreSQL has very good support of data structures and algorithms that should be favored over custom made implementations. Examples include:

- Simple linked list implementation: pg_list.h, list.c
- Integrated/inline doubly- and singly- linked lists: ilist.h, ilist.c
- Binary Heap implementation: binaryheap.c
- Hopcroft-Karp maximum cardinality algorithm for bipartite graphs: bipartite_match.c
- Bloom Filter: bloomfilter.c
- Dynamic Shared Memory-Based Hash Tables: dshash.c
- HyperLogLog cardinality estimator: hyperloglog.c
- Knapsack problem solver: knapsack.c
- Pairing Heap implementation: pairingheap.c
- Red-Black binary tree: rbtree.c
- String handling: stringinfo.c

Prior to adding a desired functionality, the behavior of the feature should be discussed in depth, with a special focus on corner cases. Corner cases are frequently overlooked and result in a substantial debugging overhead after the feature has been implemented. Another important aspect is understanding the relationship between the desired feature and other parts of PostgreSQL. Typical examples would include (but are not limited to) the changes to the system catalog, or the parser.

PostgreSQL has a great community where developers can ask questions, and questions are usually answered promptly. The web page https://www.postgresql.org/developer/ provides links to a variety of resources that are useful for PostgreSQL developers. The pgsql-general@postgresql.org mailing list is targeted for developers, and database administrators (DBAs) who have a question or problem when using PostgreSQL. The pgsql-hackers@postgresql.org mailing list is targeted for developers to submit and discuss patches, or for bug reports or issues with unreleased versions (e.g. development snapshots, beta or release candidates), and for discussion about database internals. Finally, the mailing list pgsql-novice@postgresql.org is a great starting point for all new developers, with a group of people who answer even basic questions.

Bibliographical Notes

Parts of this chapter are based on a previous version of the chapter, authored by Anastasia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou, Bianca Schroeder, Karl Schnaitter, and Gavin Sherry, which was published in the 6th edition of this textbook,

There is extensive online documentation of PostgreSQL at www.postgresql.org. This Web site is the authoritative source for information on new releases of PostgreSQL, which occur on a frequent basis. Until PostgreSQL version 8, the only way to run PostgreSQL under Microsoft Windows was by using Cygwin. Cygwin is a Linux-like environment that allows rebuilding of Linux applications from source to run under Windows. Details are at www.cygwin.com. Books on PostgreSQL include [Schonig (2018)], [Maymala (2015)] and [Chauhan and Kumar (2017)]. Rules as used in PostgreSQL are presented in [Stonebraker et al. (1990)]. Many tools and extensions for PostgreSQL are documented by the pgFoundry at www.pgfoundry.org. These include the pgtcl library and the pgAccess administration tool mentioned in this chapter. The pgAdmin tool is described on the Web at www.pgadmin.org. Additional details regarding the database-design tools TOra and PostgreSQL Maestro can be found at tora.sourceforge.net and https://www.sqlmaestro.com/products/postgresql/maestro/, respectively.

The serializable snapshot isolation protocol used in PostgreSQL is described in [Ports and Grittner (2012)].

An open-source alternative to PostgreSQL is MySQL, which is available for noncommercial use under the GNU General Public License. MySQL may be embedded in commercial software that does not have freely distributed source code, but this requires a special license to be purchased. Comparisons between the most recent versions of the two systems are readily available on the Web.

Bibliography

[Chauhan and Kumar (2017)] C. Chauhan and D. Kumar, *PostgreSQL High Performance Cookbook*, Packt Publishing (2017).

[Maymala (2015)] J. Maymala, *PostgreSQL for data architects*, Packt Publ., Birmingham (2015).

[Ports and Grittner (2012)] D. R. K. Ports and K. Grittner, "Serializable Snapshot Isolation in PostgreSQL", *Proceedings of the VLDB Endowment*, Volume 5, Number 12 (2012), pages 1850–1861.

[Schonig (2018)] H.-J. Schonig, *Mastering PostgreSQL 11*, Packt Publishing (2018).

[Stonebraker et al. (1990)] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, "On Rules, Procedure, Caching and Views in Database Systems", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1990), pages 281–290.