instances $B1$, $R2$, and $S3$, sids 22, 29, 31, 32, 58, 64, 74, 85, and 95) and then discard those who have reserved a red boat (sids 22, 31, and 64), to obtain the answer (sids 29, 32, 58, 74, 85, and 95). If we want to compute the names of such sailors, we must first compute their sids (as shown earlier) and then join with Sailors and project the *sname* values.

(Q9) Find the names of sailors who have reserved all boats.

The use of the word *all* (or *every*) is a good indication that the division operation might be applicable:

$$p(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}Boats))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

The intermediate relation Tempsids is defined using division and computes the set of *sids* of sailors who have reserved every boat (over instances *B1*, *R2*, and *S3*, this is just *sid* 22). Note how we define the two relations that the division operator (/) is applied to---the first relation has the schema *(sid,bid)* and the second has the schema *(bid)*. Division then returns all *sids* such that there is a tuple *(sid,bid)* in the first relation for each *bid* in the second. Joining Tempsids with Sailors is necessary to associate names with the selected *sids;* for sailor 22, the name is Dustin.

(Q10) Find the names of sailors who have reserved all boats called Interlake.

$$p(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}(\sigma_{bname='Interlake'}Boats)))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

The only difference with respect to the previous query is that now we apply a selection to Boats, to ensure that we compute *bids* only of boats named *Interlake* in defining the second argument to the division operator. Over instances *E1*, *R2*, and *S3*, Tempsids evaluates to *sids* 22 and 64, and the answer contains their names, Dustin and Horatio.

4.3 RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or *declarative*, in that it allows us to describe the set of answers without being explicit about how they should be computed. Relational calculus has had a big influence on the design of commercial query languages such as SQL and, especially, Query-by-Example (QBE).

The variant of the calculus we present in detail is called the **tuple relational calculus** (TRC). Variables in TRC take on tuples as values. In another vari-

ant, called the domain relational calculus (DRC), the variables range over field values. TRC has had more of an influence on SQL, while DRC has strongly influenced QBE. We discuss DRC in Section 4.3.2.[2]

4.3.1 Tuple Relational Calculus

A tuple variable is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields. A tuple relational calculus query has the form $\{ T \mid p(T) \}$, where T is a tuple variable and $p(T)$ denotes a *formula* that describes T; we will shortly define formulas and queries rigorously. The result of this query is the set of all tuples t for which the formula $p(T)$ evaluates to true with $T = t$. The language for writing formulas $p(T)$ is thus at the heart of TRC and essentially a simple subset of *first-order logic*. As a simple example, consider the following query.

(Q11) Find all sailors with a rating above 7.

$$\{S \mid S \in Sailors \wedge S.rating > 7\}$$

When this query is evaluated on an instance of the Sailors relation, the tuple variable S is instantiated successively with each tuple, and the test $S.rating > 7$ is applied. The answer contains those instances of S that pass this test. On instance *S3* of Sailors, the answer contains Sailors tuples with *sid* 31, 32, 58, 71, and 74.

Syntax of TRC Queries

We now define these concepts formally, beginning with the notion of a formula. Let *Rel* be a relation name, R and S be tuple variables, a be an attribute of R, and b be an attribute of S. Let op denote an operator in the set $\{<, >, =, \leq, \geq, \neq\}$. An atomic formula is one of the following:

- $R \in Rel$
- $R.a$ op $S.b$
- $R.a$ op *constant*, or *constant* op $R.a$

A formula is recursively defined to be one of the following, where p and q are themselves formulas and $p(R)$ denotes a formula in which the variable R appears:

2The material on DRC is referred to in the (online) chapter on QBE; with the exception of this chapter, the material on DRC and TRC can be omitted without loss of continuity.

- any atomic formula
- $\neg p$, $P \wedge q$, $P \vee q$, or $p \Rightarrow q$
- $\exists R(p(R))$, where R is a tuple variable
- $\forall R(p(R))$, where R is a tuple variable

In the last two clauses, the quantifiers $\exists$ and $\forall$ are said to bind the variable R. A variable is said to be free in a formula or *subformuia* (a formula contained in a larger formula) if the (sub)formula does not contain an occurrence of a quantifier that binds it.[3]

We observe that every variable in a TRC formula appears in a subformula that is atomic, and every relation schema specifies a domain for each field; this observation ensures that each variable in a TRC formula has a well-defined domain from which values for the variable are drawn. That is, each variable has a well-defined *type,* in the programming language sense. Informally, an atomic formula $R \in Rel$ gives R the type of tuples in *Rel,* and comparisons such as *R.a* op *S.b* and *R.a* op *constant* induce type restrictions on the field *R.a*. If a variable R does not appear in an atomic formula of the form $R \in Rel$ (Le., it appears only in atomic formulas that are comparisons), we follow the convention that the type of R is a tuple whose fields include all (and only) fields of R that appear in the formula.

We do not define types of variables formally, but the type of a variable should be clear in most cases, and the important point to note is that comparisons of values having different types should always fail. (In discussions of relational calculus, the simplifying assumption is often made that there is a single domain of constants and this is the domain associated with each field of each relation.)

A TRC query is defined to be expression of the form $\{T \mid p(T)\}$, where T is the only free variable in the formula p.

Semantics of TRC Queries

What does a TRC query mean? More precisely, what is the set of answer tuples for a given TRC query? The answer to a TRC query $\{T \mid p(T)\}$, as noted earlier, is the set of all tuples t for which the formula *peT)* evaluates to true with variable T assigned the tuple value t. To complete this definition, we must state which assignments of tuple values to the free variables in a formula make the formula evaluate to true.

[3]We make the assumption that each variable in a formula is either free or bound by exactly one occurrence of a quantifier, to avoid worrying about details such as nested occurrences of quantifiers that bind some, but not all, occurrences of variables.

A query is evaluated on a given instance of the database. Let each free variable in a formula F be bound to a tuple value. For the given assignment of tuples to variables, with respect to the given database instance, F evaluates to (or simply 'is') true if one of the following holds:

- F is an atomic formula $R \in Rel$, and R is assigned a tuple in the instance of relation *Rel*.
- F is a comparison $R.a$ op $S.b$, $R.a$ op *constant*, or *constant* op $R.a$, and the tuples assigned to Rand S have field values $R.a$ and $S.b$ that make the comparison true.
- F is of the form $\neg p$ and p is not true, or of the form $p \wedge q$, and both p and q are true, or of the form $p \vee q$ and one of them is true, or of the form $p \Rightarrow q$ and q is true whenever[4] p is true.
- F is of the form $\exists R(p(R))$, and there is some assignment of tuples to the free variables in $p(R)$, including the variable R,[5] that makes the formula $p(R)$ true.
- F is of the form $\forall R(p(R))$, and there is some assignment of tuples to the free variables in $p(R)$ that makes the formula $p(R)$ true no matter what tuple is assigned to R.

Examples of TRC Queries

We now illustrate the calculus through several examples, using the instances *B1* of Boats, *R2* of Reserves, and *S3* of Sailors shown in Figures 4.15, 4.16, and 4.17. We use parentheses as needed to make our formulas unambiguous. Often, a formula $p(R)$ includes a condition $R \in Rel$, and the meaning of the phrases *some tuple R* and *for all tuples R* is intuitive. We use the notation $\exists R \in Rel(p(R))$ for $\exists R(R \in Rel \wedge p(R))$. Similarly, we use the notation $\forall R \in Rel(p(R))$ for $\forall R(R \in Rel \Rightarrow p(R))$.

(Q12) *Find the names and ages of sailors with a rating above 7.*

$$\{P \mid \exists S \in Sailors(S.rating > 7 \wedge P.name = S.sname \wedge P.age = S.age)\}$$

This query illustrates a useful convention: P is considered to be a tuple variable with exactly two fields, which are called *name* and *age*, because these are the only fields of P mentioned and P does not range over any of the relations in the query; that is, there is no subformula of the form $P \in Relname$. The result of this query is a relation with two fields, *name* and *age*. The atomic

4 *WheneveT* should be read more precisely as 'for all assignments of tuples to the free variables.'
5Note that some of the free variables in $p(R)$ (e.g., the variable R itself) IIay be bound in P.

formulas *P.name* = *S.sname* and *Page* = *S.age* give values to the fields of an answer tuple *P*. On instances *E1*, *R2*, and *S3*, the answer is the set of tuples $\langle Lubber, 55.5\rangle$, $\langle Andy, 25.5\rangle$, $\langle Rusty, 35.0\rangle$, *(Zorba,* 16.0), and $\langle Horatio, 35.0\rangle$.

(Q1S) Find the sailor name, boat'id, and reservation date for each reservation.

$\{P \mid \exists R \in Reserves\ \exists S \in Sailors$
$(R.sid = 8.sid \wedge P.bid = R.bid \wedge P.day = R.day \wedge P.sname = S.sname)\}$

For each Reserves tuple, we look for a tuple in Sailors with the same *sid*. Given a pair of such tuples, we construct an answer tuple *P* with fields *sname*, *bid*, and *day* by copying the corresponding fields from these two tuples. This query illustrates how we can combine values from different relations in each answer tuple. The answer to this query on instances *E1*, *R2*, and 83 is shown in Figure 4.20.

sname	*bid*	*day*
Dustin	101	10/10/98
Dustin	102	10/10/98
Dustin	103	10/8/98
Dustin	104	10/7/98
Lubber	102	11/10/98
Lubber	103	11/6/98
Lubber	104	11/12/98
Horatio	101	9/5/98
Horatio	102	9/8/98
Horatio	103	9/8/98

Figure 4.20 Answer to Query Q13

(Q1) Find the names of sailors who have reserved boat 103.

$\{P \mid \exists S \in Sailors\ \exists R \in Reserves(R.s'id = S.sid \wedge R.b'id = 103$
$\wedge Psname = 8.snarne)\}$

This query can be read as follows: "Retrieve all sailor tuples for which there exists a tuple in Reserves having the same value in the *sid* field and with *b'id* = 103." That is, for each sailor tuple, we look for a tuple in Reserves that shows that this sailor has reserved boat 103. The answer tuple *P* contains just one field, *sname*.

(Q2) Find the names of sailors who have reserved a red boat.

$\{P \mid \exists S \in Sailors\ \exists R \in Reserves(R.sid = 5.sid \wedge P.sname = S.8name$

$\wedge \exists B \in Boats(B.llid = R.bid \wedge B.color = 'red'))\}$

This query can be read as follows: "Retrieve all sailor tuples S for which there exist tuples R in Reserves and B in Boats such that $S.sid = R.sid$, $R.bid = B.bid$, and $B.coior = 'red'$." Another way to write this query, which corresponds more closely to this reading, is as follows:

$\{P \mid \exists S \in SailoTs \ \exists R \in Reserves \ \exists B \in Boats$
$(R.sid = S.sid \wedge B.bid = R.bid \wedge B.color = 'red' \wedge P.sname = S.sname)\}$

(Q7) Find the names of sailors who have reserved at least two boats.

$\{P \mid \exists S \in Sailors \ \exists R1 \in Reserves \ \exists R2 \in Reserves$
$(S.sid = R1.sid \wedge R1.sid = R2.sid \wedge R1.bid \neq R2.bid$
$\wedge P.sname = S.sname)\}$

Contrast this query with the algebra version and see how much simpler the calculus version is. In part, this difference is due to the cumbersome renaming of fields in the algebra version, but the calculus version really is simpler.

(Q9) Find the narnes of sailors who have reserved all boats.

$\{P \mid \exists S \in Sailors \ \forall B \in Boats$
$(\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\}$

This query was expressed using the division operator in relational algebra. Note how easily it is expressed in the calculus. The calculus query directly reflects how we might express the query in English: "Find sailors S such that for all boats B there is a Reserves tuple showing that sailor S has reserved boat B."

(Q14) Find sailors who have reserved all red boats.

$\{S \mid S \in Sailor's \wedge \forall B \in Boats$
$(B.color = 'red' \Rightarrow (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$

This query can be read as follows: For each candidate (sailor), if a boat is red, the sailor must have reserved it. That is, for a candidate sailor, a boat being red must imply that the sailor has reserved it. Observe that since we can return an entire sailor tuple as the ans\ver instead of just the sailor's name, we avoided introducing a new free variable (e.g., the variable P in the previous example) to hold the answer values. On instances B1. *R2*, and *S3*, the answer contains the Sailors tuples with *sids* 22 and 31.

We can write this query without using implication, by observing that an expression of the form p $\Rightarrow$ *q* is logically equivalent to $\neg p \vee q$:

$\{S \mid S \in Sailors \wedge \forall B \in Boats$

$$(B.coioT \neq 'red' \text{ V } (\exists R \text{ E } ReSeTVeS(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query should be read as follows: "Find sailors S such that, for all boats B, either the boat is not red or a Reserves tuple shows that sailor S has reserved boat B."

4.3.2 Domain Relational Calculus

A **domain variable** is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers). A DRC query has the form $\{(X1,X2, \ldots ,X_n) \mid P((X1,X2, \ldots ,X_n))\}$, where each Xi is either a *domain variable* or a constant and $p((Xl, X2, \ldots ,x_n))$ denotes a **DRC formula** whose only free variables are the variables among the Xi, $1 \leq i \leq n$. The result of this query is the set of all tuples $(X1, X2, \ldots , X_n)$ for which the formula evaluates to true.

A DRC formula is defined in a manner very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables. Let op denote an operator in the set $\{<, >, =, \leq, \geq, \neq\}$ and let X and Y be domain variables. An **atomic formula** in DRC is one of the following:

- $(X1, X2, \ldots , X_n)$ E Rel, where Rei is a relation with n attributes; each Xi, $1 \leq i \leq n$ is either a variable or a constant
- X op Y
- X op *constant*, or *constant* op X

A **formula** is recursively defined to be one of the following, where P and q are themselves formulas and $p(X)$ denotes a formula in which the variable X appears:

- any atomic formula
- $\neg p$, $P \wedge q$, $P \vee q$, or $p \Rightarrow q$
- $\exists X(p(X))$, where X is a domain variable
- $\forall X(p(X))$, where X is a domain variable

The reader is invited to compare this definition with the definition of TRC formulas and see how closely these two definitions correspond. We will not define the semantics of DRC formulas formally; this is left as an exercise for the reader.

Examples of DRC Queries

We now illustrate DRC through several examples. The reader is invited to compare these with the TRC versions.

*(Q*11) *Find all sailors with a rating above 7.*

$$\{\langle I, N, T, A\rangle \mid (I, N, T, A) \in Sailors \wedge T > 7\}$$

This differs from the TRC version in giving each attribute a (variable) name. The condition $\langle I, N, T, A\rangle \in Sailors$ ensures that the domain variables I, N, T, and A are restricted to be fields of the *same* tuple. In comparison with the TRC query, we can say $T > 7$ instead of $S.rating > 7$, but we must specify the tuple (I, N, T, A) in the result, rather than just S.

*(Q*1) *Find the names of sailors who have reserved boat 103.*

$$\{(N) \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists Ir, Br, D((11', Br, D) \in Reserves \wedge 11' = I \wedge Br = 103))\}$$

Note that only the *sname* field is retained in the answer and that only N is a free variable. We use the notation $\exists Ir, Br, D(\ldots)$ as a shorthand for $\exists Ir(\exists Br(\exists D(\ldots)))$. Very often, all the quantified variables appear in a single relation, as in this example. An even more compact notation in this case is $\exists\langle Ir, Br, D\rangle \in Reserves$. With this notation, which we use henceforth, the query would be as follows:

$$\{(N) \mid \exists I, T, A((I, N, T, A) \in Sailors \wedge \exists\langle Ir, Br, D\rangle \in Reserves(Ir = I \wedge Br = 103))\}$$

The comparison with the corresponding TRC formula should now be straightforward. This query can also be written as follows; note the repetition of variable I and the use of the constant 103:

$$\{(N) \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists D((1,103, D) \in Reserves))\}$$

(Q2) Find the names of sailors who have Teserved a red boat.

$$\{(N) \mid \exists I, T, A((1, N, T, A) \in Sailors \wedge \exists\langle I, Br, D\rangle \in ReseTves \wedge \exists\langle Br, BN,'Ted') \in Boats)\}$$

(Q7) Find the names of sailors who have TeseTved at least two boats.

$$\{(N) \mid \exists I, T, A((1, N, T, A) \in Sailors \wedge \exists Br1, BT2, Dl, D2((1, Brl, \mathrm{DI}) \in Reserves \wedge (1, Br2, D2) \in Reserves \wedge Brl \neq Br2))\}$$

Note how the repeated use of variable *I* ensures that the same sailor has reserved both the boats in question.

(Q9) Find the names of sailors who have Teserved all boat8.

$$\{(N) \mid \exists I, T, A((I, N, T, A) \in Sailors! \wedge \forall B, BN, C(\neg(\langle B, BN, C) \in Boats) \vee (\exists \langle Ir, Br, D) \in Reserves(I = IT! \wedge BT = B))))\}$$

This query can be read as follows: "Find all values of N such that some tuple (I, N, T, A) in Sailors satisfies the following condition: For every $\langle B, BN, C)$, either this is not a tuple in Boats or there is some tuple *(IT, BT, D)* in Reserves that proves that Sailor I has reserved boat B." The $\forall$ quantifier allows the domain variables B, BN, and C to range over all values in their respective attribute domains, and the pattern '$\neg$((*B, BN,* C) E *Boats*)V' is necessary to restrict attention to those values that appear in tuples of Boats. This pattern is common in DRC formulas, and the notation $\forall\langle B, BN,$ C) E *Boats* can be used as a shortcut instead. This is similar to the notation introduced earlier for $\exists$. With this notation, the query would be written as follows:

$$\{(N) \mid \exists I, T, A((\mathrm{I}, N, T, A) \in Sailors\,! \wedge \forall\langle B, BN, \mathrm{C}) \in Boats (\exists \langle Ir, BT, D) \in ReseTves(I = IT! \wedge BT = B)))\}$$

(Q14) Find sailoTs who have TeseTved all Ted boats.

$$\{(I, N, T, A) \mid (I, N, T, A) \in SailoTs! \wedge \forall\langle B, BN, \mathrm{C}) \in Boats (\mathrm{C} = 'red' \Rightarrow \exists\langle Ir, BT, D) \in Reserves(I = IT! \wedge Br = B))\}$$

Here, we find all sailors such that, for every red boat, there is a tuple in Reserves that shows the sailor has reserved it.

4.4 EXPRESSIVE POWER OF ALGEBRA AND CALCULUS

We presented two formal query languages for the relational model. Are they equivalent in power? Can every query that can be expressed in relational algebra also be expressed in relational calculus? The answer is yes, it can. Can every query that can be expressed in relational calculus also be expressed in relational algebra? Before we answer this question, we consider a major problem with the calculus as we presented it.

Consider the query $\{S \mid \neg(S \in Sailors)\}$. This query is syntactically correct. However, it asks for all tuples S such that S is not in (the given instance of)

Sailors. The set of such S tuples is obviously infinite, in the context of infinite domains such as the set of all integers. This simple example illustrates an *unsafe* query. It is desirable to restrict relational calculus to disallow unsafe queries.

We now sketch how calculus queries are restricted to be safe. Consider a set I of relation instances, with one instance per relation that appears in the query Q. Let $Dom(Q, I)$ be the set of all constants that appear in these relation instances I or in the formulation of the query Q itself. Since we allow only finite instances I, $Dom(Q, I)$ is also finite.

For a calculus formula Q to be considered safe, at a minimum we want to ensure that, for any given I, the set of answers for Q contains only values in $Dom(Q, I)$. While this restriction is obviously required, it is not enough. Not only do we want the set of answers to be composed of constants in $Dom(Q, I)$, we wish to *compute* the set of answers by examining only tuples that contain constants in $Dom(Q, I)$! This wish leads to a subtle point associated with the use of quantifiers $\forall$ and $\exists$: Given a TRC formula of the form $\exists R(p(R))$, we want to find all values for variable R that make this formula true by checking only tuples that contain constants in $Dom(Q, I)$. Similarly, given a TRC formula of the form $\forall R(p(R))$, we want to find any values for variable R that make this formula false by checking only tuples that contain constants in $Dom(Q, I)$.

We therefore define a *safe* TRC formula Q to be a formula such that:

1. For any given I, the set of answers for Q contains only values that are in $Dom(Q, I)$.
2. For each subexpression of the form $\exists R(p(R))$ in Q, if a tuple r (assigned to variable R) makes the formula true, then r contains only constants in $Dom(Q,I)$.
3. For each subexpression of the form $\forall R(p(R))$ in Q, if a tuple r (assigned to variable R) contains a constant that is not in $Dom(Q, I)$, then r must make the formula true.

Note that this definition is not *constructive*, that is, it does not tell us how to check if a query is safe.

The query $Q = \{S \mid \neg(S \in Sailors)\}$ is unsafe by this definition. $Dom(Q,I)$ is the set of all values that appear in (an instance I of) Sailors. Consider the instance SI shown in Figure 4.1. The answer to this query obviously includes values that do not appear in $Dom(Q,SI)$.

Returning to the question of expressiveness, we can show that every query that can be expressed using a *safe* relational calculus query can also be expressed as a relational algebra query. The expressive power of relational algebra is often used as a metric of how powerful a relational database query language is. If a query language can express all the queries that we can express in relational algebra, it is said to be **relationally complete.** A practical query language is expected to be relationally complete; in addition, commercial query languages typically support features that allow us to express some queries that cannot be expressed in relational algebra.

4.5 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What is the input to a relational query? What is the result of evaluating a query? **(Section 4.1)**
- Database systems use some variant of relational algebra to represent query evaluation plans. Explain why algebra is suitable for this purpose. **(Section 4.2)**
- Describe the selection operator. What can you say about the cardinality of the input and output tables for this operator? (That is, if the input has k tuples, what can you say about the output?) Describe the projection operator. What can you say about the cardinality of the input and output tables for this operator? **(Section 4.2.1)**
- Describe the set operations of relational algebra, including union (U), intersection (n), set-difference (-), and cross-product (x). For each, what can you say about the cardinality of their input and output tables? **(Section 4.2.2)**
- Explain how the renaming operator is used. Is it required? That is, if this operator is not allowed, is there any query that can no longer be expressed in algebra? **(Section 4.2.3)**
- Define all the variations of the join operation. Why is the join operation given special attention? Cannot we express every join operation in terms of cross-product, selection, and projection? **(Section 4.2.4)**
- Define the division operation in terms of the basic relational algebra operations. Describe a typical query that calls for division. Unlike join, the division operator is not given special treatment in database systems. Explain why. **(Section 4.2.5)**

- Relational calculus is said to be a *declarative* language, in contrast to algebra, which is a *procedural* language. Explain the distinction. **(Section 4.3)**
- How does a relational calculus query 'describe' result tuples? Discuss the subset of first-order predicate logic used in tuple relational calculus, with particular attention to universal and existential quantifiers, bound and free variables, and restrictions on the query formula. **(Section 4.3.1).**
- What is the difference between tuple relational calculus and domain relational calculus? **(Section 4.3.2).**
- What is an *unsafe* calculus query? Why is it important to avoid such queries? **(Section 4.4)**
- Relational algebra and relational calculus are said to be equivalent in expressive power. Explain what this means, and how it is related to the notion of *relational completeness*. **(Section 4.4)**

EXERCISES

Exercise 4.1 Explain the statement that relational algebra operators can be *composed.* Why is the ability to compose operators important?

Exercise 4.2 Given two relations *R1* and *R2,* where *R1* contains N1 tuples, *R2* contains N2 tuples, and N2 > N1 > 0, give the minimum and maximum possible sizes (in tuples) for the resulting relation produced by each of the following relational algebra expressions. In each case, state any assumptions about the schemas for *R1* and *R2* needed to make the expression meaningful:

(1) R1 ∪ *R2,* (2) *R1* ∩ *R2,* (3) *R1* -- *R2,* (4) *R1* x *R2,* (5) *(Ta=5(R1),* (6) $\pi a(R1)$, and *(7) R1/R2*

Exercise 4.3 Consider the following schema:

Suppliers(*sid:* integer, *sname:* string, *address:* string)
Parts(pid: integer, *pname:* string, *color:* string)
Catalog(*sid:* integer, *pid:* integer, *cost:* real)

The key fields are underlined, and the domain of each field is listed after the field name. Therefore *sid* is the key for Suppliers, *pid* is the key for Parts, and *sid* and *pid* together form the key for Catalog. The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus:

1. Find the *names* of suppliers who supply some red part.
2. Find the *sids* of suppliers who supply some red or green part.
3. Find the *sids* of suppliers who supply some red part or are at 221 Packer Ave.
4. Find the *sids* of suppliers who supply some rcd part and some green part.

5. Find the *sids* of suppliers who supply every part.
6. Find the *sids* of suppliers who supply every red part.
7. Find the *sids* of suppliers who supply every red or green part.
8. Find the *sids* of suppliers who supply every red part or supply every green part.
9. Find pairs of *sids* such that the supplier with the first *sid* charges more for some part than the supplier with the second *sid*.
10. Find the *pids* of parts supplied by at least two different suppliers.
11. Find the *pids* of the most expensive parts supplied by suppliers named Yosemite Sham.
12. Find the *pids* of parts supplied by every supplier at less than $200. (If any supplier either does not supply the part or charges more than $200 for it, the part is not selected.)

Exercise 4.4 Consider the Supplier-Parts-Catalog schema from the previous question. State what the following queries compute:

1. $\pi_{sname}(\pi_{sid}(\sigma_{color='red'} Parts)$!XI *(O'cost<lOoCatalog)* !XI $Suppliers)$
2. π*sname*$(\pi_{sid}((\sigma_{\text{color='red'}} Parts)$!XI $(\sigma_{cost<}$*looCatalog)* !XI $Suppliers))$
3. $(\pi_{sname}$*((O'color='red'* $Parts)$ ⋈ *(crcost<looCatalog)* !XI *Suppl'iers))* n

$$(\pi_{sname}((\sigma_{color='green'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers))$$

4. *(1fsid((crcolor='red,Parts)* !XI *(crcost<10oCatalog)* ⋈ *Suppliers))* n

$$(\pi_{sid}((\sigma_{color='green'} Parts) \bowtie$$ *(crcost<lOoCatalog)* !XI *Suppliers))*

5. $\pi_{sname}((\pi_{sid,sname}((\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers))$ ∩

$$(\pi_{sid,sname}((\sigma$$*CO!07'='green'* $Parts)$!XI $(\sigma$*cost< lOoCatalog)* ⋈ *Suppliers)))*

Exercise 4.5 Consider the following relations containing airline flight information:

Flights(fino: integer, *from:* string, *to:* string,
d·istance: integer, *departs:* time, *arrives:* time)
Aircraft(*aid:* integer, *aname:* string, *cTuisingrange:* integer)
Certified(*eid:* integer, *aid:* integer)
Employees(*eid:* integer, *ename:* string, *salary:* integer)

Note that the Employees relation describes pilots and other kinds of employees as well; every pilot is certified for some aircraft (otherwise, he or she would not qualify as a pilot), and only pilots are certified to fly.

Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus. Note that some of these queries may not be expressible in relational algebra (and, therefore, also not expressible in tuple and domain relational calculus)! For such queries, informally explain why they cannot be expressed. (See the exercises at the end of Chapter 5 for additional queries over the airline schenla.)

1. Finel the *eids* of pilots certified for some Boeing aircraft.
2. Find the *names* of pilots certified for some Boeing aircraft.
3. Find the *aids* of all aircraft that. can be used on non-stop flights from Bonn to Madras.

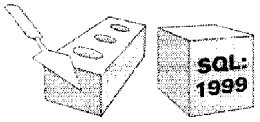
4. Identify the flights that can be piloted by every pilot whose salary is more than $100,000.
5. Find the names of pilots who can operate planes with a range greater than 3,000 miles but are not certified on any Boeing aircraft.
6. Find the *eids* of employees who make the highest salary.
7. Find the *eids* of employees who make the second highest salary.
8. Find the *eids* of employees who are certified for the largest number of aircraft.
9. Find the *eids* of employees who are certified for exactly three aircraft.
10. Find the total amount paid to employees as salaries.
11. Is there a sequence of flights from Madison to Timbuktu? Each flight in the sequence is required to depart from the city that is the destination of the previous flight; the first flight must leave Madison, the last flight must reach Timbuktu, and there is no restriction on the number of intermediate flights. Your query must determine whether a sequence of flights from Madison to Timbuktu exists for *any* input Flights relation instance.

Exercise **4.6** What is *relational completeness?* If a query language is relationally complete, can you write any desired query in that language?

Exercise **4.7** What is an *unsafe* query? Give an example and explain why it is important to disallow such queries.

BIBLIOGRAPHIC NOTES

Relational algebra was proposed by Codd in [187], and he showed the equivalence of relational algebra and TRC in [189]. Earlier, Kuhns [454] considered the use of logic to pose queries. LaCroix and Pirotte discussed DRC in [459]. Klug generalized the algebra and calculus to include aggregate operations in [439]. Extensions of the algebra and calculus to deal with aggregate functions are also discussed in [578]. Merrett proposed an extended relational algebra with quantifiers such as *the number of* that go beyond just universal and existential quantification [530]. Such generalized quantifiers are discussed at length in [52].



5

SQL: QUERIES, CONSTRAINTS, TRIGGERS

- What is included in the SQL language? What is SQL:1999?
- How are queries expressed in SQL? How is the meaning of a query specified in the SQL standard?
- How does SQL build on and extend relational algebra and calculus?
- What is grouping? How is it used with aggregate operations?
- What are nested queries?
- What are *null* values?
- How can we use queries in writing complex integrity constraints?
- What are triggers, and why are they useful? How are they related to integrity constraints?
- **Key concepts:** SQL queries, connection to relational algebra and calculus; features beyond algebra, DISTINCT clause and multiset semantics, grouping and aggregation; nested queries, correlation; set-comparison operators; *null* values, outer joins; integrity constraints specified using queries; triggers and active databases, event-condition-action rules.

What men or gods are these? What Inaiclens loth?
What mad pursuit? What struggle to escape?
\Vhat pipes and tilubrels? \Vhat wild ecstasy?

— John Keats, *Odc on a Grecian Urn*

Structured Query Language (SQL) is the most widely used conunercial relational database language. It was originally developed at IBlVI in the SEQUEL-

SQL Standards Conformance: SQL:1999 has a collection of features called Core SQL that a vendor must implement to claim conformance with the SQL:1999 standard. It is estimated that all the major vendors can comply with Core SQL with little effort. Many of the remaining features are organized into packages.

For example, packages address each of the following (with relevant chapters in parentheses): *enhanced date and time, enhanced integrity management and active databases* (this chapter), *external language interfaces* (Chapter 6), *OLAP* (Chapter 25), and *object features* (Chapter 23). The SQL/MM standard complements SQL:1999 by defining additional packages that support *data mining* (Chapter 26), *spatial data* (Chapter 28) and *text documents* (Chapter 27). Support for XML data and queries is forthcoming.

XRM and System-R projects (1974-1977). Almost immediately, other vendors introduced DBMS products based on SQL, and it is now a de facto standard. SQL continues to evolve in response to changing needs in the database area. The current ANSI/ISO standard for SQL is called SQL:1999. While not all DBMS products support the full SQL:1999 standard yet, vendors are working toward this goal and most products already support the core features. The SQL:1999 standard is very close to the previous standard, SQL-92, with respect to the features discussed in this chapter. Our presentation is consistent with both SQL-92 and SQL:1999, and we explicitly note any aspects that differ in the two versions of the standard.

5.1 OVERVIEW

The SQL language has several aspects to it.

- The Data Manipulation Language (DML): This subset of SQL allows users to pose queries and to insert, delete, and modify rows. Queries are the main focus of this chapter. We covered DML commands to insert, delete, and modify rows in Chapter 3.
- The Data Definition Language (DDL): This subset of SQL supports the creation, deletion, and modification of definitions for tables and views. *Integrity constraints* can be defined on tables, either when the table is created or later. We cocvered the DDL features of SQL in Chapter 3. Although the standard does not discuss indexes, commercial implementations also provide commands for creating and deleting indexes.
- Triggers and Advanced Integrity Constraints: The new SQL:1999 standard includes support for *triggers*, which are actions executed by the

DBMS whenever changes to the database meet conditions specified in the trigger. We cover triggers in this chapter. SQL allows the use of queries to specify complex integrity constraint specifications. We also discuss such constraints in this chapter.

- Embedded and Dynamic SQL: Embedded SQL features allow SQL code to be called from a host language such as C or COBOL. Dynamic SQL features allow a query to be constructed (and executed) at run-time. \Ve cover these features in Chapter 6.
- Client-Server Execution and Remote Database Access: These commands control how a *client* application program can connect to an SQL database *server,* or access data from a database over a network. We cover these commands in Chapter 7.
- Transaction Management: Various commands allow a user to explicitly control aspects of how a transaction is to be executed. We cover these commands in Chapter 21.
- Security: SQL provides mechanisms to control users' access to data objects such as tables and views. We cover these in Chapter 21.
- Advanced features: The SQL:1999 standard includes object-oriented features (Chapter 23), recursive queries (Chapter 24), decision support queries (Chapter 25), and also addresses emerging areas such as data mining (Chapter 26), spatial data (Chapter 28), and text and XML data management (Chapter 27).

5.1.1 Chapter Organization

The rest of this chapter is organized as follows. We present basic SQL queries in Section 5.2 and introduce SQL's set operators in Section 5.3. We discuss nested queries, in which a relation referred to in the query is itself defined within the query, in Section 5.4. We cover aggregate operators, which allow us to write SQL queries that are not expressible in relational algebra, in Section 5.5. \Ve discuss *null* values, which are special values used to indicate unknown or nonexistent field values, in Section 5.6. We discuss complex integrity constraints that can be specified using the SQL DDL in Section 5.7, extending the SQL DDL discussion from Chapter 3; the new constraint specifications allow us to fully utilize the query language capabilities of SQL.

Finally, we discuss the concept of an *active database* in Sections 5.8 and 5.9. An active database has a collection of triggers, which are specified by the DBA. A trigger describes actions to be taken when certain situations arise. The DBMS lllonitors the database, detects these situations, and invokes the trigger.

The SQL:1999 standard requires support for triggers, and several relational DBMS products already support some form of triggers.

About the Examples

We will present a number of sample queries using the following table definitions:

Sailors(*sid:* integer, *sname:* string, *rating:* integer, *age:* real)
Boats(*bid:* integer, *bname:* string, *color:* string)
Reserves(*sid:* integer, *bid:* integer, *day:* date)

We give each query a unique number, continuing with the numbering scheme used in Chapter 4. The first new query in this chapter has number Q15. Queries Q1 through Q14 were introduced in Chapter 4.[1] We illustrate queries using the instances S3 of Sailors, *R2* of Reserves, and *B1* of Boats introduced in Chapter 4, which we reproduce in Figures 5.1, 5.2, and 5.3, respectively.

All the example tables and queries that appear in this chapter are available online on the book's webpage at

http://www.cs.wisc.edu/-dbbook

The online material includes instructions on how to set up Orade, IBM DB2, Microsoft SQL Server, and MySQL, and scripts for creating the example tables and queries.

5.2 THE FORM OF A BASIC SQL QUERY

This section presents the syntax of a simple SQL query and explains its meaning through a *conceptual evaluation strategy.* A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand rather than efficient. A DBMS would typically execute a query in a different and more efficient way.

The basic form of an SQL query is as follows:

```
SELECT [DISTINCT] select-list
FROM   from-list
WHERE  qualification
```

[1] All references to a query can be found in the subject index for the book.

sid	sname	rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 5.1 An Instance 53 of Sailors

sid	bid	day
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 5.2 An Instance *R2* of Reserves

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 5.3 An Instance *B1* of Boats

Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause.

Such a query intuitively corresponds to a relational algebra expression involving selections, projections, and cross-products. The close relationship between SQL and relational algebra is the basis for query optimization in a relational DBMS, as we will see in Chapters 12 and 15. Indeed, execution plans for SQL queries are represented using a variation of relational algebra expressions (Section 15.1).

Let us consider a simple example.

(Q15) Find the' names and ages of all sailors.

```
SELECT DISTINCT S.sname, S.age
FROM    Sailors S
```

The answer is a *set* of rows, each of which is a pair *(sname, age).* If two or more sailors have the same name and age, the answer still contains just one pair

with that name and age. This query is equivalent to applying the projection operator of relational algebra.

If we omit the keyword DISTINCT, we would get a copy of the row *(s,a)* for each sailor with name *s* and age *a;* the answer would be a *multiset* of rows. A **multiset** is similar to a set in that it is an unordered collection of elements, but there could be several copies of each element, and the number of copies is significant-two multisets could have the same elements and yet be different because the number of copies is different for some elements. For example, {a, b, b} and {b, a, b} denote the same multiset, and differ from the multiset {a, a, b}.

The answer to this query with and without the keyword DISTINCT on instance 53 of Sailors is shown in Figures 5.4 and 5.5. The only difference is that the tuple for Horatio appears twice if DISTINCT is omitted; this is because there are two sailors called Horatio and age 35.

snarne	*age*
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Art	25.5
Bob	63.5

Figure 5.4 Answer to Q15

sname	*age*
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Horatio	35.0
Art	25.5
Bob	63.5

Figure 5.5 Answer to Q15 without DISTINCT

Our next query is equivalent to an application of the selection operator of relational algebra.

*(Q*11) *Find all sailors with a rating above 7.*

```
SELECT  S.sid, S.sname, S.rating, S.age
FROM    Sailors AS S
WHERE   S.rating > 7
```

This query uses the optional keyword AS to introduce a range variable. Incidentally, when we want to retrieve all columns, as in this query, SQL provides a

convenient shorthand: We can simply write SELECT *. This notation is useful for interactive querying, but it is poor style for queries that are intended to be reused and maintained because the schema of the result is not clear from the query itself; we have to refer to the schema of the underlying Sailors table.

As these two examples illustrate, the SELECT clause is actually used to do *projection*, whereas *selections* in the relational algebra sense are expressed using the WHERE clause! This mismatch between the naming of the selection and projection operators in relational algebra and the syntax of SQL is an unfortunate historical accident.

We now consider the syntax of a basic SQL query in more detail.

- The from-list in the FROM clause is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The select-list is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The qualification in the WHERE clause is a boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form *expression* op *expression*, where op is one of the comparison operators {<, <=, =, <>, >=, >}.2 An *expression* is a *column* name, a *constant*, or an (arithmetic or string) expression.
- The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain *duplicates*, that is, two copies of the same row. The default is that duplicates are not eliminated.

Although the preceding rules describe (informally) the syntax of a basic SQL query, they do not tell us the *meaning* of a query. The answer to a query is itself a relation which is a *multiset* of rows in SQL!--whose contents can be understood by considering the following conceptual evaluation strategy:

1. Cmnpute the cross-product of the tables in the from-list.
2. Delete rows in the cross-product that fail the qualification conditions.
3. Delete all columns that do not appear in the select-list.
4. If DISTINCT is specified, eliminate duplicate rows.

2ExpressiollS with NOT can always be replaced by equivalent expressions without NOT given the set of comparison operators just listed.

This straightforward conceptual evaluation strategy makes explicit the rows that must be present in the answer to the query. However, it is likely to be quite inefficient. We will consider how a DB:MS actually evaluates queries in later chapters; for now, our purpose is simply to explain the meaning of a query. \Ve illustrate the conceptual evaluation strategy using the following query':

(Q1) *Find the names of sailors 'Who have reseTved boat number 103.*

It can be expressed in SQL as follows.

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid = R.sid AND R.bid=103
```

Let us compute the answer to this query on the instances *R3* of Reserves and 84 of Sailors shown in Figures 5.6 and 5.7, since the computation on our usual example instances *(R2* and 83) would be unnecessarily tedious.

sid	*bid*	*day*
22	101	10/10/96
58	103	11/12/96

Figure 5.6 Instance *R3* of Reserves

sid	*sname*	*Tating*	*age*
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Figure 5.7 Instance 54 of Sailors

The first step is to construct the cross-product 84 x *R3,* which is shown in Figure 5.8.

sid	*sname·j*	*rating*	*age*	*sid*	*bid*	*day*
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	3.5.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Figure 5.8 *S4* x *R3*

The second step is to apply the qualification *S.sid* = *R.sid* AND *R.bid=103.* (Note that the first part of this qualification requires a join operation.) This step eliminates all but the last row from the instance shown in Figure 5.8. The third step is to eliminate unwanted columns; only *sname* appears in the SELECT clause. This step leaves us with the result shown in Figure .5.9, which is a table with a single column and, as it happens, just one row.

sname
rusty

Figure 5.9 Answer to Query Q1 011 R3 and 84

5.2.1 Examples of Basic SQL Queries

We now present several example queries, many of which were expressed earlier in relational algebra and calculus (Chapter 4). Our first example illustrates that the use of range variables is optional, unless they are needed to resolve an ambiguity. Query Q1, which we discussed in the previous section, can also be expressed as follows:

```
SELECT  sname
FROM    Sailors 5, Reserves R
WHERE   S.sid = R.sid AND bid=103
```

Only the occurrences of *sid* have to be qualified, since this column appears in both the Sailors and Reserves tables. An equivalent way to write this query is:

```
SELECT  SHame
FROM    Sailors, Reserves
WHERE   Sailors.sid = Reserves.sid AND bid=103
```

This query shows that table names can be used implicitly as row variables. Range variables need to be introduced explicitly only when the FROM clause contains more than one occurrence of a relation.[3] However, we recommend the explicit use of range variables and full qualification of all occurrences of columns with a range variable to improve the readability of your queries. We will follow this convention in all our examples.

(Q16) Find the sids of sailors who have TeseTved a red boat.

```
SELECT    R.sid
FROM      Boats B, Reserves R
WHERE     B.bid = R.bid AND 8.color = 'red'
```

This query contains a join of two tables, followed by a selection on the color of boats. We can think of 13 and R as rows in the corresponding tables that

[3]The table name cannot be used as an implicit. range variable once a range variable is introduced for t.he relation.

'prove' that a sailor with sid R.sid reserved a reel boat B.bid. On our example instances *R2* and 83 (Figures 5.1 and 5.2), the answer consists of the *sids* 22, 31, and 64. If we want the names of sailors in the result, we must also consider the Sailors relation, since Reserves does not contain this information, as the next example illustrates.

(Q2) Find the names of sailors who have reserved a Ted boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats 13
WHERE   S.sid = R.sid AND R.bid = 13.bid AND B.color = 'red'
```

This query contains a join of three tables followed by a selection on the color of boats. The join with Sailors allows us to find the name of the sailor who, according to Reserves tuple R, has reserved a red boat described by tuple 13.

(Q3) Find the coloTS of boats reseTved by LubbeT.

```
SELECT  13.color
FROM    Sailors S, Reserves R, Boats 13
WHERE   S.sid = R.sid AND R.bid = B.bid AND S.sname = 'Lubber'
```

This query is very similar to the previous one. Note that in general there may be more than one sailor called Lubber (since *sname* is not a key for Sailors); this query is still correct in that it will return the colors of boats reserved by *some* Lubber, if there are several sailors called Lubber.

(Q4) Find the names of sailors who have Teserved at least one boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid = R.sid
```

The join of Sailors and Reserves ensures that for each selected *sname*, the sailor has made some reservation. (If a sailor has not made a reservation, the second step in the conceptual evaluation strategy would eliminate all rows in the cross-product that involve this sailor.)

5.2.2 Expressions and Strings in the SELECT Command

SQL supports a more general version of the select-list than just a list of colu1nn8. Each item in a select-list can be of the form *expression* AS *column_name*, where *expression* is any arithmetic or string expression over column

names (possibly prefixed by range variables) and constants, and *column_name* is a new name for this column in the output of the query. It can also contain *aggregates* such as *sum* and *count*, which we will discuss in Section 5.5. The SQL standard also includes expressions over date and time values, which we will not discuss. Although not part of the SQL standard, many implementations also support the use of built-in functions such as *sqrt, sin,* and *rnod.*

(Q17) Compute increments for the mtings of peTsons who have sailed two different boats on the same day.

```
SELECT  S.sname, S.rating+1 AS rating
FROM    Sailors S, Reserves R1, Reserves R2
WHERE   S.sid = R1.sid AND S.sid = R2.sid
        AND R1.day = R2.day AND R1.bid <> R2.bid
```

Also, each item in a *qualification* can be as general as *expTession1 = expression2.*

```
SELECT  S1.sname AS name1, S2.sname AS name2
FROM    Sailors Sl, Sailors S2
WHERE   2*S1.rating = S2.rating-1
```

For string comparisons, we can use the comparison operators (=, <, >, etc.) with the ordering of strings determined alphabetically as usual. If we need to sort strings by an order other than alphabetical (e.g., sort strings denoting month names in the calendar order January, February, March, etc.), SQL supports a general concept of a collation, or sort order, for a character set. A collation allows the user to specify which characters are 'less than' which others and provides great flexibility in string manipulation.

In addition, SQL provides support for pattern matching through the LIKE operator, along with the use of the wild-card symbols % (which stands for zero or more arbitrary characters) and _ (which stands for exactly one, arbitrary, character). Thus, '_AB%' denotes a pattern matching every string that contains at least three characters, with the second and third characters being A and B respectively. Note that unlike the other comparison operators, blanks can be significant for the LIKE operator (depending on the collation for the underlying character set). Thus, *'Jeff' = 'Jeff'* is true while *'Jeff'LIKE 'Jeff '* is false. An example of the use of LIKE in a query is given below.

(Q18) Find the ages of sailors wh08e name begins and ends with B and has at least three chamcters.

```
SELECT  S.age
```

Regular Expressions in SQL: Reflecting the increased importance of text data, SQL:1999 includes a more powerful version of the LIKE operator called SIMILAR. This operator allows a rich set of regular expressions to be used as patterns while searching text. The regular expressions are similar to those supported by the Unix operating system for string searches, although the syntax is a little different.

Relational Algebra and SQL: The set operations of SQL are available in relational algebra. The main difference, of course, is that they are *multiset* operations in SQL, since tables are multisets of tuples.

```
FROM    Sailors S
WHERE   S.sname LIKE 'B.%B'
```

The only such sailor is Bob, and his age is 63.5.

5.3 UNION, INTERSECT, AND EXCEPT

SQL provides three set-manipulation constructs that extend the basic query form presented earlier. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT.[4] SQL also provides other set operations: IN (to check if an element is in a given set), op ANY, op ALL (to compare a value with the elements in a given set, using comparison operator op), and EXISTS (to check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning. We cover UNION, INTERSECT, and EXCEPT in this section, and the other operations in Section 5.4.

Consider the following query:

(Q5) Find the names of sailors who have reserved a red or a green boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid
        AND (B.color = 'red' OR B.color = 'green')
```

[4]Note that although the SQL standard includes these operations, many systems currently support only UNION. Also. many systems recognize the keyword MINUS for EXCEPT.

This query is easily expressed using the OR connective in the WHERE clause. However, the following query, which is identical except for the use of 'and' rather than 'or' in the English version, turns out to be much more difficult:

(Q6) Find the names of sailors who have reserved both a red and a green boat.

If we were to just replace the use of OR in the previous query by AND, in analogy to the English statements of the two queries, we would retrieve the names of sailors who have reserved a boat that is both red and green. The integrity constraint that *bid* is a key for Boats tells us that the same boat cannot have two colors, and so the variant of the previous query with AND in place of OR will always return an empty answer set. A correct statement of Query Q6 using AND is the following:

```
SELECT S.sname
FROM   Sailors S, Reserves RI, Boats BI, Reserves R2, Boats B2
WHERE  S.sid = Rl.sid AND R1.bid = Bl.bid
       AND S.sid = R2.sid AND R2.bid = B2.bid
       AND B1.color='red' AND B2.color = 'green'
```

We can think of RI and BI as rows that prove that sailor S.sid has reserved a red boat. R2 and B2 similarly prove that the same sailor has reserved a green boat. S.sname is not included in the result unless five such rows S, RI, BI, R2, and B2 are found.

The previous query is difficult to understand (and also quite inefficient to execute, as it turns out). In particular, the similarity to the previous OR query (Query Q5) is completely lost. A better solution for these two queries is to use UNION and INTERSECT.

The OR query (Query Q5) can be rewritten as follows:

```
SELECT S.sname
FROM   Sailors S, Reserves R, Boats B
WHERE  S.sicl = R.sid AND R.bid = B.bid AND B.color = 'red'
UNION
SELECT S2.sname
FROM   Sailors S2, Boats B2, Reserves R2
WHERE  S2.sid = H2.sid AND R2.bid = B2.bicl AND B2.color = 'green'
```

This query says that we want the union of the set of sailors who have reserved red boats and the set of sailors who have reserved green boats. In complete symmetry, the AND query (Query Q6) can be rewritten as follows:

```
SELECT S.snarne
```

```
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
INTERSECT
SELECT  S2.sname
FROM    Sailors S2, Boats B2, Reserves R2
WHERE   S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

This query actually contains a subtle bug-if there are two sailors such as Horatio in our example instances *B1, R2,* and 83, one of whom has reserved a red boat and the other has reserved a green boat, the name Horatio is returned even though no one individual called Horatio has reserved both a red and a green boat. Thus, the query actually computes sailor names such that some sailor with this name has reserved a red boat and some sailor with the same name (perhaps a different sailor) has reserved a green boat.

As we observed in Chapter 4, the problem arises because we are using *sname* to identify sailors, and *sname* is not a key for Sailors! If we select *sid* instead of *sname* in the previous query, we would compute the set of *sids* of sailors who have reserved both red and green boats. (To compute the names of such sailors requires a nested query; we will return to this example in Section 5.4.4.)

Our next query illustrates the set-difference operation in SQL.

*(Q*19) *Find the* sids *of all sailor's who have reserved red boats but not green boats.*

```
SELECT  S.sid
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
EXCEPT
SELECT  S2.sid
FROM    Sailors S2, Reserves R2, Boats B2
WHERE   S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

Sailors 22, 64, and 31 have reserved red boats. Sailors 22, 74, and 31 have reserved green boats. Hence, the answer contains just the *sid 64.*

Indeed, since the Reserves relation contains sid information, there is no need to look at the Sailors relation, and we can use the following simpler query:

```
SELECT  R.sid
FROM    Boats B, Reserves R
WHERE   R.bicl = B.bid AND B.color = 'red'
EXCEPT
```

```
SELECT R2.sid
FROM   Boats B2, Reserves R2
WHERE  R2.bicl = B2.bid AND B2.color = :green'
```

Observe that this query relies on referential integrity; that is, there are no reservations for nonexisting sailors. Note that UNION, INTERSECT, and EXCEPT can be used on *any* two tables that are union-compatible, that is, have the same number of columns and the columns, taken in order, have the same types. For example, we can write the following query:

(Q20) Find all sids of sailors who have a rating of 10 or *reserved boat 104.*

```
SELECT S.sid
FROM   Sailors S
WHERE  S.rating = 10
UNION
SELECT R.sid
FROM   Reserves R
WHERE  R.bid = 104
```

The first part of the union returns the *sids* 58 and 71. The second part returns 22 and 31. The answer is, therefore, the set of *sids* 22, 31, 58, and 71. A final point to note about UNION, INTERSECT, and EXCEPT follows. In contrast to the default that duplicates are not eliminated unless DISTINCT is specified in the basic query form, the default for UNION queries is that duplicates *are* eliminated! To retain duplicates, UNION ALL must be used; if so, the number of copies of a row in the result is always m + n, where m and n are the numbers of times that the row appears in the two parts of the union. Similarly, INTERSECT ALL retains cluplicates--the number of copies of a row in the result is $min(m, n)$—and EXCEPT ALL also retains duplicates—the number of copies of a row in the result is m - n, where 'm corresponds to the first relation.

5.4 NESTED QUERIES

One of the most powerful features of SQL is nested queries. A nested query is a query that has another query embedded within it; the embedded query is called a suhquery. The embedded query can of course be a nested query itself; thus queries that have very deeply nested structures are possible. When writing a query, we sornetimes need to express a condition that refers to a table that must itself be computed. The query used to compute this subsidiary table is a subquery and appears as part of the main query. A subquery typically appears within the WHERE clause of a query. Subqueries can sometimes appear in the FROM clause or the HAVING clause (which we present in Section 5.5).

Relational Algebra and SQL: Nesting of queries is a feature that is not available in relational algebra, but nested queries can be translated into algebra, as we will see in Chapter 15. Nesting in SQL is inspired more by relational calculus than algebra. In conjunction with some of SQL's other features, such as (multi)set operators and aggregation, nesting is a very expressive construct.

This section discusses only subqueries that appear in the WHERE clause. The treatment of subqueries appearing elsewhere is quite similar. Some examples of subqueries that appear in the FROM clause are discussed later in Section 5.5.1.

5.4.1 Introduction to Nested Queries

As an example, let us rewrite the following query, which we discussed earlier, using a nested subquery:

(Q1) Find the names of sailors who have reserved boat 103.

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN ( SELECT R.sid
                  FROM   Reserves R
                  WHERE  R.bid = 103 )
```

The nested subquery computes the (multi)set of *sids* for sailors who have reserved boat 103 (the set contains 22,31, and 74 on instances *R2* and 83), and the top-level query retrieves the names of sailors whose *sid* is in this set. The IN operator allows us to test whether a value is in a given set of elements; an SQL query is used to generate the set to be tested. Note that it is very easy to modify this query to find all sailors who have *not* reserved boat 103-we can just replace IN by NOT IN!

The best way to understand a nested query is to think of it in terms of a conceptual evaluation strategy. In our example, the strategy consists of examining rows in Sailors and, for each such row, evaluating the subquery over Reserves. In general, the conceptual evaluation strategy that we presented for defining the semantics of a query can be extended to cover nested queries as follows: Construct the cross-product of the tables in the FROM clause of the top-level query as before. For each row in the cross-product, while testing the qllalifica-

tion in the WHERE clause, (re)compute the subquery.[5] Of course, the subquery might itself contain another nested subquery, in which case we apply the same idea one more time, leading to an evaluation strategy with several levels of nested loops.

As an example of a multiply nested query, let us rewrite the following query.

(Q2) Find the names of sailors who have reserved a red boat.

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN ( SELECT R.sid
                  FROM   Reserves R
                  WHERE  R.bid IN (SELECT B.bid
                                   FROM   Boats B
                                   WHERE  B.color = 'red'
```

The innermost subquery finds the set of *bids* of red boats (102 and 104 on instance *E1).* The subquery one level above finds the set of *sids* of sailors who have reserved one of these boats. On instances *E1, R2,* and 83, this set of *sids* contains 22, 31, and 64. The top-level query finds the names of sailors whose *sid* is in this set of *sids;* we get Dustin, Lubber, and Horatio.

To find the names of sailors who have not reserved a red boat, wc replace the outermost occurrence of IN by NOT IN, as illustrated in the next query.

(Q21) Find the names of sailors who have not *reserved a red boat.*

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid NOT IN ( SELECT R.sid
                      FROM   Reserves R
                      WHERE  R.bid IN ( SELECT B.bid
                                        FROM   Boats B
                                        WHERE  B.color = 'red' )
```

This qucry computes the names of sailors whose *sid* is *not* in the set 22, 31, and 64.

In contrast to Query Q21, we can modify the previous query (the nested version of Q2) by replacing the inner occurrence (rather than the outer occurence) of

[5]Since the inner subquery in our example does not depend on the 'current' row from the outer query ill any way, you rnight wonder why we have to recompute the subquery for each outer row. For an answer, see Section 5.4.2.

IN with NOT IN. This modified query would compute the names of sailors who have reserved a boat that is not red, that is, if they have a reservation, it is not for a red boat. Let us consider how. In the inner query, we check that *R.bid* is *not* either 102 or 104 (the *bids* of red boats). The outer query then finds the *sids* in Reserves tuples where the *bid* is not 102 or 104. On instances $B1$, $R2$, and 53, the outer query computes the set of *sids* 22, 31, 64, and 74. Finally, we find the names of sailors whose *sid* is in this set.

\Ve can also modify the nested query Q2 by replacing both occurrences of IN with NOT IN. This variant finds the names of sailors who have not reserved a boat that is not red, that is, who have reserved only red boats (if they've reserved any boats at all). Proceeding as in the previous paragraph, on instances *E1, R2,* and 53, the outer query computes the set of *sids* (in Sailors) other than 22, 31, 64, and 74. This is the set 29, 32, 58, 71, 85, and 95. We then find the names of sailors whose *sid* is in this set.

5.4.2 Correlated Nested Queries

In the nested queries seen thus far, the inner subquery has been completely independent of the outer query. In general, the inner subquery could depend on the row currently being examined in the outer query (in terms of our conceptual evaluation strategy). Let us rewrite the following query once more.

(Q1) Find the names of sailors who have reserved boat number 103.

```
SELECT S.sname
FROM   Sailors S
WHERE  EXISTS ( SELECT *
                FROM   Reserves R
                WHERE  R.bid = 103
                       AND R.sid = S.sid )
```

The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty, an implicit comparison with the empty set. Thus, for each Sailor row 5, we test whether the set of Reserves rows *R* such that *R.bid = 103* AND *S.sid = R.sid* is nonempty. If so, sailor 5 has reserved boat 103, and we retrieve the name. '1'he subquery clearly depends on the current row Sand IllUSt be re-evaluated for each row in Sailors. The occurrence of *S* in the subquery (in the form of the literal *S.sid)* is called a *cOTTelation,* and such queries are called *correlated queries.*

This query also illustrates the use of the special symbol * in situations where all we want to do is to check that a qualifying row exists, and do Hot really

want to retrieve any columns from the row. This is one of the two uses of * in the SELECT clause that is good programming style; the other is as an argument of the COUNT aggregate operation, which we describe shortly.

As a further example, by using NOT EXISTS instead of EXISTS, we can compute the names of sailors who have not reserved a red boat. Closely related to EXISTS is the UNIQUE predicate. \When we apply UNIQUE to a subquery, the resulting condition returns true if no row appears twice in the answer to the subquery, that is, there are no duplicates; in particular, it returns true if the answer is empty. (And there is also a NOT UNIQUE version.)

5.4.3 Set-Comparison Operators

We have already seen the set-comparison operators EXISTS, IN, and UNIQUE, along with their negated versions. SQL also supports op ANY and op ALL, where op is one of the arithmetic comparison operators {<, <=, =, <>, >=, >}. (SOME is also available, but it is just a synonym for ANY.)

(Q22) Find sailors whose rating is better than some sailor called Horatio.

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.rating > ANY ( SELECT  S2.rating
                         FROM    Sailors S2
                         WHERE   S2.sname = 'Horatio' )
```

If there are several sailors called Horatio, this query finds all sailors whose rating is better than that of *some* sailor called Horatio. On instance 83, this computes the *sids* 31, 32, 58, 71, and 74. What if there were *no* sailor called Horatio? In this case the comparison *S.rating* > ANY ... is defined to return false, and the query returns an elnpty answer set. To understand comparisons involving ANY, it is useful to think of the comparison being carried out repeatedly. In this example, *S.rating* is successively compared with each rating value that is an answer to the nested query. Intuitively, the subquery must return a row that makes the comparison true, in order for S.*rat'ing* > ANY ... to return true.

(Q23) *Find sailors whose rating is better than every sailor' called Horat·to.*

We can obtain all such queries with a simple modification to Query Q22: Just replace ANY with ALL in the WHERE clause of the outer query. On instance $S3$, we would get the *sids* 58 and 71. If there were no sailor called Horatio, the comparison *S.rating* > ALL ... is defined to return true! The query would then return the names of all sailors. Again, it is useful to think of the comparison

being carried out repeatedly. Intuitively, the comparison must be true for every returned row for *S.rating* > ALL ... to return true.

As another illustration of ALL, consider the following query.

(Q24J Find the sailors with the highest rating.

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.rating >= ALL ( SELECT  S2.rating
                          FROM    Sailors S2 )
```

The subquery computes the set of all rating values in Sailors. The outer WHERE condition is satisfied only when *S.rating* is greater than or equal to each of these rating values, that is, when it is the largest rating value. In the instance 53, the condition is satisfied only for *rating* 10, and the answer includes the *sids* of sailors with this rating, Le., 58 and 71.

Note that IN and NOT IN are equivalent to = ANY and <> ALL, respectively.

5.4.4 More Examples of Nested Queries

Let us revisit a query that we considered earlier using the INTERSECT operator.

(Q6) Find the names of sailors who have reserved both a red and a green boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
        AND S.sid IN ( SELECT  S2.sid
                       FROM    Sailors S2, Boats B2, Reserves R2
                       WHERE   S2.sid = R2.sid AND R2.bid = B2.bid
                               AND B2.color = 'green' )
```

This query can be understood as follows: "Find all sailors who have reserved a red boat and, further, have *sids* that are included in the set of *sids* of sailors who have reserved a green boat." This formulation of the query illustrates how queries involving INTERSECT can be rewritten using IN, which is useful to know if your system does not support INTERSECT. Queries using EXCEPT can be similarly rewritten by using NOT IN. To find the *sids* of sailors who have reserved red boats but not green boats, we can simply replace the keyword IN in the previous query by NOT IN.

As it turns out, writing this query (Q6) using INTERSECT is more complicated because we have to use *sids* to identify sailors (while intersecting) and have to return sailor names:

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN (( SELECT R.sid
                   FROM   Boats B, Reserves R
                   WHERE  R.bid = B.bid AND B.color = 'red' )
                   INTERSECT
                   (SELECT R2.sid
                   FROM   Boats B2, Reserves R2
                   WHERE  R2.bid = B2.bid AND B2.color = 'green' ))
```

Our next example illustrates how the *division* operation in relational algebra can be expressed in SQL.

(Q9) Find the names of sailors who have TeseTved all boats.

```
SELECT S.sname
FROM   Sailors S
WHERE  NOT EXISTS (( SELECT B.bid
                     FROM   Boats B )
                     EXCEPT
                     (SELECT R.bid
                     FROM   Reserves R
                     WHERE  R.sid = S.sid ))
```

Note that this query is correlated--for each sailor *S,* we check to see that the set of boats reserved by *S* includes every boat. An alternative way to do this query without using EXCEPT follows:

```
SELECT S.sname
FROM   Sailors S
WHERE  NOT EXISTS ( SELECT B.bid
                    FROM   Boats B
                    WHERE  NOT EXISTS ( SELECT R.bid
                                        FROM   Reserves R
                                        WHERE  R.bid = B.bid
                                               AND R.sid = S.sid ))
```

Intuitively, for each sailor we check that there is no boat that has not been reserved by this sailor.

SQL:1999 Aggregate **Functions:** The collection of aggregate functions is greatly expanded in the new standard, including several statistical functions such as standard deviation, covariance, and percentiles. However, the new aggregate functions are in the SQLjOLAP package and may not be supported by all vendors.

5.5 AGGREGATE OPERATORS

In addition to simply retrieving data, we often want to perform some computation or summarization. As we noted earlier in this chapter, SQL allows the use of arithmetic expressions. We now consider a powerful class of constructs for computing *aggregate values* such as MIN and SUM. These features represent a significant extension of relational algebra. SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.

Note that it does not make sense to specify DISTINCT in conjunction with MIN or MAX (although SQL does not preclude this).

(Q25) Find the average age of all sailors.

```
SELECT AVG (S.age)
FROM   Sailors S
```

On instance 53, the average age is 37.4. Of course, the WHERE clause can be used to restrict the sailors considered in computing the average age.

(Q26) Find the average age of sailors with a rating of 10.

```
SELECT AVG (S.age)
FROM   Sailors S
WHERE  S.rating = 10
```

There are two such sailors, and their average age is 25.5. MIN (or MAX) can be used instead of AVG in the above queries to find the age of the youngest (oldest)

sailor. However) finding both the name and the age of the oldest sailor is more tricky, as the next query illustrates.

(Q27) Find the name and age of the oldest sailor.

Consider the following attempt to answer this query:

```
SELECT S.sname, MAX (S.age)
FROM   Sailors S
```

The intent is for this query to return not only the maximum age but also the name of the sailors having that age. However, this query is illegal in SQL-if the SELECT clause uses an aggregate operation, then it must use *only* aggregate operations unless the query contains a GROUP BY clause! (The intuition behind this restriction should become clear when we discuss the GROUP BY clause in Section 5.5.1.) Therefore, we cannot use MAX (S.age) as well as S.sname in the SELECT clause. We have to use a nested query to compute the desired answer to Q27:

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.age = ( SELECT MAX (S2.age)
                 FROM Sailors S2 )
```

Observe that we have used the result of an aggregate operation in the subquery as an argument to a comparison operation. Strictly speaking, we are comparing an age value with the result of the subquery, which is a relation. However, because of the use of the aggregate operation, the subquery is guaranteed to return a single tuple with a single field, and SQL converts such a relation to a field value for the sake of the comparison. The following equivalent query for Q27 is legal in the SQL standard but, unfortunately, is not supported in many systems:

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  ( SELECT MAX (S2.age)
         FROM Sailors S2 ) = S.age
```

We can count the number of sailors using COUNT. This example illustrates the use of * as an argument to COUNT, which is useful when \ve want to count all rows.

(Q28) Count the number of sailors.

```
SELECT COUNT (*)
```

```
FROM    Sailors S
```

We can think of * as shorthand for all the columns (in the cross-product of the **from-list** in the FROM clause). Contrast this query with the following query, which computes the number of distinct sailor names. (Remember that *sname* is not a key!)

(Q29) Count the number of different sailor names.

```
SELECT  COUNT ( DISTINCT  S.sname )
FROM    Sailors S
```

On instance 83, the answer to Q28 is 10, whereas the answer to Q29 is 9 (because two sailors have the same name, Horatio). If DISTINCT is omitted, the answer to Q29 is 10, because the name Horatio is counted twice. If COUNT does not include DISTINCT, then COUNT (*) gives the same answer as COUNT *(x)* , where *x* is any set of attributes. In our example, without DISTINCT Q29 is equivalent to Q28. However, the use of COUNT (*) is better querying style, since it is immediately clear that all records contribute to the total count.

Aggregate operations offer an alternative to the ANY and ALL constructs. For example, consider the following query:

(Q30) Find the names of sailors who are older than the oldest sailor with a rating of 10.

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.age > ( SELECT  MAX  ( S2.age )
                  FROM    Sailors S2
                  WHERE   S2.rating = 10 )
```

On instance 83, the oldest sailor with rating 10 is sailor 58, whose age is 35. The names of older sailors are Bob, Dustin, Horatio, and Lubber. Using ALL, this query could alternatively be written as follows:

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.age > ALL  ( SELECT  S2.age
                       FROM    Sailors S2
                       WHERE   S2.rating = 10 )
```

However, the ALL query is more error proncone could easily (and incorrectly!) use ANY instead of ALL, and retrieve sailors who are older than *some* sailor with

Relational Algebra and SQL: Aggregation is a fundamental operation that canIlot be expressed in relational algebra. Similarly, SQL's grouping construct cannot be expressed in algebra.

a rating of 10. The use of ANY intuitively corresponds to the use of MIN, instead of MAX, in the previous query.

5.5.1 The GROUP BY and HAVING Clauses

Thus far, we have applied aggregate operations to all (qualifying) rows in a relation. Often we want to apply aggregate operations to each of a number of groups of rows in a relation, where the number of groups depends on the relation instance (i.e., is not known in advance). For example, consider the following query.

(Q31) Find the age of the youngest sailor for each rating level.

If we know that ratings are integers in the range 1 to la, we could write 10 queries of the form:

```
SELECT MIN (S.age)
FROM   Sailors S
WHERE  S.rating = i
```

where i = 1,2, ... ,10. Writing 10 such queries is tedious. More important, we may not know what rating levels exist in advance.

To write such queries, we need a major extension to the basic SQL query form, namely, the GROUP BY clause. In fact, the extension also includes an optional HAVING clause that can be used to specify qualificatioIls over groups (for example, we may be interested only in rating levels> 6. The general form of an SQL query with these extensions is:

```
SELECT    [ DISTINCT] select-list
FROM      from-list
WHERE     'qualification
GROUP BY  grouping-list
HAVING    group-qualification
```

Using the GROUP BY clause, we can write Q31 as follows:

```
SELECT    S.rating, MIN (S.age)
```

```
FROM      Sailors S
GROUP BY  S.rating
```

Let us consider some important points concerning the new clauses:

- The select-list in the SELECT clause consists of (1) a list of column names and (2) a list of terms having the form aggop (*column-name*) AS *new-name*. We already saw AS used to rename output columns. Columns that are the result of aggregate operators do not already have a column name, and therefore giving the column a name with AS is especially useful.

  Every column that appears in (1) must also appear in grouping-list. The reason is that each row in the result of the query corresponds to one *group*, which is a collection of rows that agree on the values of columns in grouping-list. In general, if a column appears in list (1), but not in grouping-list, there can be multiple rows within a group that have different values in this column, and it is not clear what value should be assigned to this column in an answer row.

  We can sometimes use primary key information to verify that a column has a unique value in all rows within each group. For example, if the grouping-list contains the primary key of a table in the from-list, every column of that table has a unique value within each group. In SQL:1999, such columns are also allowed to appear in part (1) of the select-list.

- The expressions appearing in the group-qualification in the HAVING clause must have a *single* value per group. The intuition is that the HAVING clause determines whether an answer row is to be generated for a given group. To satisfy this requirement in SQL-92, a column appearing in the group-qualification must appear as the argument to an aggregation operator, or it must also appear in grouping-list. In SQL:1999, two new set functions have been introduced that allow us to check whether *every* or *any* row in a group satisfies a condition; this allows us to use conditions similar to those in a WHERE clause.

- If GROUP BY is omitted, the entire table is regarded as a single group.

We explain the semantics of such a query through an example.

(Q32) Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors.

```
SELECT    S.rating, MIN (S.age) AS minage
FROM      Sailors S
WHERE     S.age >= 18
GROUP BY  S.rating
HAVING    COUNT (*) > 1
```

We will evaluate this query on instance 83 of Sailors, reproduced in Figure 5.10 for convenience. The instance of Sailors on which this query is to be evaluated is shown in Figure 5.10. Extending the conceptual evaluation strategy presented in Section 5.2, we proceed as follows. The first step is to construct the cross-product of tables in the from-list. Because the only relation in the from-list in Query Q32 is Sailors, the result is just the instance shown in Figure 5.10.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5
96	Frodo	3	25.5

Figure 5.10 Instance 53 of Sailors

The second step is to apply the qualification in the WHERE clause, *S.age* >= 18. This step eliminates the row (71, *zorba,* 10, 16). The third step is to eliminate unwanted columns. Only columns mentioned in the SELECT clause, the GROUP BY clause, or the HAVING clause are necessary, which means we can eliminate *sid* and *sname* in our example. The result is shown in Figure 5.11. Observe that there are two identical rows with *rating* 3 and *age* 25.5-SQL does not eliminate duplicates except when required to do so by use of the DISTINCT keyword! The number of copies of a row in the intermediate table of Figure 5.11 is determined by the number of rows in the original table that had these values in the projected columns.

The fourth step is to sort the table according to the GROUP BY clause to identify the groups. The result of this step is shown in Figure 5.12.

The fifth step is to apply the group-qualification in the HAVING clause, that is, the condition COUNT (*) > 1. This step eliminates the groups with *rating* equal to 1, 9, and 10. Observe that the order in which the WHERE and GROUP BY clauses are considered is significant: If the WHERE clause were not considered first, the group with *rating=10* would have met the group-qualification in the HAVING clause. The sixth step is to generate one answer row for each remaining group. The answer row corresponding to a group consists of a subset

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
9	35.0
3	25.5
3	63.5
3	25.5

Figure 5.11 After Evaluation Step 3

rating	age
11	33.0
3	25.5
3	25.5
3	63.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

Figure 5.12 After Evaluation Step 4

of the grouping columns, plus one or more columns generated by applying an aggregation operator. In our example, each answer row has a *rating* column and a *minage* column, which is computed by applying MIN to the values in the *age* column of the corresponding group. The result of this step is shown in Figure 5.13.

rating	minage
3	25.5
7	35.0
8	25.5

Figure 5.13 Final Result in Sample Evaluation

If the query contains DISTINCT in the SELECT clause, duplicates are eliminated in an additional, and final, step.

SQL:1999 has introduced two new set functions, EVERY and ANY. To illustrate these functions, we can replace the HAVING clause in our example by

```
HAVING  COUNT (*) > 1 AND EVERY ( S.age <= 60 )
```

The fifth step of the conceptual evaluation is the one affected by the change in the HAVING clause. Consider the result of the fourth step, shown in Figure 5.12. The EVERY keyword requires that every row in a group must satisfy the attached condition to meet the group-qualification. The group for *rating* 3 does meet this criterion and is dropped; the result is shown in Figure 5.14.

SQL:1999 Extensions: Two new set functions, EVERY and ANY, have been added. When they are used in the HAVING clause, the basic intuition that the clause specifies a condition to be satisfied by each group, taken as a whole, remains unchanged. However, the condition can now involve tests on individual tuples in the group, whereas it previously relied exclusively on aggregate functions over the group of tuples.

It is worth contrasting the preceding query with the following query, in which the condition on *age* is in the WHERE clause instead of the HAVING clause:

```
SELECT   S.rating, MIN (S.age) AS minage
FROM     Sailors S
WHERE    S.age >= 18 AND S.age <= 60
GROUP BY S.rating
HAVING   COUNT (*) > 1
```

Now, the result after the third step of conceptual evaluation no longer contains the row with *age* 63.5. Nonetheless, the group for *rating* 3 satisfies the condition COUNT (*) > 1, since it still has two rows, and meets the group-qualification applied in the fifth step. The final result for this query is shown in Figure 5.15.

rating	*minage*
7	45 0
8	55.5

Figure 5.14 Final Result of EVERY Query

rating	*minage*
3	25.5
7	45.0
8	55.5

Figure 5.15 Result of Alternative Query

5.5.2 More Examples of Aggregate Queries

(Q33) For each red boat; find the number of reservations for this boat.

```
SELECT   B.bid, COUNT (*) AS reservationcount
FROM     Boats B, Reserves R
WHERE    R.bid = B.bid AND B.color = 'red'
GROUP BY B.bid
```

On instances *B1* and *R2,* the answer to this query contains the two tuples (102, 3) and (104, 2).

Observe that this version of the preceding query is illegal:

```
SELECT    B.bicl, COUNT (*) AS reservationcount
FROM      Boats B, Reserves R
WHERE     R.bid = B.bid
GROUP BY  B.bid
HAVING    B.color = 'red'
```

Even though the gToup-qualification *B.coloT = 'Ted'is* single-valued per group, since the grouping attribute *bid* is a key for Boats (and therefore determines *coloT)*, SQL disallows this query.6 Only columns that appear in the GROUP BY clause can appear in the HAVING clause, unless they appear as arguments to an aggregate operator in the HAVING clause.

(Q34) Find the avemge age of sailoTs fOT each rating level that has at least two sailoTs.

```
SELECT    S.rating, AVG (S.age) AS avgage
FROM      Sailors S
GROUP BY  S.rating
HAVING    COUNT (*) > 1
```

After identifying groups based on *mting,* we retain only groups with at least two sailors. The answer to this query on instance 83 is shown in Figure 5.16.

mting	*avgage*
3	44.5
7	40.0
8	40.5
10	25.5

Figure 5.16 Q34 Answer

mting	*avgage*
3	45.5
7	40.0
8	40.5
10	35.0

Figure 5.17 Q35 Answer

rating	*avgage*
3	45.5
7	40.0
8	40.5

Figure 5.18 Q36 Answer

The following alternative formulation of Query Q34 illustrates that the HAVING clause can have a nested subquery, just like the WHERE clause. Note that we can use *S.rating* inside the nested subquery in the HAVING clause because it has a single value for the current group of sailors:

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
GROUP BY  S.rating
HAVING    1 < ( SELECT COUNT (*)
                FROM   Sailors S2
                WHERE  S.rating = S2.rating )
```

[6]This query can be easily rewritten to be legal in SQL:1999 using EVERY in the HAVING clause.

(Q35) Find the average age of sailors who aTe of voting age (i.e., at least 18 years old) for each rating level that has at least two sailors.

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
WHERE     S. age >= 18
GROUP BY  S.rating
HAVING    1 < ( SELECT COUNT (*)
                FROM   Sailors S2
                WHERE  S.rating = S2.rating )
```

In this variant of Query Q34, we first remove tuples with *age* <= 18 and group the remaining tuples by *rating*. For each group, the subquery in the HAVING clause computes the number of tuples in Sailors (without applying the selection *age* <= 18) with the same *rating* value as the current group. If a group has less than two sailors, it is discarded. For each remaining group, we output the average age. The answer to this query on instance 53 is shown in Figure 5.17. Note that the answer is very similar to the answer for Q34, with the only difference being that for the group with rating 10, we now ignore the sailor with age 16 while computing the average.

(Q36) Find the average age oj sailors who aTe of voting age (i.e., at least 18 yeaTs old) JOT each rating level that has at least two such *sailors.*

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
WHERE     S. age> 18
GROUP BY  S.rating
HAVING    1 < ( SELECT COUNT (*)
                FROM   Sailors S2
                WHERE  S.rating = S2.rating AND S2.age >= 18 )
```

This formulation of the query reflects its similarity to Q35. The answer to Q36 on instance 53 is shown in Figure 5.18. It differs from the answer to Q35 in that there is no tuple for rating 10, since there is only one tuple with rating 10 and *age* $\geq$ 18.

Query Q36 is actually very similar to Q32, as the following simpler formulation shows:

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
WHERE     S. age> 18
GROUP BY  S.rating
```

```
HAVING    COUNT (*) > 1
```

This formulation of Q36 takes advantage of the fact that the WHERE clause is applied before grouping is done; thus, only sailors with *age* > 18 are left when grouping is done. It is instructive to consider yet another way of writing this query:

```
SELECT Temp.rating, Temp.avgage
FROM   ( SELECT    S.rating, AVG ( S.age ) AS avgage,
                   COUNT (*) AS ratingcount
         FROM      Sailors S
         WHERE     S. age> 18
         GROUP BY  S.rating) AS Temp
WHERE Temp.ratingcount > 1
```

This alternative brings out several interesting points. First, the FROM clause can also contain a nested subquery according to the SQL standard.[7] Second, the HAVING clause is not needed at all. Any query with a HAVING clause can be rewritten without one, but many queries are simpler to express with the HAVING clause. Finally, when a subquery appears in the FROM clause, using the AS keyword to give it a name is necessary (since otherwise we could not express, for instance, the condition *Temp. ratingcount* > 1).

(Q37) Find those ratings for which the average age of sailors is the m'inirnum over all ratings.

We use this query to illustrate that aggregate operations cannot be nested. One might consider writing it as follows:

```
SELECT    S.rating
FROM      Sailors S
WHERE     AVG (S.age) = ( SELECT    MIN (AVG (S2.age))
                          FROM      Sailors S2
                          GROUP BY  S2.rating )
```

A little thought shows that this query will not work even if the expression MIN (AVG (S2.age)), which is illegal, were allowed. In the nested query, Sailors is partitioned into groups by rating, and the average age is computed for each rating value. For each group, applying MIN to this average age value for the group will return the same value! A correct version of this query follows. It essentially computes a temporary table containing the average age for each rating value and then finds the rating(s) for which this average age is the minimum.

7Not all commercial database systems currently support nested queries in the FROM clause.

The Relational Model and SQL: Null values are not part of the basic relational model. Like SQL's treatment of tables as multisets of tuples, this is a departure from the basic model.

```
SELECT Temp.rating, Temp.avgage
FROM   ( SELECT   S.rating, AVG (S.age) AS avgage,
         FROM     Sailors S
         GROUP BY S.rating) AS Temp
WHERE  Temp.avgage = ( SELECT MIN (Temp.avgage) FROM Temp)
```

The answer to this query on instance 53 is $\langle 10, 25.5 \rangle$.

As an exercise, consider whether the following query computes the same answer.

```
SELECT   Temp.rating, MIN (Temp.avgage )
FROM     ( SELECT   S.rating, AVG (S.age) AS avgage,
           FROM     Sailors S
           GROUP BY S.rating) AS Temp
GROUP BY Temp.rating
```

5.6 NULL VALUES

Thus far, we have assumed that column values in a row are always known. In practice column values can be unknown. For example, when a sailor, say Dan, joins a yacht club, he may not yet have a rating assigned. Since the definition for the Sailors table has a *rating* column, what row should we insert for Dan? What is needed here is a special value that denotes *unknown*. Suppose the Sailor table definition was modified to include a *rnaiden-name* column. However, only married women who take their husband's last name have a maiden name. For women who do not take their husband's name and for men, the *maiden-name* column is *inapplicable*. Again, what value do we include in this column for the row representing Dan?

SQL provides a special column value called *null* to use in such situations. We use *null* when the column value is either *unknown* or *inapplicable*. Using our Sailor table definition, we might enter the row (98. *Dan, null*, 39) to represent Dan. The presence of *null* values complicates rnany issues, and we consider the impact of *null* values on SQL in this section.

5.6.1 Comparisons Using Null Values

Consider a comparison such as *rating* = 8. If this is applied to the row for Dan, is this condition true or false? Since Dan's rating is unknown, it is reasonable to say that this comparison should evaluate to the value unknown. In fact, this is the case for the comparisons *rating* > 8 and *rating* < 8 as well. Perhaps less obviously, if we compare two *null* values using <, >, =, and so on, the result is always unknown. For example, if we have *null* in two distinct rows of the sailor relation, any comparison returns unknown.

SQL also provides a special comparison operator IS NULL to test whether a column value is *null*; for example, we can say *rating* IS NULL, which would evaluate to true on the row representing Dan. We can also say *rating* IS NOT NULL, which would evaluate to false on the row for Dan.

5.6.2 Logical Connectives AND, OR, and NOT

Now, what about boolean expressions such as *rating* = 8 OR *age* < 40 and *mting* = 8 AND *age* < 40? Considering the row for Dan again, because *age* < 40, the first expression evaluates to true regardless of the value of *rating*, but what about the second? We can only say unknown.

But this example raises an important point—once we have *null* values, we must define the logical operators AND, OR, and NOT using a *three-valued* logic in which expressions evaluate to true, false, or unknown. We extend the usual interpretations of AND, OR, and NOT to cover the case when one of the arguments is unknown as follows. The expression NOT unknown is defined to be unknown. OR of two arguments evaluates to true if either argument evaluates to true, and to unknown if one argument evaluates to false and the other evaluates to unknown. (If both arguments are false, of course, OR evaluates to false.) AND of two arguments evaluates to false if either argument evaluates to false, and to unknown if one argument evaluates to unknown and the other evaluates to true or unknown. (If both arguments are true, AND evaluates to true.)

5.6.3 Impact on SQL Constructs

Boolean expressions arise in many contexts in SQL, and the impact of *null* values must be recognized. For example, the qualification in the WHERE clause eliminates rows (in the cross-product of tables named in the FROM clause) for which the qualification does not evaluate to true. Therefore, in the presence of *null* values, any row that evaluates to false or unknown is eliminated. Eliminating rows that evaluate to unknown has a subtle but signifieant impaet on queries, especially nested queries involving EXISTS or UNIQUE.

Another issue in the presence of *null* values is the definition of when two rows in a relation instance are regarded as *duplicates*. The SQL definition is that two rows are duplicates if corresponding columns are either equal, or both contain *null*. Contrast this definition with the fact that if we compare two *null* values using =, the result is unknown! In the context of duplicates, this comparison is implicitly treated as true, which is an anomaly.

As expected, the arithmetic operations +, -, *, and / all return *null* if one of their arguments is *null*. However, nulls can cause some unexpected behavior with aggregate operations. COUNT(*) handles *'null* values just like other values; that is, they get counted. All the other aggregate operations (COUNT, SUM, AVG, MIN, MAX, and variations using DISTINCT) simply discard *null* values—thus SUM cannot be understood as just the addition of all values in the (multi)set of values that it is applied to; a preliminary step of discarding all *null* values must also be accounted for. As a special case, if one of these operators-other than COUNT-is applied to *only* null values, the result is again *null*.

5.6.4 Outer Joins

Some interesting variants of the join operation that rely on *null* values, called outer joins, are supported in SQL. Consider the join of two tables, say Sailors $\bowtie_c$ Reserves. Tuples of Sailors that do not match some row in Reserves according to the join condition c do not appear in the result. In an outer join, on the other hanel, Sailor rows without a matching Reserves row appear exactly once in the result, with the result columns inherited from Reserves assigned *null* values.

In fact, there are several variants of the outer join idea. In a left outer join, Sailor rows without a matching Reserves row appear in the result, but not vice versa. In a right outer join, Reserves rows without a matching Sailors row appear in the result, but not vice versa. In a **full** outer join, both Sailors and Reserves rows without a match appear in the result. (Of course, rows with a match always appear in the result, for all these variants, just like the usual joins, sometimes called *inner* joins, presented in Chapter 4.)

SQL allows the desired type of join to be specified in the FROM clause. For example, the following query lists *(sid, b'id)* pairs corresponding to sailors and boats they have reserved:

```
SELECT  S.sid, R.bid
FROM    Sailors S NATURAL LEFT OUTER JOIN Reserves R
```

The NATURAL keyword specifies that the join condition is equality on all common attributes (in this example, *sid),* and the WHERE clause is not required (unless

we want to specify additional, non-join conditions). On the instances of Sailors and Reserves shown in Figure 5.6, this query computes the result shown in Figure 5.19.

sid	*bid*
22	101
31	*null*
58	103

Figure 5.19 Left Outer Join of *Sailor1* and *Reserves1*

5.6.5 Disallowing Null Values

We can disallow *null* values by specifying NOT NULL as part of the field definition; for example, *sname* CHAR(20) NOT NULL. In addition, the fields in a primary key are not allowed to take on *null* values. Thus, there is an implicit NOT NULL constraint for every field listed in a PRIMARY KEY constraint.

Our coverage of *null* values is far from complete. The interested reader should consult one of the many books devoted to SQL for a more detailed treatment of the topic.

5.7 COMPLEX INTEGRITY CONSTRAINTS IN SQL

In this section we discuss the specification of complex integrity constraints that utilize the full power of SQL queries. The features discussed in this section complement the integrity constraint features of SQL presented in Chapter 3.

5.7.1 Constraints over a Single Table

We can specify complex constraints over a single table using table constraints, which have the form CHECK *conditional-expression*. For example, to ensure that *rating* must be an integer in the range 1 to 10, we could use:

```
CREATE TABLE Sailors ( sid      INTEGER,
                       sname    CHAR(10),
                       rating   INTEGER,
                       age      REAL,
                       PRIMARY KEY (sid),
                       CHECK (rating >= 1 AND rating <= 10 ))
```

To enforce the constraint that Interlake boats cannot be reserved, we could use:

```
CREATE TABLE Reserves (sid      INTEGER,
                       bid      INTEGER,
                       day      DATE,
                       FOREIGN KEY (sid) REFERENCES Sailors
                       FOREIGN KEY (bid) REFERENCES Boats
                       CONSTRAINT noInterlakeRes
                       CHECK ( 'Interlake' <>
                               ( SELECT B.bname
                                 FROM   Boats B
                                 WHERE  B.bid = Reserves.bid )))
```

When a row is inserted into Reserves or an existing row is modified, the *conditional expression* in the CHECK constraint is evaluated. If it evaluates to false, the command is rejected.

5.7.2 Domain Constraints and Distinct Types

A user can define a new domain using the CREATE DOMAIN statement, which uses CHECK constraints.

```
CREATE DOMAIN ratingval INTEGER DEFAULT 1
                        CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

INTEGER is the underlying, or *source*, type for the domain ratingval, and every ratingval value must be of this type. Values in ratingval are further restricted by using a CHECK constraint; in defining this constraint, we use the keyword VALUE to refer to a value in the domain. By using this facility, we can constrain the values that belong to a domain using the full power of SQL queries. Once a domain is defined, the name of the domain can be used to restrict column values in a table; we can use the following line in a schema declaration, for example:

```
rating   ratingval
```

The optional DEFAULT keyword is used to associate a default value with a domain. If the domain ratingval is used for a column in some relation and no value is entered for this column in an inserted tuple, the default value 1 associated with ratingval is used.

SQL's support for the concept of a domain is limited in an important respect. For example, we can define two domains called SailorId and BoatId, each

SQL:1999 Distinct Types: :Many systems, e.g., Informix UDS and IBM DB2, already support this feature. With its introduction, we expect that the support for domains will be *deprecated*, and eventually eliminated, in future versions of the SQL standard. It is really just one part of a broad set of object-oriented features in SQL:1999, which we discuss in Chapter 23.

using INTEGER as the underlying type. The intent is to force a comparison of a SailorId value with a BoatId value to always fail (since they are drawn from different domains); however, since they both have the same base type, INTEGER, the comparison will succeed in SQL. This problem is addressed through the introduction of distinct types in SqL:1999:

```
CREATE TYPE ratingtype AS INTEGER
```

This statement defines a new *distinct* type called ratingtype, with INTEGER as its source type. Values of type ratingtype can be compared with each other, but they cannot be compared with values of other types. In particular, ratingtype values are treated as being distinct from values of the source type, INTEGER—we cannot compare them to integers or combine them with integers (e.g., add an integer to a ratingtype value). If we want to define operations on the new type, for example, an *average* function, we must do so explicitly; none of the existing operations on the source type carryover. We discuss how such functions can be defined in Section 23.4.1.

5.7.3 Assertions: ICs over Several Tables

Table constraints are associated with a single table, although the conditional expression in the CHECK clause can refer to other tables. Table constraints are required to hold *only* if the a,ssociated table is nonempty. Thus, when a constraint involves two or more tables, the table constraint mechanism is sometimes cumbersome and not quite what is desired. To cover such situations, SQL supports the creation of assertions, which are constraints not associated with anyone table.

As an example, suppose that we wish to enforce the constraint that the number of boats plus the number of sailors should be less than 100. (This condition Illight be required, say, to qualify as a 'small' sailing club.) We could try the following table constraint:

```
CREATE TABLE Sailors ( sid     INTEGER,
                       sname   CHAR(10),
```

```
        rating  INTEGER,
        age     REAL,
        PRIMARY KEY (sid),
        CHECK ( rating >= 1 AND rating <= 10)
        CHECK ( ( SELECT COUNT (S.sid) FROM Sailors S )
                + ( SELECT COUNT (B.bid) FROM Boats B )
                < 100 ))
```

This solution suffers from two drawbacks. It is associated with Sailors, although it involves Boats in a completely symmetric way. More important, if the Sailors table is empty, this constraint is defined (as per the semantics of table constraints) to always hold, even if we have more than 100 rows in Boats! We could extend this constraint specification to check that Sailors is nonempty, but this approach becomes cumbersome. The best solution is to create an assertion, as follows:

```
CREATE ASSERTION smallClub
CHECK (( SELECT COUNT (S.sid) FROM Sailors S )
       + ( SELECT COUNT (B.bid) FROM Boats B)
       < 100 )
```

5.8 TRIGGERS AND ACTIVE DATABASES

A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an active database. A trigger description contains three parts:

- Event: A change to the database that activates the trigger.
- Condition: A query or test that is run when the trigger is activated.
- Action: A procedure that is executed when the trigger is activated and its condition is true.

A trigger can be thought of as a 'daemon' that monitors a database, and is executed when the database is modified in a way that matches the *event* specification. An insert, delete, or update statement could activate a trigger, regardless of which user or application invoked the activating statement; users may not even be aware that a trigger was executed as a side effect of their program.

A *condition* in a trigger can be a true/false statement (e.g., all employee salaries are less than $100,000) or a query. A query is interpreted as *true* if the answer

set is nonempty and *false* if the query has no answers. If the condition part evaluates to true, the action associated with the trigger is executed.

A trigger *action* can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute Hew queries, and make changes to the database. In fact, an action can even execute a series of data-definition commands (e.g., create new tables, change authorizations) and transaction-oriented commands (e.g., commit) or call host-language procedures.

An important issue is when the action part of a trigger executes in relation to the statement that activated the trigger. For example, a statement that inserts records into the Students table may activate a trigger that is used to maintain statistics on how many students younger than 18 are inserted at a time by a typical insert statement. Depending on exactly what the trigger does, we may want its action to execute *before* changes are made to the Students table or *afterwards:* A trigger that initializes a variable used to count the nurnber of qualifying insertions should be executed before, and a trigger that executes once per qualifying inserted record and increments the variable should be executed after each record is inserted (because we may want to examine the values in the new record to determine the action).

5.8.1 Examples of Triggers in SQL

The examples shown in Figure 5.20, written using Oracle Server syntax for defining triggers, illustrate the basic concepts behind triggers. (The SQL:1999 syntax for these triggers is similar; we will see an example using SQL:1999 syntax shortly.) The trigger called *init_count* initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation. The trigger called *incr_count* increments the counter for each inserted tuple that satisfies the condition *age* < 18.

One of the example triggers in Figure 5.20 executes before the aetivating statement, and the other example executes after it. A trigger can also be scheduled to execute *instead of* the activating statement; or in *deferred* fashion, at the end of the transaction containing the activating statement; or in *asynchronous* fashion, as part of a separate transaction.

The example in Figure 5.20 illustrates another point about trigger execution: A user must be able to specify whether a trigger is to be executed once per modified record or once per activating statement. If the action depends on individual changed records, for example, we have to examine the *age* field of the inserted Students record to decide whether to increment the count, the trigger-

```
CREATE TRIGGER iniLeount BEFORE INSERT ON Students      1* Event *1
    DECLARE
        count INTEGER:
    BEGIN                                               1* Action */
        count := 0:
    END

CREATE TRIGGER incLcount AFTER INSERT ON Students       1* Event *1
    WHEN (new.age < 18)  1* Condition; 'new' is just-inserted tuple *1
    FOR EACH ROW
    BEGIN        1* Action; a procedure in Oracle's PL/SQL syntax *1
        count := count + 1;
    END
```

Figure 5.20 Examples Illustrating Triggers

ing event should be defined to occur for each modified record; the FOR EACH ROW clause is used to do this. Such a trigger is called a row-level trigger. On the other hand, the *iniLcount* trigger is executed just once per INSERT statement, regardless of the number of records inserted, because we have omitted the FOR EACH ROW phrase. Such a trigger is called a statement-level trigger.

In Figure 5.20, the keyword new refers to the newly inserted tuple. If an existing tuple were modified, the keywords old and new could be used to refer to the values before and after the modification. SQL:1999 also allows the action part of a trigger to refer to the *set* of changed records, rather than just one changed record at a time. For example, it would be useful to be able to refer to the set of inserted Students records in a trigger that executes once after the INSERT statement; we could count the number of inserted records with *age* < 18 through an SQL query over this set. Such a trigger is shown in Figure 5.21 and is an aJternative to the triggers shown in Figure 5.20.

The definition in Figure 5.21 uses the syntax of SQL:1999, in order to illustrate the similarities and differences with respect to the syntax used in a typical current DBMS. The keyword clause NEW TABLE enables us to give a table name (InsertedTuples) to the set of newly inserted tuples. The FOR EACH STATEMENT clause specifies a statement-level trigger and can be omitted because it is the default. This definition does not have a WHEN clause; if such a clause is included, it follows the FOR EACH STATEMENT clause, just before the action specification.

The trigger is evaluated once for each SQL statement that inserts tuples into Students, and inserts a single tuple into a table that contains statistics on mod-

ifications to database tables. The first two fields of the tuple contain constants (identifying the modified table, Students, and the kind of modifying statement, an INSERT), and the third field is the number of inserted Students tuples with *age* < 18. (The trigger in Figure 5.20 only computes the count; an additional trigger is required to insert the appropriate tuple into the statistics table.)

```
CREATE TRIGGER seLcount AFTER INSERT ON Students        /* Event */
REFERENCING NEW TABLE AS InsertedTuples
FOR EACH STATEMENT
    INSERT                                              /* Action */
        INTO StatisticsTable(ModifiedTable, ModificationType, Count)
        SELECT 'Students', 'Insert', COUNT *
        FROM InsertedTuples I
        WHERE 1.age < 18
```

Figure 5.21 Set-Oriented Trigger

5.9 DESIGNING ACTIVE DATABASES

Triggers offer a powerful mechanism for dealing with changes to a database, but they must be used with caution. The effect of a collection of triggers can be very complex, and maintaining an active database can become very difficult. Often, a judicious use of integrity constraints can replace the use of triggers.

5.9.1 Why Triggers Can Be Hard to Understand

In an active database system, when the DBMS is about to execute a statement that modifies the database, it checks whether some trigger is activated by the statement. If so, the DBMS processes the trigger by evaluating its condition part, and then (if the condition evaluates to true) executing its action part.

If a statement activates more than one trigger, the DBMS typically processes all of them, in senne arbitrary order. An important point is that the execution of the action part of a trigger could in turn activate another trigger. In particular, the execution of the action part of a trigger could again activate the sarne trigger; such triggers are called recursive triggers. The potential for such *chain* activations and the unpredictable order in which a DBMS processes activated triggers can make it difficult to understand the effect of a collection of triggers.

5.9.2 Constraints versus Triggers

A common use of triggers is to maintain database consistency, and in such cases, we should always consider whether using an integrity constraint (e.g., a foreign key constraint) achieves the same goals. The meaning of a constraint is not defined operationally, unlike the effect of a trigger. This property makes a constraint easier to understand, and also gives the DBMS more opportunities to optimize execution. A constraint also prevents the data from being made inconsistent by *any* kind of statement, whereas a trigger is activated by a specific kind of statement (INSERT, DELETE, or UPDATE). Again, this restriction makes a constraint easier to understand.

On the other hand, triggers allow us to maintain database integrity in more flexible ways, as the following examples illustrate.

- Suppose that we have a table called Orders with fields *iternid, quantity, custornerid,* and *unitprice.* When a customer places an order, the first three field values are filled in by the user (in this example, a sales clerk). The fourth field's value can be obtained from a table called Items, but it is important to include it in the Orders table to have a complete record of the order, in case the price of the item is subsequently changed. We can define a trigger to look up this value and include it in the fourth field of a newly inserted record. In addition to reducing the number of fields that the clerk has to type in, this trigger eliminates the possibility of an entry error leading to an inconsistent price in the Orders table.
- Continuing with this example, we may want to perform some additional actions when an order is received. For example, if the purchase is being charged to a credit line issued by the company, we may want to check whether the total cost of the purchase is within the current credit limit. We can use a trigger to do the check; indeed, we can even use a CHECK constraint. Using a trigger, however, allows us to implement more sophisticated policies for dealing with purchases that exceed a credit limit. For instance, we may allow purchases that exceed the limit by no more than 10% if the customer has dealt with the company for at least a year, and add the customer to a table of candidates for credit limit increases.

5.9.3 Other Uses of Triggers

Many potential uses of triggers go beyond integrity maintenance. Triggers can alert users to unusual events (as reflected in updates to the database). For example, we may want to check whether a customer placing an order has made enough purchases in the past month to qualify for an additional discount; if so, the sales clerk must be informed so that he (or she) can tell the customer

and possibly generate additional sales! \Ve can relay this information by using a trigger that checks recent purchases and prints a message if the customer qualifies for the discount.

Triggers can generate a log of events to support auditing and security checks. For example, each time a customer places an order, we can create a record with the customer's ID and current credit limit and insert this record in a customer history table. Subsequent analysis of this table might suggest candidates for an increased credit limit (e.g., customers who have never failed to pay a bill on time and who have come within 10% of their credit limit at least three times in the last month).

As the examples in Section 5.8 illustrate, we can use triggers to gather statistics on table accesses and modifications. Some database systems even use triggers internally as the basis for managing replicas of relations (Section 22.11.1). Our list of potential uses of triggers is not exhaustive; for example, triggers have also been considered for workflow management and enforcing business rules.

5.10 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What are the parts of a basic SQL query? Are the input and result tables of an SQL query sets or multisets? How can you obtain a set of tuples as the result of a query? (Section 5.2)
- What are range variables in SQL? How can you give names to output columns in a query that are defined by arithmetic or string expressions? What support does SQL offer for string pattern matching? (Section 5.2)
- What operations does SQL provide over (multi)sets of tuples, and how would you use these in writing queries? (Section 5.3)
- What are nested queries? What is *correlation* in nested queries? How would you use the operators IN, EXISTS, UNIQUE, ANY, and ALL in writing nested queries? Why are they useful? Illustrate your answer by showing how to write the *division* operator in SQL. (Section 5.4)
- What aggregate operators does SQL support? (Section 5.5)
- What is *grouping*? Is there a counterpart in relational algebra? Explain this feature, and discllss the interaction of the HAVING and WHERE clauses. Mention any restrictions that mllst be satisfied by the fields that appear in the GROUP BY clause. (Section 5.5.1)

- \What are *null* values? Are they supported in the relational model, as described in Chapter 3'1 How do they affect the meaning of queries? Can primary key fields of a table contain *null* values? (Section 5.6)
- What types of SQL constraints can be specified using the query language? Can you express primary key constraints using one of these new kinds of constraints? If so, why does SQL provide for a separate primary key constraint syntax? (Section 5.7)
- What is a *trigger*, and what are its three parts? What are the differences between row-level and statement-level triggers? (Section 5.8)
- \Vhy can triggers be hard to understand? Explain the differences between triggers and integrity constraints, and describe when you would use triggers over integrity constrains and vice versa. What are triggers used for? (Section 5.9)

EXERCISES

Online material is available for all exercises in this chapter on the book's webpage at

`http://www.cs.wisc.edu/~dbbook`

This includes scripts to create tables for each exercise for use with Oracle, IBM DB2, Microsoft SQL Server, and MySQL.

Exercise 5.1 Consider the following relations:

Student(snum: integer, *sname:* string, *major:* string, *level:* string, *age:* integer)
Class(*name:* string, *meets_at:* time, *room:* string, *fid:* integer)
Enrolled(snum: integer, *cname:* string)
Faculty(*fid:* integer, *fnarne:* string, *deptid:* integer)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

Write the following queries in SQL. No duplicates should be printed in any of the ans\vers.

1. Find the nari1es of all Juniors (level = JR) who are enrolled in a class taught by 1. Teach.
2. Find the age of the oldest student who is either a History major or enrolled in a course taught by I. Teach.
3. Find the names of all classes that either meet in room R128 or have five or more students enrolled.
4. Find the Ilames of all students who are enrolled in two classes that meet at the same time.

5. Find the names of faculty members \vho teach in every room in which some class is taught.
6. Find the names of faculty members for \vhorn the combined enrollment of the courses that they teach is less than five.
7. Print the level and the average age of students for that level, for each level.
8. Print the level and the average age of students for that level, for all levels except JR.
9. For each faculty member that has taught classes only in room *R128,* print the faculty member's name and the total number of classes she or he has taught.
10. Find the names of students enrolled in the maximum number of classes.
11. Find the names of students not enrolled in any class.
12. For each age value that appears in Students, find the level value that appears most often. For example, if there are more FR level students aged 18 than SR, JR, or SO students aged 18, you should print the pair (18, FR).

Exercise 5.2 Consider the following schema:

Suppliers(*sid:* integer, *sname:* string, *address:* string)
Parts(pid: integer, *pname:* string, *color:* string)
Catalog(*sid:* integer, *pid:* integer, *cost:* real)

The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in SQL:

1. Find the *pnames* of parts for which there is some supplier.
2. Find the *snames* of suppliers who supply every part.
3. Find the *snames* of suppliers who supply every red part.
4. Find the *pnamcs* of parts supplied by Acme Widget Suppliers and no one else.
5. Find the *sids* of suppliers who charge more for some part than the average cost of that part (averaged over all the suppliers who supply that part).
6. For each part, find the *sname* of the supplier who charges the most for that part.
7. Find the *sids* of suppliers who supply only red parts.
8. Find the *sids* of suppliers who supply a red part anel a green part.
9. Find the *sids* of suppliers who supply a red part or a green part.
10. For every supplier that only supplies green parts, print the name of the supplier and the total number of parts that she supplies.
11. For every supplier that supplies a green part and a reel part, print the name and price of the most expensive part that she supplies.

Exercise 5.3 The following relations keep track of airline flight information:

Flights(.flno: integer, *from:* string, *to:* string, *di8tance:* integer,
departs: time, *arrives:* time, *price:* integer)
Aircraft(*aid:* integer, *aname:* string, *cruisingrange:* integer)
Certified(*eid:* integer, *aid:* integer)
Employees(*eid:* integer, *ename:* string, *salary:* integer)

Note that the Employees relation describes pilots and other kinds of employees as well; every pilot is certified for some aircraft, and only pilots are certified to fly. Write each of the following queries in SQL. (*Additional queries using the same schema are listed in the exercises for Chapter 4.*)

1. Find the names of aircraft such that all pilots certified to operate them earn more than $80,000.
2. For each pilot who is certified for more than three aircraft, find the *eid* and the maximum *cruisingrange* of the aircraft for which she or he is certified.
3. Find the names of pilots whose *salary* is less than the price of the cheapest route from Los Angeles to Honolulu.
4. For all aircraft with *cruisingrange* over 1000 miles, find the name of the aircraft and the average salary of all pilots certified for this aircraft.
5. Find the names of pilots certified for some Boeing aircraft.
6. Find the *aids* of all aircraft that can be used on routes from Los Angeles to Chicago.
7. Identify the routes that can be piloted by every pilot who makes more than $100,000.
8. Print the *enames* of pilots who can operate planes with *cruisingmnge* greater than 3000 miles but are not certified on any Boeing aircraft.
9. A customer wants to travel from Madison to New York with no more than two changes of flight. List the choice of departure times from Madison if the customer wants to arrive in New York by 6 p.m.
10. Compute the difference between the average salary of a pilot and the average salary of all employees (including pilots).
11. Print the name and salary of every nonpilot whose salary is more than the average salary for pilots.
12. Print the names of employees who are certified only on aircrafts with cruising range longer than 1000 miles.
13. Print the names of employees who are certified only on aircrafts with cruising range longer than 1000 miles, but on at least two such aircrafts.
14. Print the names of employees who are certified only on aircrafts with cruising range longer than 1000 miles and who are certified on some Boeing aircraft.

Exercise 5.4 Consider the following relational schema. An employee can work in more than one department; the *pct_time* field of the Works relation shows the percentage of time that a given employee works in a given department.

Emp(eid: integer, *ename:* string, *age:* integer, *salary:* real)
Works(eid: integer, *did:* integer, *pet_time:* integer)
Dept(did.: integer, *budget:* real, *managerid:* integer)

Write the following queries in SQL:

1. Print the names and ages of each employee who works in both the Hardware department and the Software department.
2. For each department with more than 20 full-time-equivalent employees (i.e., where the part-time and full-time employees add up to at least that many full-time employees), print the *did* together with the number of employees that work in that department.

sid	sname	rating	age
18	jones	3	30.0
41	jonah	6	56.0
22	ahab	7	44.0
63	moby	*null*	15.0

Figure 5.22 An Instance of Sailors

3. Print the name of each employee whose salary exceeds the budget of all of the departments that he or she works in.
4. Find the *managerids* of managers who manage only departments with budgets greater than $1 million.
5. Find the *enames* of managers who manage the departments with the largest budgets.
6. If a manager manages more than one department, he or she *controls* the sum of all the budgets for those departments. Find the *managerids* of managers who control more than $5 million.
7. Find the *managerids* of managers who control the largest amounts.
8. Find the *enames* of managers who manage only departments with budgets larger than $1 million, but at least one department with budget less than $5 million.

Exercise 5.5 Consider the instance of the Sailors relation shown in Figure 5.22.

1. Write SQL queries to compute the average rating, using AVG; the sum of the ratings, using SUM; and the number of ratings, using COUNT.
2. If you divide the sum just computed by the count, would the result be the same as the average? How would your answer change if these steps were carried out with respect to the *age* field instead of *rating*?
3. Consider the following query: *Find the names of sailors with a higher rating than all sailors with age* < 21. The following two SQL queries attempt to obtain the answer to this question. Do they both compute the result? If not, explain why. Under what conditions would they compute the same result?

```
SELECT S.sname
FROM   Sailors S
WHERE  NOT EXISTS ( SELECT *
                    FROM   Sailors S2
                    WHERE  S2.age < 21
                           AND S.rating <= S2.rating )
```

```
SELECT *
FROM   Sailors S
WHERE  S.rating > ANY (SELECT S2.rating
                       FROM   Sailors S2
                       WHERE  S2.age < 21
```

4. Consider the instance of Sailors shown in Figure 5.22. Let us define instance S1 of Sailors to consist of the first two tuples, instance S2 to be the last two tuples, and S to be the given instance.

Show the left outer join of S with itself, with the join condition being *sid=sid.*

(b) Show the right outer join of S with itself, with the join condition being *sid=sid.*

(c) Show the full outer join of S with itself, with the join condition being *S'id=sid.*

(d) Show the left outer join of S1 with S2, with the join condition being *sid=sid.*

(e) Show the right outer join of S1 with S2, with the join condition being *sid=sid.*

(f) Show the full outer join of 81 with S2, with the join condition being *sid=sid.*

Exercise 5.6 Answer the following questions:

1. Explain the term *'impedance mismatch* in the context of embedding SQL commands in a host language such as C.
2. How can the value of a host language variable be passed to an embedded SQL command?
3. Explain the WHENEVER command's use in error and exception handling.
4. Explain the need for cursors.
5. Give an example of a situation that calls for the use of embedded SQL; that is, interactive use of SQL commands is not enough, and some host lang;uage capabilities are needed.
6. Write a C program with embedded SQL commands to address your example in the previous answer.
7. Write a C program with embedded SQL commands to find the standard deviation of sailors' ages.
8. Extend the previous program to find all sailors whose age is within one standard deviation of the average age of all sailors.
9. Explain how you would write a C program to compute the transitive closure of a graph, represented as an 8QL relation Edges(*from, to),* using embedded SQL commands. (You need not write the program, just explain the main points to be dealt with.)
10. Explain the following terms with respect to cursors: *updatability, sens,itivity,* and *scrollability.*
11. Define a cursor on the Sailors relation that is updatable, scrollable, and returns answers sorted by *age.* Which fields of Sailors can such a cursor *not* update? Why?
12. Give an example of a situation that calls for dynamic 8QL; that is, even embedded SQL is not sufficient.

Exercise 5.7 Consider the following relational schema and briefly answer the questions that follow:

Emp(*eid:* integer, *ename:* string, *age:* integer, *salary:* real)
\Vorks(*eid:* integer, *did:* integer, *pct_time:* integer)
Dept(*did:* integer, *budget:* real, *managerid:* integer)

1. Define a table constraint on Emp that will ensure that every employee makes at least $10,000.
2. Define a table constraint on Dept that will ensure that all managers have $age > 30$.
3. Define an assertion on Dept that will ensure that all managers have $age > 30$. Compare this assertion with the equivalent table constraint. Explain which is better.

4. Write SQL statements to delete all information about employees whose salaries exceed that of the manager of one or more departments that they work in. Be sure to ensure that all the relevant integrity constraints are satisfied after your updates.

Exercise 5.8 Consider the following relations:

Student(*snum:* integer, *sname:* string, *rnajor:* string, *level:* string, *age:* integer)
Class(narne: string, *meets_at:* time, *roorn:* string, *fid:* integer)
Enrolled(*snum:* integer, *cnarne:* string)
Faculty(*fid:* integer, *fnarne:* string, *deptid:* integer)

The meaning of these relations is straightforward; for example, Enrolled has one record per student-class pair such that the student is enrolled in the class.

1. Write the SQL statements required to create these relations, including appropriate versions of all primary and foreign key integrity constraints.
2. Express each of the following integrity constraints in SQL unless it is implied by the primary and foreign key constraint; if so, explain how it is implied. If the constraint cannot be expressed in SQL, say so. For each constraint, state what operations (inserts, deletes, and updates on specific relations) must be monitored to enforce the constraint.
   (a) Every class has a minimum enrollment of 5 students and a maximum enrollment of 30 students.
   (b) At least one dass meets in each room.
   (c) Every faculty member must teach at least two courses.
   (d) Only faculty in the department with *deptid=33* teach more than three courses.
   (e) Every student must be enrolled in the course called lVlathlOl.
   (f) The room in which the earliest scheduled class (i.e., the class with the smallest *meets_at* value) meets should not be the same as the room in which the latest scheduled class meets.
   (g) Two classes cannot meet in the same room at the same time.
   (h) The department with the most faculty members must have fewer than twice the number of faculty members in the department with the fewest faculty members.
   (i) No department can have more than 10 faculty members.
   (j) A student cannot add more than two courses at a time (i.e., in a single update).
   (k) The number of CS majors must be more than the number of Math majors.
   (l) The number of distinct courses in which CS majors are enrolled is greater than the number of distinct courses in which Math majors are enrolled.
   (rn) The total enrollment in courses taught by faculty in the department with *deptid=33* is greater than the number of ivlath majors.
   (n) There lllUst be at least one CS major if there are any students whatsoever.
   (0) Faculty members from different departments cannot teach in the same room.

Exercise 5.9 Discuss the strengths and weaknesses of the trigger mechanism. Contrast triggers with other integrity constraints supported by SQL.

Exercise **5.10** Consider the following relational schema. An employee can work in more than one department; the *pct_time* field of the \Works relation shows the percentage of time that a given employee works in a given department.

Emp(*eid:* integer, *ename:* string, *age:* integer, *salary:* real)
Works(*eid:* integer, *did:* integer, *pct_time:* integer)
Dept(*did:* integer, *budget:* real, *mana,gerid:* integer)

\Vrite SQL-92 integrity constraints (domain, key, foreign key, or CHECK constraints; or asser·· bons) or SQL:1999 triggers to ensure each of the following requirements, considered independently.

1. Employees must make a minimum salary of $1000.
2. Every manager must be also be an employee.
3. The total percentage of aU appointments for an employee must be under 100%.
4. A manager must always have a higher salary than any employee that he or she manages.
5. Whenever an employee is given a raise, the manager's salary must be increased to be at least as much.
6. Whenever an employee is given a raise, the manager's salary must be increased to be at least as much. Further, whenever an employee is given a raise, the department's budget must be increased to be greater than the sum of salaries of aU employees in the department.

PROJECT-BASED EXERCISE

Exercise **5.11** Identify the subset of SQL queries that are supported in Minibase.

BIBLIOGRAPHIC NOTES

The original version of SQL was developed as the query language for IBM's System R project, and its early development can be traced in [107, 151]. SQL has since become the most widely used relational query language, and its development is now subject to an international standardization process.

A very readable and comprehensive treatment of SQL-92 is presented by Melton and Simon in [524], and the central features of SQL:1999 are covered in [525]. We refer readers to these two books for an authoritative treatment of SQL. A short survey of the SQL:1999 standard is presented in [237]. Date offers an insightful critique of SQL in [202]. Although some of the problems have been addressed in SQL-92 and later revisions, others remain. A formal semantics for a large subset ofSQL queries is presented in [560]. SQL:1999 is the current International Organization for Standardization (ISO) and American National Standards Institute (ANSI) standard. Melton is the editor of the ANSI and ISO SQL:1999 standard, document ANSI/ISO/IEe 9075-:1999. The corresponding ISO document is ISO/lEe 9075-:1999. A successor, planned for 2003, builds on SQL:1999 SQL:2003 is close to ratification (as of June 20(2). Drafts of the SQL:2003 deliberations are available at the following URL:

ftp://sqlstandards.org/SC32/

[774] contains a collection of papers that cover the active database field. [794] includes a good in-depth introduction to active rules, covering smnantics, applications and design issues. [251] discusses SQL extensions for specifying integrity constraint checks through triggers. [123] also discusses a procedural mechanism, called an *alerter*, for monitoring a database. [185] is a recent paper that suggests how triggers might be incorporated into SQL extensions. Influential active database prototypes include Ariel [366], HiPAC [516J, ODE [18], Postgres [722], RDL [690], and Sentinel [36]. [147] compares various architectures for active database systems.

[32] considers conditions under which a collection of active rules has the same behavior, independent of evaluation order. Semantics of active databases is also studied in [285] and [792]. Designing and managing complex rule systems is discussed in [60, 225]. [142] discusses rule management using Chimera, a data model and language for active database systems.

PART II

APPLICATION DEVELOPMENT

6

DATABASE APPLICATION DEVELOPMENT

- How do application programs connect to a DBMS?
- How can applications manipulate data retrieved from a DBMS?
- How can applications modify data in a DBMS?
- What are cursors?
- What is JDBC and how is it used?
- What is SQLJ and how is it used?
- What are stored procedures?
- **Key concepts:** Embedded SQL, Dynamic SQL, cursors; JDBC, connections, drivers, ResultSets, java.sql, SQLJ; stored procedures, SQL/PSM

He profits most who serves best.

------Ivlotto for Rotary International

In Chapter 5, we looked at a wide range of SQL query constructs, treating SQL as an independent language in its own right. A relational DBMS supports an *interactive SQL* interface, and users can directly enter SQL commands. This simple approach is fine as long as the task at hand can be accomplished entirely with SQL cormnands. In practice, we often encounter situations in which we need the greater flexibility of a general-purpose programming language in addition to the data manipulation facilities provided by SQL. For example, we rnay want to integrate a database application with a nice graphical user interface, or we may want to integrate with other existing applications.

Applications that rely on the DBMS to manage data run as separate processes that connect to the DBlvIS to interact with it. Once a connection is established, SQL commands can be used to insert, delete, and modify data. SQL queries can be used to retrieve desired data. but we need to bridge an important difference in how a database system sees data and how an application program in a language like Java or C sees data: The result of a database query is a set (or multiset) or records, hut Java has no set or multiset data type. This mismatch is resolved through additional SQL constructs that allow applications to obtain a handle on a collection and iterate over the records one at a time.

We introduce Embedded SQL, Dynamic SQL, and cursors in Section 6.1. Embedded SQL allows us to access data using static SQL queries in application code (Section 6.1.1); with Dynamic SQL, we can create the queries at run-time (Section 6.1.3). Cursors bridge the gap between set-valued query answers and programming languages that do not support set-values (Section 6.1.2).

The emergence of Java as a popular application development language, especially for Internet applications, has made accessing a DBMS from Java code a particularly important topic. Section 6.2 covers JDBC, a prograruming interface that allows us to execute SQL queries from a Java program and use the results in the Java program. JDBC provides greater portability than Embedded SQL or Dynamic SQL, and offers the ability to connect to several DBMSs without recompiling the code. Section 6.4 covers SQLJ, which does the same for static SQL queries, but is easier to program in than Java, with JDBC.

Often, it is useful to execute application code at the database server, rather than just retrieve data and execute application logic in a separate process. Section 6.5 covers stored procedures, which enable application logic to be stored and executed at the database server. We conclude the chapter by discussing our B&N case study in Section 6.6.

While writing database applications, we must also keep in mind that typically many application programs run concurrently. The transaction concept, introduced in Chapter 1, is used to encapsulate the effects of an application on the database. An application can select certain transaction properties through SQL cormnands to control the degree to which it is exposed to the changes of other concurrently running applications. We touch on the transaction concept at many points in this chapter, and, in particular, cover transaction-related aspects of JDBC. A full discussion of transaction properties and SQL's support for transactions is deferred until Chapter 16.

Exainples that appear in this chapter are available online at

http://www.cs.wisc.edu/-dbbook

6.1 ACCESSING DATABASES FROM APPLICATIONS

In this section, we cover how SQL commands can be executed from within a program in a host language such as C or Java. The use of SQL commands within a host language program is called Embedded SQL. Details of Embedded SQL also depend on the host language. Although similar capabilities are supported for a variety of host languages, the syntax sometimes varies.

We first cover the basics of Embedded SQL with static SQL queries in Section 6.1.1. We then introduce cursors in Section 6.1.2. We discuss Dynamic SQL, which allows us to construct SQL queries at runtime (and execute them) in Section 6.1.3.

6.1.1 Embedded SQL

Conceptually, embedding SQL commands in a host language program is straightforward. SQL statements (i.e., not declarations) can be used wherever a statement in the host language is allowed (with a few restrictions). SQL statements must be clearly marked so that a preprocessor can deal with them before invoking the compiler for the host language. Also, any host language variables used to pass arguments into an SQL command must be declared in SQL. In particular, some special host language variables *must* be declared in SQL (so that, for example, any error conditions arising during SQL execution can be communicated back to the main application program in the host language).

There are, however, two complications to bear in mind. First, the data types recognized by SQL may not be recognized by the host language and vice versa. This mismatch is typically addressed by casting data values appropriately before passing them to or frorn SQL commands. (SQL, like other programming languages, provides an operator to cast values of alle type into values of another type.) The second complication has to do with SQL being set-oriented, and is addressed using cursors (see Section 6.1.2. Commands operate on and produce tables, which are sets

In our discussion of Embedded SQL, we assume that the host language is C for concreteness. because minor differcnces exist in how SQL statements are embedded in differcnt host languages.

Declaring Variables and Exceptions

SQL statements can refer to variables defined in the host program. Such host-language variables must be prefixed by a colon (:) in SQL statements and be declared between the commands EXEC SQL BEGIN DECLARE SECTION and EXEC

SQL END DECLARE SECTION. The declarations are similar to how they would look in a C program and, as usual in C. are separated by semicolons. For example. we can declare variables *c_sname*, *c_sid*, *c_rating*, and *c_age* (with the initial *c* used as a naming convention to emphasize that these are host language variables) as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

The first question that arises is which SQL types correspond to the various C types, since we have just declared a collection of C variables whose values are intended to be read (and possibly set) in an SQL run-time environment when an SQL statement that refers to them is executed. The SQL-92 standard defines such a correspondence between the host language types and SQL types for a number of host languages. In our example, *c_snamc* has the type CHARACTER(20) when referred to in an SQL statement, *c_sid* has the type INTEGER, *c_rating* has the type SMALLINT, and *c_age* has the type REAL.

We also need some way for SQL to report what went wrong if an error condition arises when executing an SQL statement. The SQL-92 standard recognizes two special variables for reporting errors, SQLCODE and SQLSTATE. SQLCODE is the older of the two and is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes. SQLSTATE, introduced in the SQL-92 standard for the first time, associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reported. One of these two variables *must* be declared. The appropriate C type for SQLCODE is long and the appropriate C type for SQLSTATE is char[6J , that is, a character string five characters long. (Recall the null-terminator in C strings.) In this chapter, we assume that SQLSTATE is declared.

Embedding SQL Statements

All SQL staternents embedded within a host program must be clearly marked, with the details dependent on the host language; in C, SQL statements must be prefixed by EXEC SQL. An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.

As a simple example, the following Embedded SQL statement inserts a row, whose column values are based on the values of the host language variables contained in it, into the Sailors relation:

```
EXEC SQL
INSERT INTO Sailors VALUES (:c_sname, :c_sid, :c_rating, :c_age);
```

Observe that a semicolon terminates the command, as per the convention for terminating statements in C.

The SQLSTATE variable should be checked for errors and exceptions after each Embedded SQL statement. SQL provides the WHENEVER command to simplify this tedious task:

```
EXEC SQL WHENEVER [SQLERROR | NOT FOUND] [ CONTINUE | GOTO st'mt ]
```

The intent is that the value of SQLSTATE should be checked after each Embedded SQL statement is executed. If SQLERROR is specified and the value of SQLSTATE indicates an exception, control is transferred to *stmt*, which is presumably responsible for error and exception handling. Control is also transferred to *stmt* if NOT FOUND is specified and the value of SQLSTATE is 02000, which denotes NO DATA.

6.1.2 Cursors

A major problem in embedding SQL statements in a host language like C is that an *impedance mismatch* occurs because SQL operates on *sets* of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation.

This mechanism is called a cursor. We can declare a cursor on any relation or on any SQL query (because every query returns a set of rows). Once a cursor is declared, we can open it (which positions the cursor just before the first row); fetch the next row; move the cursor (to the next row, to the row after the next *n*, to the first row, or to the previous row, etc., by specifying additional parameters for the FETCH command); or close the cursor. Thus, a cursor essentially allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

Basic Cursor Definition and Usage

Cursors enable us to examine, in the host language program, a collection of rows computed by an Embedded SQL statement:

- \Ve usually need to open a cursor if the embedded statement is a SELECT (i.e.) a query). However, we can avoid opening a cursor if the answer contains a single row, as we see shortly.
- INSERT, DELETE, and UPDATE staternents typically require no cursor, although some variants of DELETE and UPDATE use a cursor.

As an example, we can find the name and age of a sailor, specified by assigning a value to the host variable *c_sid*, declared earlier, as follows:

```
EXEC SQL SELECT S.sname, S.age
         INTO    :c_sname, :c_age
         FROM    Sailors S
         WHERE   S.sid = :c_sid;
```

The INTO clause allows us to assign the columns of the single answer row to the host variables *c_sname* and *c_age*. Therefore, we do not need a cursor to embed this query in a host language program. But what about the following query, which computes the names and ages of all sailors with a rating greater than the current value of the host variable *c_minrating*?

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating
```

This query returns a collection of rows, not just one row. 'When executed interactively, the answers are printed on the screen. If we embed this query in a C program by prefixing the cOlnmand with EXEC SQL, how can the answers be bound to host language variables? The INTO clause is inadequate because we must deal with several rows. The solution is to use a cursor:

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating;
```

This code can be included in a C program, and once it is executed, the cursor *sinfo* is defined. Subsequently, we can open the cursor:

```
OPEN sinfo:
```

The value of *c_minrating* in the SQL query associated with the cursor is the value of this variable when we open the cursor. (The cursor declaration is processed at compile-time, and the OPEN command is executed at run-time.)

A cursor can be thought of as 'pointing' to a row in the collection of answers to the query associated with it. When a cursor is opened, it is positioned just before the first row. We can use the FETCH command to read the first row of cursor *sinfo* into host language variables:

```
FETCH sinfo INTO :c_sname, :c_age;
```

When the FETCH statement is executed, the cursor is positioned to point at the next row (which is the first row in the table when FETCH is executed for the first time after opening the cursor) and the column values in the row are copied into the corresponding host variables. By repeatedly executing this FETCH statement (say, in a while-loop in the C program), we can read all the rows computed by the query, one row at a time. Additional parameters to the FETCH command allow us to position a cursor in very flexible ways, but we do not discuss them.

How do we know when we have looked at all the rows associated with the cursor? By looking at the special variables SQLCODE or SQLSTATE, of course. SQLSTATE, for example, is set to the value 02000, which denotes NO DATA, to indicate that there are no more rows if the FETCH statement positions the cursor after the last row.

When we are done with a cursor, we can close it:

```
CLOSE sinfo;
```

It can be opened again if needed, and the value of : *c_minrating* in the SQL query associated with the cursor would be the value of the host variable *c_minrating* at that time.

Properties of Cursors

The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR
        [WITH HOLD]
        FOR some query
        [ ORDER BY order-item-list ]
        [ FOR READ ONLY | FOR UPDATE ]
```

A cursor can be declared to be a read-only cursor (FOR READ ONLY) or, if it is a cursor on a base relation or an updatable view, to be an updatable cursor (FOR UPDATE). If it is IIpdatable, simple variants of the UPDATE and

DELETE commands allow us to update or delete the row on which the cursor is positioned. For example, if *sinfa* is an updatable cursor and open, we can execute the following statement:

```
UPDATE  Sailors S
SET     S.rating = S.rating - 1
WHERE   CURRENT of sinfo;
```

This Embedded SQL statement modifies the *rating* value of the row currently pointed to by cursor *sinfa;* similarly, we can delete this row by executing the next statement:

```
DELETE  Sailors S
WHERE   CURRENT of sinfo;
```

A cursor is updatable by default unless it is a scrollable or insensitive cursor (see below), in which case it is read-only by default.

If the keyword SCROLL is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; otherwise, only the basic FETCH command, which retrieves the next row, is allowed.

If the keyword INSENSITIVE is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows. Otherwise, and by default, other actions of some transaction could modify these rows, creating unpredictable behavior. For example, while we are fetching rows using the *sinfa* cursor, we might modify *rating* values in Sailor rows by concurrently executing the command:

```
UPDATE  Sailors S
SET     S.rating = S.rating -
```

Consider a Sailor row such that (1) it has not yet been fetched, and (2) its original *rating* value would have met the condition in the WHERE clause of the query associated with *sinfa,* but the new *rating* value does not. Do we fetch such a Sailor row? If INSENSITIVE is specified, the behavior is as if all answers were computed and stored when *sinfo* was opened; thus, the update command has no effect on the rows fetched by *sinfa* if it is executed after *sinfo* is opened. If INSENSITIVE is not specified, the behavior is implementation dependent in this situation.

A holdable cursor is specified using the WITH HOLD clause, and is not closed when the transaction is conunitted. The motivation for this cornes from long

transactions in which we access (and possibly change) a large number of rows of a table. If the transaction is aborted for any reason, the system potentially has to redo a lot of work when the transaction is restarted. Even if the transaction is not aborted, its locks are held for a long time and reduce the concurrency of the system. The alternative is to break the transaction into several smaller transactions, but remembering our position in the table between transactions (and other similar details) is complicated and error-prone. Allowing the application program to commit the transaction it initiated, while retaining its handle on the active table (i.e., the cursor) solves this problem: The application can commit its transaction and start a new transaction and thereby save the changes it has made thus far.

Finally, in what order do FETCH commands retrieve rows? In general this order is unspecified, but the optional ORDER BY clause can be used to specify a sort order. Note that columns mentioned in the ORDER BY clause cannot be updated through the cursor!

The order-item-list is a list of order-items; an order-item is a column name, optionally followed by one of the keywords ASC or DESC. Every column mentioned in the ORDER BY clause must also appear in the select-list of the query associated with the cursor; otherwise it is not clear what columns we should sort on. The keywords ASC or DESC that follow a column control whether the result should be sorted-with respect to that column-in ascending or descending order; the default is ASC. This clause is applied as the last step in evaluating the query.

Consider the query discussed in Section 5.5.1, and the answer shown in Figure 5.13. Suppose that a cursor is opened on this query, with the clause:

```
ORDER BY minage ASC, rating DESC
```

The answer is sorted first in ascending order by *minage,* and if several rows have the same *minage* value, these rows are sorted further in descending order by *rating.* The cursor would fetch the rows in the order shown in Figure 6.1.

rating	*minage*
8	25.5
3	25.5
7	35.0

Figure 6.1 Order in which Tuples Are Fetched

6.1.3 Dynamic SQL

Consider an application such as a spreadsheet or a graphical front-end that needs to access data from a DBMS. Such an application must accept commands from a user and, based on what the user needs, generate appropriate SQL statements to retrieve the necessary data. In such situations, we may not be able to predict in advance just what SQL statements need to be executed, even though there is (presumably) some algorithm by which the application can construct the necessary SQL statements once a user's command is issued.

SQL provides some facilities to deal with such situations; these are referred to as Dynamic SQL. We illustrate the two main commands, PREPARE and EXECUTE, through a simple example:

```
char c_sqlstring[] = {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

The first statement declares the C variable *c_sqlstring* and initializes its value to the string representation of an SQL command. The second statement results in this string being parsed and compiled as an SQL command, with the resulting executable bound to the SQL variable *readytogo*. (Since *readytogo* is an SQL variable, just like a cursor name, it is not prefixed by a colon.) The third statement executes the command.

Many situations require the use of Dynamic SQL. However, note that the preparation of a Dynamic SQL command occurs at run-time and is run-time overhead. Interactive and Embedded SQL commands can be prepared once at compile-time and then re-executecl as often as desired. Consequently you should limit the use of Dynamic SQL to situations in which it is essential.

There are many more things to know about Dynamic SQL---how we can pass parameters from the host language program to the SQL statement being pre-parcel, for example--but we do not discuss it further.

6.2 AN INTRODUCTION TO JDBC

Embedded SQL enables the integration of SQL with a general-purpose programming language. As described in Section 6.1.1, a DBMS-specific preprocessor transforms the Embedded SQL statements into function calls in the host language. The details of this translation vary across DBMSs, and therefore even though the source code can be cOlnpiled to work with different DBMSs, the final executable works only with one specific DBMS.

ODBC and JDBC, short for Open DataBase Connectivity and Java DataBase Connectivity, also enable the integration of SQL with a general-purpose programming language. Both ODBC and JDBC expose database capabilities in a standardized way to the application programmer through an application programming interface (API). In contrast to Embedded SQL, ODBC and JDBC allow a single executable to access different DBMSs *without recompilation.* Thus, while Embedded SQL is DBMS-independent only at the source code level, applications using ODBC or JDBC are DBMS-independent at the source code level and at the level of the executable. In addition, using ODBC or JDBC, an application can access not just one DBMS but several different ones simultaneously.

ODBC and JDBC achieve portability at the level of the executable by introducing an extra level of indirection. All direct interaction with a specific DBMS happens through a DBMS-specific driver. A driver is a software program that translates the ODBC or JDBC calls into DBMS-specific calls. Drivers are loaded dynamically on demand since the DBMSs the application is going to access are known only at run-time. Available drivers are registered with a driver manager.

One interesting point to note is that a driver does not necessarily need to interact with a DBMS that understands SQL. It is sufficient that the driver translates the SQL commands from the application into equivalent commands that the DBMS understands. Therefore, in the remainder of this section, we refer to a data storage subsystem with which a driver interacts as a data source.

An application that interacts with a data source through ODBC or JDBC selects a data source, dynamically loads the corresponding driver, and establishes a connection with the data source. There is no limit on the number of open connections, and an application can have several open connections to different data sources. Each connection has transaction semantics; that is, changes from one connection are visible to other connections only after the connection has committed its changes. While a connection is opcn, transactions are executed by submitting SQL statements, retrieving results, processing errors, and finally committing or rolling back. The application disconnects from the data source to terminate the interaction.

In the remainder of this chapter, we concentrate on JDBC.

JDBC Drivers: The most up-to-date source of .IDBC drivers is the Sun JDBC Driver page at
http://industry.java.sun.com/products/jdbc/drivers
JDBC drivers are available for all major database sytems.

6.2.1 Architecture

The architecture of JDBC has four main components: the *application,* the *driver manager,* several data source specific *drivers,* and the corresponding *data SOUTces.*

The *application* initiates and terminates the connection with a data source. It sets transaction boundaries, submits SQL statements, and retrieves the results-----all through a well-defined interface as specified by the JDBC API. The primary goal of the *driver manager* is to load JDBC drivers and pass JDBC function calls from the application to the correct driver. The driver manager also handles JDBC initialization and information calls from the applications and can log all function calls. In addition, the driver manager performs·some rudimentary error checking. The *driver* establishes the connection with the data source. In addition to submitting requests and returning request results, the driver translates data, error formats, and error codes from a form that is specific to the data source into the JDBC standard. The *data source* processes commands from the driver and returns the results.

Depending on the relative location of the data source and the application, several architectural scenarios are possible. Drivers in JDBC are cla.ssified into four types depending on the architectural relationship between the application and the data source:

- Type I Bridges: This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS. An example is a JDBC-ODBC bridge; an application can use JDBC calls to access an ODBC compliant data source. The application loads only one driver, the bridge. Bridges have the advantage that it is easy to piggy-back the applica.tion onto an existing installation, and no new drivers have to be installed. But using bridges has several drawbacks. The increased number of layers between data source and application affects performance. In addition, the user is limited to the functionality that the ODBC driver supports.
- Type II Direct Translation to the Native API via Non-Java Driver: This type of driver translates JDBC function calls directly into method invocations of the API of one specific data source. The driver is

usually written using a combination of C++ and Java; it is dynamically linked and specific to the data source. This architecture performs significantly better than a JDBC-ODBC bridge. One disadvantage is that the database driver that implements the API needs to be installed on each computer that runs the application.

- Type III—Network Bridges: The driver talks over a network to a middleware server that translates the JDBC requests into DBMS-specific method invocations. In this case, the driver on the client site (Le., the network bridge) is not DBMS-specific. The JDBC driver loaded by the application can be quite small, as the only functionality it needs to implement is sending of SQL statements to the middleware server. The middleware server can then use a Type II JDBC driver to connect to the data source.
- Type IV-Direct Translation to the Native API via Java Driver: Instead of calling the DBMS API directly, the driver communicates with the DBMS through Java sockets. In this case, the driver on the client side is written in Java, but it is DBMS-specific. It translates JDBC calls into the native API of the database system. This solution does not require an intermediate layer, and since the implementation is all Java, its performance is usually quite good.

6.3 JDBC CLASSES AND INTERFACES

JDBC is a collection of Java classes and interfaces that enables database access from prograrl1s written in the Java language. It contains methods for connecting to a remote data source, executing SQL statements, examining sets of results from SQL statements, transaction management, and exception handling. The classes and interfaces are part of the `java.sql` package. Thus, all code fragments in the remainder of this section should include the statement `import java.sql.*` at the beginning of the code; we omit this statement in the remainder of this section. JDBC 2.0 also includes the `javax.sql` package, the JDBC Optional Package. The package `javax.sql` adds, among other things, the capability of connection pooling and the `RowSet` interface. We discuss connection pooling in Section 6.3.2, and the `ResultSet` interface in Section 6.3.4.

We now illustrate the individual steps that are required to submit a database query to a data source and to retrieve the results.

6.3.1 JDBC Driver Management

In JDBC, data source drivers are managed by the Drivermanager class, which maintains a list of all currently loaded drivers. The Drivermanager class has

methods registerDriver, deregisterDriver, and getDrivers to enable dynamic addition and deletion of drivers.

The first step in connecting to a data source is to load the corresponding JDBC driver. This is accomplished by using the Java mechanism for dynamically loading classes. The static method forName in the Class class returns the Java class as specified in the argument string and executes its static constructor. The static constructor of the dynamically loaded class loads an instance of the Driver class, and this Driver object registers itself with the DriverManager class.

The following Java example code explicitly loads a JDBC driver:

```
Class.forName("oracle/jdbc.driver.OracleDriver");
```

There are two other ways ofregistering a driver. We can include the driver with -Djdbc. drivers=oracle/jdbc. driver at the command line when we start the Java application. Alternatively, we can explicitly instantiate a driver, but this method is used only rarely, as the name of the driver has to be specified in the application code, and thus the application becomes sensitive to changes at the driver level.

After registering the driver, we connect to the data source.

6.3.2 Connections

A session with a data source is started through creation of a Connection object; A connection identifies a logical session with a data source; multiple connections within the same Java program can refer to different data sources or the same data source. Connections are specified through a **JDBC URL**, a URL that uses the jdbc protocol. Such a URL has the form

```
jdbc:<subprotocol>:<otherParameters>
```

The code example shown in Figure 6.2 establishes a connection to an Oracle database assuming that the strings userId and password are set to valid values.

In JDBC, connections can have different properties. For example, a connection can specify the granularity of transactions. If autocommit is set for a connection, then each SQL statement is considered to be its own transaction. If autocommit is off, then a series of statements that compose a transaction can be committed using the commit 0 method of the Connection class, or aborted using the rollbackO method. The Connection class has methods to set the

```
String uri = "jdbc:oracle:www.bookstore.com:3083"
Connection connection;
try {
    Connection connection =
        DriverManager.getConnection(urI,userId,password);
}
catch(SQLException excpt) {
    System.out.println(excpt.getMessageO);
    return;
}
```

Figure 6.2 Establishing a Connection with JDBC

JDBC Connections: Remember to close connections to data sources and return shared connections to the connection pool. Database systems have a limited number of resources available for connections, and orphan connections can often only be detected through time-outs-and while the database system is waiting for the connection to time-out, the resources used by the orphan connection are wasted.

autocommit mode (Connection. setAutoCommit) and to retrieve the current autocommit mode (getAutoCommit). The following methods are part of the Connection interface and permit setting and getting other properties:

- public int getTransactionIsolation() throws SQLExceptionand public void setTransactionlsolation(int 1) throws SQLException. These two functions get and set the current level of isolation for transactions handled in the current connection. All five SQL levels of isolation (see Section 16.6 for a full discussion) are possible, and argument *1* can be set as follows:
  - TRANSACTIONJNONE
  - TRANSACTIONJREAD.UNCOMMITTED
  - TRANSACTIONJREAD.COMMITTED
  - TRANSACTIONJREPEATABLEJREAD
  - TRANSACTION.BERIALIZABLE
- public boolean getReadOnlyO throws SQLException and public void setReadOnly(boolean readOnly) throws SQLException. These two functions allow the user to specify whether the transactions executecl through this connection are rcad only.

- public boolean isClosed() throws SQLException.
  Checks whether the current connection has already been closed.
- setAutoCommit and getAutoCommit.
  We already discussed these two functions.

Establishing a connection to a data source is a costly operation since it involves several steps, such as establishing a network connection to the data source, authentication, and allocation of resources such as memory. In case an application establishes many different connections from different parties (such as a Web server), connections are often **pooled** to avoid this overhead. A **connection pool** is a set of established connections to a data source. Whenever a new connection is needed, one of the connections from the pool is used, instead of creating a new connection to the data source.

Connection pooling can be handled either by specialized code in the application, or the optional javax.sql package, which provides functionality for connection pooling and allows us to set different parameters, such as the capacity of the pool, and shrinkage and growth rates. Most application servers (see Section 7.7.2) implement the javax.sql package or a proprietary variant.

6.3.3 Executing SQL Statements

We now discuss how to create and execute SQL statements using JDBC. In the JDBC code examples in this section, we assume that we have a Connection object named con. JDBC supports three different ways of executing statements: Statement, PreparedStatement, and CallableStatement. The Statement class is the base class for the other two statment classes. It allows us to query the data source with any static or dynamically generated SQL query. We cover the PreparedStatement class here and the CallableStatement class in Section 6.5, when we discuss stored procedures.

The PreparedStatement class dynamicaJly generates precompiled SQL statements that can be used several times; these SQL statements can have parameters, but their structure is fixed when the PreparedStatement object (representing the SQL statement) is created.

Consider the sample code using a PreparedStatment object shown in Figure 6.3. The SQL query specifies the query string, but uses '?' for the values of the parameters, which are set later using methods setString, setFloat, and setInt. The "1' placeholders can be used anywhere in SQL statements where they can be replaced with a value. Examples of places where they can appear include the WHERE clause (e.g., 'WHERE author=?'), or in SQL UPDATE and INSERT statements, as in Figure 6.3. The method setString is one way

```
// initial quantity is always zero
String sql = "INSERT INTO Books VALUES(?, 7, ?, ?, 0, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);

// now instantiate the parameters with values
// assume that isbn, title, etc. are Java variables that
// contain the values to be inserted
pstmt.clearParameters();
pstmt.setString(l, isbn);
pstmt.setString(2, title);
pstmt.setString(3, author);
pstmt.setFloat(5, price);
pstmt.setInt(6, year);

int numRows = pstmt.executeUpdate();
```

Figure 6.3 SQL Update Using a PreparedStatement Object

to set a parameter value; analogous methods are available for int, float, and date. It is good style to always use clearParameters 0 before setting parameter values in order to remove any old data.

There are different ways of submitting the query string to the data source. In the example, we used the executeUpdate command, which is used if we know that the SQL statement does not return any records (SQL UPDATE, INSERT, ALTER, and DELETE statements). The executeUpdate method returns an integer indicating the number of rows the SQL statement modified; it returns 0 for successful execution without modifying any rows.

The executeQuery method is used if the SQL statement returns data, such as in a regular SELECT query. JDBC has its own cursor mechanism in the form of a ResultSet object, which we discuss next. The execute method is more general than executeQuery and executeUpdate; the references at the end of the chapter provide pointers with more details.

6.3.4 ResultSets

As discussed in the previous section, the statement executeQuery returns a ResultSet object, which is similar to a cursor. ResultSet cursors in JDBC 2.0 are very powerful; they allow forward and reverse scrolling and in-place editing and insertions.

In its most basic form, the ResultSet object allows us to read one row of the output of the query at a time. Initially, the ResultSet is positioned before the first row, and we have to retrieve the first row with an explicit call to the next() method. The next method returns false if there are no more rows in the query answer, and true other\vise. The code fragment shown in Figure 6.4 illustrates the basic usage of a ResultSet object.

```
ResultSet rs=stmt.executeQuery(sqlQuery);
// rs is now a cursor
// first call to rs.nextO moves to the first record
// rs.nextO moves to the next row
String sqlQuery;
ResultSet rs = stmt.executeQuery(sqlQuery)
while (rs.next()) {
    // process the data
}
```

Figure 6.4 Using a ResultSet Object

While next() allows us to retrieve the logically next row in the query answer, we can move about in the query answer in other ways too:

- previous() moves back one row.
- absolute(int num) moves to the row with the specified number.
- relative(int num) moves forward or backward (if num is negative) relative to the current position. relative(-1) has the same effect as previous.
- first() moves to the first row, and last() moves to the last row.

Matching Java and SQL Data Types

In considering the interaction of an application with a data source, the issues we encountered in the context of Embedded SQL (e.g., passing information between the application and the data source through shared variables) arise again. To deal with such issues, JDBC provides special data types and specifies their relationship to corresponding SQL data types. Figure 6.5 shows the accessor methods in a ResultSet object for the most common SQL datatypes. With these accessor methods, we can retrieve values from the current row of the query result referenced by the ResultSet object. There are two forms for each accessor method: One method retrieves values by column index, starting at one, and the other retrieves values by column name. The following example shows how to access fields of the current ResultSet row using accesssor methods.

SQL Type	Java class	ResultSet get method
BIT	Boolean	getBooleanO
CHAR	String	getStringO
VARCHAR	String	getStringO
DOUBLE	Double	getDoubleO
FLOAT	Double	getDoubleO
INTEGER	Integer	getIntO
REAL	Double	getFloatO
DATE	java.sql.Date	getDateO
TIME	java.sql.Time	getTimeO
TIMESTAMP	java.sql.TimeStamp	getTimestamp()

Figure 6.5 Reading SQL Datatypes from a ResultSet Object

```
ResultSet rs=stmt.executeQuery(sqIQuery);
String sqlQuerYi
ResultSet rs = stmt.executeQuery(sqIQuery)
while (rs.nextO) {
    isbn = rs.getString(l);
    title = rs.getString("TITLE");
    // process isbn and title
}
```

6.3.5 Exceptions and Warnings

Similar to the SQLSTATE variable, most of the methods in java. sql can throw an exception of the type SQLException if an error occurs. The information includes SQLState, a string that describes the error (e.g., whether the statement contained an SQL syntax error). In addition to the standard getMessage 0 method inherited from Throwable, SQLException has two additional methods that provide further information, and a method to get (or chain) additional exceptions:

- public String getSQLState 0 returns an SQLState identifier based on the SQL:1999 specification, as discussed in Section 6.1.1.
- public int getErrorCode () retrieves a vendor-specific error code.
- public SQLException getNextExceptionO gets the next exception in a chain of exceptions associated with the current SQLException object.

An SQL\¥arning is a subclass of SQLException. Warnings are not as severe as errors and the program can usually proceed without special handling of warnings. \Varnings are not thrown like other exceptions, and they are not caught as

part of the try"-catch block around a java.sql statement. We Heed to specifically test whether warnings exist. Connection, Statement, and ResultSet objects all have a getWarnings 0 method with which we can retrieve SQL warnings if they exist. Duplicate retrieval of warnings can be avoided through clearWarnings O. Statement objects clear warnings automatically on execution of the next statement; ResultSet objects clear warnings every time a new tuple is accessed.

Typical code for obtaining SQLWarnings looks similar to the code shown in Figure 6.6.

```
try {
    stmt = con.createStatement();
    warning = con.getWarnings();
    while( warning != null) {
        // handleSQLWarnings          //code to process warning
        warning = warning.getNextWarningO;      //get next warning
    }
    con.clear\Varnings();

    stmt.executeUpdate( queryString );
    warning = stmt.getWarnings();
    while( warning != null) {
        // handleSQLWarnings          //code to process warning
        warning = warning.getNextWarningO;      //get next warning
    }
} // end try
catch ( SQLException SQLe) {
    // code to handle exception
} // end catch
```

Figure 6.6 Processing JDBC Warnings and Exceptions

6.3.6 Examining Database Metadata

We can use the DatabaseMetaData object to obtain information about the database system itself, as well as information frorn the database catalog. For example, the following code fragment shows how to obtain the name and driver version of the JDBC driver:

```
DatabaseMetaData md = con.getMetaD<Lta():

System.out.println("Driver Information:");
```

```
System.out.println("Name:" + md.getDriverNameO
            + "; version:" + mcl.getDriverVersion());
```

The DatabaseMetaData object has many more methods (in JDBC 2.0, exactly 134); we list some methods here:

- public ResultSet getCatalogs0 throws SqLException. This function returns a ResultSet that can be used to iterate over all the catalog relations. The functions getIndexInfo0 and getTables0 work analogously.
- pUblic int getMaxConnections0 throws SqLException. This function returns the ma.ximum number of connections possible.

We will conclude our discussion of JDBC with an example code fragment that examines all database metadata shown in Figure 6.7.

```
DatabaseMetaData dmd = con.getMetaDataO;
ResultSet tablesRS = dmd.getTables(null,null,null,null);
string tableName;

while(tablesRS.next()) {
    tableNarne = tablesRS.getString("TABLE_NAME");

    // print out the attributes of this table
    System.out.println("The attributes of table"
            + tableName + " are:");
    ResultSet columnsRS = dmd.getColums(null,null,tableName, null);
    while (columnsRS.next()) {
        System.out.print(columnsRS.getString("COLUMN_NAME")
                +" ");
    }

    // print out the primary keys of this table
    System.out.println("The keys of table" + tableName + " are:");
    ResultSet keysRS = dmd.getPrimaryKeys(null,null,tableName);
    while (keysRS.next()) {
        'System.out.print(keysRS.getStringC'COLUMN_NAME") +" ");
    }
}
```

Figure 6.7 Obtaining Infon-nation about a Data Source

6.4 SQLJ

SQLJ (short for 'SQL-Java') was developed by the SQLJ Group, a group of database vendors and Sun. SQLJ was developed to complement the dynamic way of creating queries in JDBC with a static model. It is therefore very close to Embedded SQL. Unlike JDBC, having semi-static SQL queries allows the compiler to perform SQL syntax checks, strong type checks of the compatibility of the host variables with the respective SQL attributes, and consistency of the query with the database schema-tables, attributes, views, and stored procedures—all at compilation time. For example, in both SQLJ and Embedded SQL, variables in the host language always are bound statically to the same arguments, whereas in JDBC, we need separate statements to bind each variable to an argument and to retrieve the result. For example, the following SQLJ statement binds host language variables title, price, and author to the return values of the cursor books.

```
#sql books = {
    SELECT title, price INTO :title, :price
    FROM Books WHERE author = :author
};
```

In JDBC, we can dynamically decide which host language variables will hold the query result. In the following example, we read the title of the book into variable ftitle if the book was written by Feynman, and into variable otitle otherwise:

```
// assume we have a ResultSet cursor rs
author = rs.getString(3);

if (author=="Feynman") {
    ftitle = rs.getString(2):
}
else {
    otitle = rs.getString(2);
}
```

When writing SQLJ applications, we just write regular Java code and embed SQL statements according to a set of rules. SQLJ applications are pre-processed through an SQLJ translation program that replaces the embedded SQLJ code with calls to an SQLJ Java library. The modified program code can then be compiled by any Java compiler. Usually the SQLJ Java library makes calls to a JDBC driver, which handles the connection to the database system.

An important philosophical difference exists between Embedded SQL and SQLJ and JDBC. Since vendors provide their own proprietary versions of SQL, it is advisable to write SQL queries according to the SQL-92 or SQL:1999 standard. However, when using Embedded SQL, it is tempting to use vendor-specific SQL constructs that offer functionality beyond the SQL-92 or SQL:1999 standards. SQLJ and JDBC force adherence to the standards, and the resulting code is much more portable across different database systems.

In the remainder of this section, we give a short introduction to SQLJ.

6.4.1 Writing SQLJ Code

We will introduce SQLJ by means of examples. Let us start with an SQLJ code fragment that selects records from the Books table that match a given author.

```
String title; Float price; String atithor;
#sql iterator Books (String title, Float price);
Books books;

// the application sets the author
// execute the query and open the cursor
#sql books = {
    SELECT title, price INTO :titIe, :price
    FROM Books WHERE author = :author
};

// retrieve results
while (books.next()) {
    System.out.println(books.titleO + ", " + books.price());
}
books.close();
```

The corresponding JDBC code fragment looks as follows (assuming we also declared price, name, and author:

```
PrcparcdStatcment stmt = connection.prepareStatement(
"SELECT title, price FROM Books WHERE author = ?");

// set the parameter in the query ancl execute it
stmt.setString(1, author);
ResultSet 1'8 = stmt.executeQuery();

// retrieve the results
while (rs.next()) {
```

```
System.out.println(rs.getString(l) + ", " + rs.getFloat(2));
}
```

Comparing the JDBC and SQLJ code, we see that the SQLJ code is much easier to read than the JDBC code. Thus, SQLJ reduces software development and maintenance costs.

Let us consider the individual components of the SQLJ code in more detail. All SQLJ statements have the special prefix #sql. In SQLJ, we retrieve the results of SQL queries with iterator objects, which are basically cursors. An iterator is an instance of an iterator class. Usage of an iterator in SQLJ goes through five steps:

- Declare the Iterator Class: In the preceding code, this happened through the statement
  #sql iterator Books (String title, Float price);
  This statement creates a new Java class that we can use to instantiate objects.
- Instantiate an Iterator Object from the New Iterator Class: We instantiated our iterator in the statement Books books;.
- Initialize the Iterator Using a SQL Statement: In our example, this happens through the statement #sql books = ...
- Iteratively, Read the Rows From the Iterator Object: This step is very similar to reading rows through a ResultSet object in JDBC.
- Close the Iterator Object.

There are two types of iterator classes: named iterators and positional iterators. For named iterators, we specify both the variable type and the name of each column of the iterator. This allows us to retrieve individual columns by name as in our previous example where we could retrieve the title colunm from the Books table using the expression books.titIe(). For positional iterators, we need to specify only the variable type for each column of the iterator. To access the individual columns of the iterator, we use a FETCH ... INTO eonstruct, similar to Embedded SQL. Both iterator types have the same performance; which iterator to use depends on the programmer's taste.

Let us revisit our example. \Ve can make the iterator a positional iterator through the following statement:

```
#sql iterator Books (String, Float);
```

We then retrieve the individual rows from the iterator as follows:

```
while (true) {
    #sql { FETCH :books INTO :title, :price, };
    if (books.endFetch()) {
        break:
    }

    // process the book
}
```

6.5 STORED PROCEDURES

It is often important to execute some parts of the application logic directly in the process space of the database system. Running application logic directly at the database has the advantage that the amount of data that is transferred between the database server and the client issuing the SQL statement can be minimized, while at the same time utilizing the full power of the database server.

When SQL statements are issued from a remote application, the records in the result of the query need to be transferred from the database system back to the application. If we use a cursor to remotely access the results of an SQL statement, the DBMS has resources such as locks and memory tied up while the application is processing the records retrieved through the cursor. In contrast, a stored procedure is a program that is executed through a single SQL statement that can be locally executed and completed within the process space of the database server. The results can be packaged into one big result and returned to the application, or the application logic can be performed directly at the server, without having to transmit the results to the client at all.

Stored procedures are also beneficial for software engineering reasons. Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance easy. In addition, application programmers do not need to know the the database schema if we encapsulate all database access into stored procedures.

Although they are called stored *procedures*, they do not have to be procedures in a programming language sense; they can be functions.

6.5.1 Creating a Simple Stored Procedure

Let us look at the example stored procedure written in SQL shown in Figure 6.8. We see that stored procedures must have a name; this stored procedure

has the name 'ShowNumberOfOrders.' Otherwise, it just contains an SQL statement that is precompiled and stored at the server.

```
CREATE PROCEDURE ShowNumberOfOrders
SELECT C.cid, C.cname, COUNT(*)
       FROM    Customers C, Orders a
       WHERE   C.cid = O.cid
       GROUP BY C.cid, C.cname
```

Figure 6.8 A Stored Procedure in SQL

Stored procedures can also have parameters. These parameters have to be valid SQL types, and have one of three different modes: IN, OUT, or INOUT. IN parameters are arguments to' the stored procedure. OUT parameters are returned from the stored procedure; it assigns values to all OUT parameters that the user can process. INOUT parameters combine the properties of IN and OUT parameters: They contain values to be passed to the stored procedures, and the stored procedure can set their values as return values. Stored procedures enforce strict type conformance: If a parameter is of type INTEGER, it cannot be called with an argument of type VARCHAR.

Let us look at an example of a stored procedure with arguments. The stored procedure shown in Figure 6.9 has two arguments: book_isbn and addedQty. It updates the available number of copies of a book with the quantity from a new shipment.

```
CREATE PROCEDURE AddInventory (
                 IN book_isbn CHAR(lO),
                 IN addedQty INTEGER)
UPDATE Books
       SET     qty_in_stock = qtyjn_stock + addedQty
       WHERE   bookjsbn = isbn
```

Figure 6.9 A Stored Procedure with Arguments

Stored procedures do not have to be written in SQL; they can be written in any host language. As an example, the stored procedure shown in Figure 6.10 is a Java function that is dynamically executed by the database server whenever it is called by the dient:

6.5.2 Calling Stored Procedures

Stored procedures can be called in interactive SQL with the CALL statement:

```
CREATE PROCEDURE RallkCustomers(IN number INTEGER)
LANGUAGE Java
EXTERNAL NAME 'file:///c:/storedProcedures/rank.jar'
```

Figure 6.10 A Stored Procedure in Java

```
CALL storedProcedureName(argumentl, argument2, ... , argumentN);
```

In Embedded SQL, the arguments to a stored procedure are usually variables in the host language. For example, the stored procedure AddInventory would be called as follows:

```
EXEC SQL BEGIN DECLARE SECTION
char isbn[lO];
long qty;
EXEC SQL END DECLARE SECTION

// set isbn and qty to some values
EXEC SQL CALL AddInventory(:isbn,:qty);
```

Calling Stored Procedures from JDBC

We can call stored procedures from JDBC using the CallableStatment class. CallableStatement is a subclass of PreparedStatement and provides the same functionality. A stored procedure could contain multiple SQL staternents or a series of SQL statements-thus, the result could be many different ResultSet objects. We illustrate the case when the stored procedure result is a single ResultSet.

```
CallableStatement cstmt=
            con.prepareCall("{call ShowNumberOfOrders}");
ResultSet rs = cstmt.executeQueryO
while (rs.next())
```

Calling Stored Procedures from SQLJ

The stored procedure 'ShowNumberOfOrders' is called as follows using SQLJ:

```
// create the cursor class
#sql !terator CustomerInfo(int cid, String cname, int count);

// create the cursor
```

```
CustomerInfo customerinfo;

// call the stored procedure
#sql customerinfo = {CALL ShowNumberOfOrders};
while (customerinfo.nextO) {
    System.out.println(customerinfo.cid() + "," +
                customerinfo.count());
}
```

6.5.3 SQLIPSM

All major database systems provide ways for users to write stored procedures in a simple, general purpose language closely aligned with SQL. In this section, we briefly discuss the SQL/PSM standard, which is representative of most vendor-specific languages. In PSM, we define modules, which are collections of stored procedures, temporary relations, and other declarations.

In SQL/PSM, we declare a stored procedure as follows:

```
CREATE PROCEDURE name (parameter1,..., parameterN)
    local variable declarations
    procedure code;
```

We can declare a function similarly as follows:

```
CREATE FUNCTION name (parameterl,..., parameterN)
            RETURNS sqIDataType
    local variable declarations
    function code;
```

Each parameter is a triple consisting of the mode (IN, OUT, or INOUT as discussed in the previous section), the parameter name, and the SQL datatype of the parameter. We can seen very simple SQL/PSM procedures in Section 6.5.1. In this case, the local variable declarations were empty, and the procedure code consisted of an SQL query.

We start out with an example of a SQL/PSM function that illustrates the main SQL/PSM constructs. The function takes as input a customer identified by her *cid* and a year. The function returns the rating of the customer, which is defined as follows: Customers who have bought more than ten books during the year are rated 'two'; customer who have purchased between 5 and 10 books are rated 'one', otherwise the customer is rated 'zero'. The following SQL/PSM code computes the rating for a given customer and year.

```
CREATE PROCEDURE RateCustomer
```

```
                    (IN custId INTEGER, IN year INTEGER)
                    RETURNS INTEGER
DECLARE rating INTEGER;
DECLARE numOrders INTEGER;
SET numOrders =
        (SELECT COUNT(*) FROM Orders 0 WHERE O.tid = custId);
IF (numOrders> 10) THEN rating=2;
ELSEIF (numOrders>5) THEN rating=1;
ELSE rating=O;
END IF;
RETURN rating;
```

Let us use this example to give a short overview of some SQL/PSM constructs:

- We can declare local variables using the DECLARE statement. In our example, we declare two local variables: 'rating', and 'numOrders'.
- PSM/SQL functions return values via the RETURN statement. In our example, we return the value of the local variable 'rating'.
- We can assign values to variables with the SET statement. In our example, we assigned the return value of a query to the variable 'numOrders'.
- SQL/PSM has branches and loops. Branches have the following form:

```
IF (condition) THEN statements;
ELSEIF statements;

ELSEIF statements;
ELSE statements; END IF
```

  Loops are of the form

```
LOOP
        staternents:
END LOOP
```

- Queries can be used as part of expressions in branches; queries that return a single value can be assigned to variables as in our example above.
- 'We can use the same cursor statements as in Embedded SQL (OPEN, FETCH, CLOSE), but we do not need the EXEC SQL constructs, and variables do not have to be prefixed by a colon ':'.

We only gave a very short overview of SQL/PSM; the references at the end of the chapter provide more information.

6.6 CASE STUDY: THE INTERNET BOOK SHOP

DBDudes finished logical database design, as discussed in Section 3.8, and now consider the queries that they have to support. They expect that the application logic will be implemented in Java, and so they consider JDBC and SQLJ as possible candidates for interfacing the database system with application code.

Recall that DBDudes settled on the following schema:

Books(*isbn:* CHAR(10), *title:* CHAR(8), *author:* CHAR(80),
qty_in_stock: INTEGER, *price:* REAL, *year_published:* INTEGER)
Customers(*cid:* INTEGER, *cname:* CHAR(80), *address:* CHAR(200))
Orders(*ordernum:* INTEGER, *isbn:* CHAR(lO), *cid:* INTEGER,
cardnum: CHAR(l6), *qty:* INTEGER, *order_date:* DATE, *ship_date:* DATE)

Now, DBDudes considers the types of queries and updates that will arise. They first create a list of tasks that will be performed in the application. Tasks performed by customers include the following.

- Customers search books by author name, title, or ISBN.
- Customers register with the website. Registered customers might want to change their contact information. DBDudes realize that they have to augment the Customers table with additional information to capture login and password information for each customer; we do not discuss this aspect any further.
- Customers check out a final shopping basket to complete a sale.
- Customers add and delete books from a 'shopping basket' at the website.
- Customers check the status of existing orders and look at old orders.

Administrative tasks performed by employees of B&N are listed next.

- Employees look up customer contact information.
- Employees add new books to the inventory.
- Employees fulfill orders, and need to update the shipping date of individual books.
- Employees analyze the data to find profitable customers and customers likely to respond to special marketing campaigns.

Next, DBDudes consider the types of queries that will arise out of these tasks. To support searching for books by name, author, title, or ISBN, DBDudes decide to write a stored procedure as follows:

```
CREATE PROCEDURE SearchByISBN (IN book.isbn CHAR(10))
    SELECT B.title, B.author, B.qty_in_stock, B.price, B.yeaLpublished
    FROM   Books B
    WHERE  B.isbn = book.isbn
```

Placing an order involves inserting one or more records into the Orders table. Since DBDudes has not yet chosen the Java-based technology to program the application logic, they assume for now that the individual books in the order are stored at the application layer in a Java array. To finalize the order, they write the following JDBC code shown in Figure 6.11, which inserts the elements from the array into the Orders table. Note that this code fragment assumes several Java variables have been set beforehand.

```
String sql = "INSERT INTO Orders VALUES(7, 7, 7, 7, 7, 7)";
PreparedStatement pstmt = con.prepareStatement(sql);
con.setAutoCommit(false);

try {
    // orderList is a vector of Order objects
    // ordernum is the current order number
    // dd is the ID of the customer, cardnum is the credit card number
    for (int i=O; iiorderList.lengthO; i++)
        // now instantiate the parameters with values
        Order currentOrder = orderList[i];
        pstmt.clearParameters();
        pstmt.setInt(l, ordernum);
        pstmt.setString(2, Order.getlsbnO);
        pstmt.setInt(3, dd);
        pstmt.setString(4, creditCardNum);
        pstmt.setlnt(5, Order.getQtyO);
        pstmt.setDate(6, null);

        pstmt.executeUpdate();
    }
    con.commit();
catch (SqLException e){
    con.rollbackO;
    System.out.println(e.getMessage());
}
```

Figure 6.11 Inserting a Completed Order into the Database

DBDudes writes other JDBC code and stored procedures for all of the remaining tasks. They use code similar to some of the fragments that we have seen in this chapter.

- Establishing a connection to a database, as shown in Figure 6.2.
- Adding new books to the inventory, as shown in Figure 6.3.
- Processing results from SQL queries as shown in Figure 6.4-
- For each customer, showing how many orders he or she has placed. We showed a sample stored procedure for this query in Figure 6.8.
- Increasing the available number of copies of a book by adding inventory, as shown in Figure 6.9.
- Ranking customers according to their purchases, as shown in Figure 6.10.

DBDudcs takes care to make the application robust by processing exceptions and warnings, as shown in Figure 6.6.

DBDudes also decide to write a trigger, which is shown in Figure 6.12. Whenever a new order is entered into the Orders table, it is inserted with ship_date set to NULL. The trigger processes each row in the order and calls the stored procedure 'UpdateShipDate'. This stored procedure (whose code is not shown here) updates the (anticipated) ship_date of the new order to 'tomorrow', in case qtyjlLstock of the corresponding book in the Books table is greater than zero. Otherwise, the stored procedme sets the ship_date to two weeks.

```
CREATE TRIGGER update_ShipDate
        AFTER INSERT ON Orders                    /* Event */
  FOR EACH ROW
  BEGIN CALL UpdatcShipDate(new); END             /* Action */
```

Figure 6.12 Trigger to Update the Shipping Date of New Orders

6.7 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- Why is it not straightforward to integrate SQL queries with a host programming language? (Section 6.1.1)
- How do we declare variables in Ernbcdded SQL? (Section 6.1.1)

- How do we use SQL statements within a host langl.lage? How do we check for errors in statement execution? (Section 6.1.1)
- Explain the impedance mismatch between host languages and SQL, and describe how cursors address this. (Section 6.1.2)
- What properties can cursors have? (Section 6.1.2)
- What is Dynamic SQL and how is it different from Embedded SQL? (Section 6.1.3)
- What is JDBC and what are its advantages? (Section 6.2)
- What are the components of the JDBC architecture? Describe four different architectural alternatives for JDBC drivers. (Section 6.2.1)
- How do we load JDBC drivers in Java code? (Section 6.3.1)
- How do we manage connections to data sources? What properties can connections have? (Section 6.3.2)
- What alternatives does JDBC provide for executing SQL DML and DDL statements? (Section 6.3.3)
- How do we handle exceptions and warnings in JDBC? (Section 6.3.5)
- 'What functionality provides the DatabaseMetaDataclass? (Section 6.3.6)
- What is SQLJ and how is it different from JDBC? (Section 6.4)
- Why are stored procedures important? How do we declare stored procedures and how are they called from application code? (Section 6.5)

EXERCISES

Exercise 6.1 Briefly answer the following questions.

1. Explain the following terms: Cursor, Embedded SQL, JDBC, SQLJ, stored procedure.
2. What are the differences between JDBC and SQLJ? \Nhy do they both exist?
3. Explain the term *stored procedure*, and give examples why stored procedures are useful.

Exercise 6.2 Explain how the following steps are performed in JDBC:

1. Connect to a data source.
2. Start, commit, and abort transactions.
3. Call a stored procedure.

How are these steps performed in SQLJ?

Exercise 6.3 Compare exception handling and handling of warnings ill embedded SQL, dynamic SQL, .IDBC, and SQL.I.

Exercise 6.4 Answer the following questions.

1. Why do we need a precompiler to translate embedded SQL and SQL.J? Why do we not need a precompiler for .IDBC?
2. SQL.J and embedded SQL use variables in the host language to pass parameters to SQL queries, whereas .JDBC uses placeholders marked with a '?'. Explain the difference, and why the different mechanisms are needed.

Exercise 6.5 A dynamic web site generates HTML pages from information stored in a database. Whenever a page is requested, is it dynamically assembled from static data and data in a database, resulting in a database access. Connecting to the database is usually a time-consuming process, since resources need to be allocated, and the user needs to be authenticated. Therefore, connection pooling--setting up a pool of persistent database connections and then reusing them for different requests can significantly improve the performance of database-backed websites. Since servlets can keep information beyond single requests, we can create a connection pool, and allocate resources from it to new requests.

Write a connection pool class that provides the following methods:

- Create the pool with a specified number of open connections to the database system.
- Obtain an open connection from the pool.
- Release a connection to the pool.
- Destroy the pool and close all connections.

PROJECT-BASED EXERCISES

In the following exercises, you will create database-backed applications. In this chapter, you will create the parts of the application that access the database. In the next chapter, you will extend this code to other aspects of the application. Detailed information about these exercises and material for more exercises can be found online at

http://www.cs.wisc.edu/-dbbook

Exercise 6.6 Recall the Notown Records database that you worked with in Exercise 2.5 and Exercise 3.15. You have now been tasked with designing a website for Notown. It should provide the following functionality:

- Users can search for records by name of the musician, title of the album, and Bame of the song.
- Users can register with the site, and registered users ca.n log on to the site. Once logged on, users should not have to log on again unless they are inactive for a long time.
- Users who have logged on to the site can add items to a shopping basket.
- Users with items in their shopping basket can check out and ma.ke a purchase.

NOtOWll wants to use JDBC to access the database. Write .JDBC code that performs the necessary data access and manipulation. You will integrate this code with application logic and presentation in the next chapter.

If Notown had chosen SQLJ instead of JDBC, how would your code change?

Exercise 6.7 Recall the database schema for Prescriptions-R-X that you created in Exercise 2.7. The Prescriptions-R-X chain of pharmacies has now engaged you to design their new website. The website has two different classes of users: doctors and patients. Doctors should be able to enter new prescriptions for their patients and modify existing prescriptions. Patients should be able to declare themselves as patients of a doctor; they should be able to check the status of their prescriptions online; and they should be able to purchase the prescriptions online so that the drugs can be shipped to their home address.

Follow the analogous steps from Exercise 6.6 to write JDBC code that performs the necessary data access and manipulation. You will integrate this code with application logic and presentation in the next chapter.

Exercise 6.8 Recall the university database schema that you worked with in Exercise 5.l. The university has decided to move enrollment to an online system. The website has two different classes of users: faculty and students. Faculty should be able to create new courses and delete existing courses, and students should be able to enroll in existing courses.

Follow the analogous steps from Exercise 6.6 to write JDBC code that performs the necessary data access and manipulation. You will integrate this code with application logic and presentation in the next chapter.

Exercise 6.9 Recall the airline reservation schema that you worked on in Exercise 5.3. Design an online airline reservation system. The reservation system will have two types of users: airline employees, and airline passengers. Airline employees can schedule new flights and cancel existing flights. Airline passengers can book existing flights from a given destination.

Follow the analogous steps from Exercise 6.6 to write JDBC code that performs the necessary data access and manipulation. You will integrate this code with application logic and presentation in the next chapter.

BIBLIOGRAPHIC NOTES

Information on ODBC can be found on Microsoft's web page (www.microsoft.com/data/odbc), and information on JDBC can be found on the Java web page (java. sun. com/products/jdbc). There exist rnany books on ODBC, for example, Sanders' ODBC Developer's Guicle [652] and the lvIicrosoft ODBC SDK [533]. Books on JDBC include works by Hamilton et al. [359], Reese [621], and White et a!. [773].

7

INTERNET APPLICATIONS

- How do we name resources on the Internet?
- How do Web browsers and webservers communicate?
- How do we present documents on the Internet? How do we differentiate between formatting and content?
- What is a three-tier application architecture? How do we write three-tiered applications?
- Why do we have application servers?
- Key concepts: Uniform Resource Identifiers (URI), Uniform Resource Locators (URL); Hypertext Transfer Protocol (HTTP), stateless protocol; Java; HTML; XML, XML DTD; three-tier architecture, client-server architecture; HTML forms; JavaScript; cascading style sheets, XSL; application server; Common Gateway Interface (CGI); servlet; JavaServer Page (JSP); cookie

Wow! They've got the Internet on computers now!

--Homer Simpson, *The Simpsons*

7.1 INTROpUCTION

The proliferation of computer networks, including the Internet and corporate 'intranets,' has enabled users to access a large number of data sources. This increased access to databases is likely to have a great practical impact; data and services can now be offered directly to customers in ways impossible until

recently. Examples of such electronic commerce applications include purchasing books through a \Veb retailer such as Amazon.com, engaging in online auctions at a site such as eBay, and exchanging bids and specifications for products between companies. The emergence of standards such as XrvIL for describing the content of documents is likely to further accelerate electronic commerce and other online applications.

While the first generation of Internet sites were collections of HTML files, most major sites today store a large part (if not all) of their data in database systems. They rely on DBMSs to provide fast, reliable responses to user requests received over the Internet. This is especially true of sites for electronic commerce and other business applications.

In this chapter, we present an overview of concepts that are central to Internet application development. We start out with a basic overview of how the Internet works in Section 7.2. We introduce HTML and XML, two data formats that are used to present data on the Internet, in Sections 7.3 and 7.4. In Section 7.5, we introduce three-tier architectures, a way of structuring Internet applications into different layers that encapsulate different functionality. In Sections 7.6 and 7.7, we describe the presentation layer and the middle layer in detail; the DBMS is the third layer. We conclude the chapter by discussing our B&N case study in Section 7.8.

Examples that appear in this chapter are available online at

http://www.cs.wisc.edu/-dbbook

7.2 INTERNET CONCEPTS

The Internet has emerged as a universal connector between globally distributed software systems. To understand how it works, we begin by discussing two basic issues: how sites on the Internet are identified, and how programs at one site communicate with other sites.

We first introduce Uniform Resource Identifiers, a naming schema for locating resources on the Internet in Section 7.2.1. \Ve then talk about the most popular protocol for accessing resources over the Web, the hypertext transfer protocol (HTTP) in Section 7.2.2.

7.2.1 Uniform Resource Identifiers

Uniform Resource Identifiers (URIs), are strings that uniquely identify resources 011 the Internet. A resource is any kind of information that can

Distributed Applications and Service-Oriented Architectures:

The advent of XML, due to its loosely-coupled nature, has made information exchange between different applications feasible to an extent previously unseen. By using XML for information exchange, applications can be written in different programming languages, run on different operating systems, and yet they can still share information with each other. There are also standards for externally describing the intended content of an XML file or message, most notably the recently adopted W3C XML Schemas standard.

A promising concept that has arisen out of the XML revolution is the notion of a Web service. A Web service is an application that provides a well-defined service, packaged as a set of remotely callable procedures accessible through the Internet. Web services have the potential to enable powerful new applications by *composing* existing Web services-all communicating seamlessly thanks to the use of standardizedXML-based information exchange. Several technologies have been developed or are currently under development that facilitate design and implementation of distributed applications. SOAP is a W3C standard for XML-based invocation of remote services (think XML RPC) that allows distributed applications to communicate either synchronously or asynchronously via structured, typed XML messages. SOAP calls can ride on a variety of underlying transport layers, including HTTP (part of what is making SOAP so successful) and various reliable messaging layers. Related to the SOAP standard are W3C's Web Services Description Language (WSDL) for describing Web service interfaces, and Universal Description, Discovery, and Integration (UDDI), a WSDL-based Web services registry standard (think yellow pages for Web services).

SOAP-based Web services are the foundation for Microsoft's recently released .NET framework, their application development infrastructure and associated run-time system for developing distributed applications, as well as for the Web services offerings of major software vendors such as IBM, BEA, and others. Many large software application vendors (major companies like PeopleSoft and SAP) have announced plans to provide Web service interfaces to their products and the data that they manage, and many are hoping that XML and Web services will finally provide the answer to the long-standing problem of enterprise application integration. Web services are also being looked to as a natural foundation for the next generation of business process management (or workflow) systems.

be identified by a URI, and examples include webpages, images, downloadable files, services that can be remotely invoked, mailboxes, and so on. The most common kind of resource is a static file (such as a HTML document), but a resource may also be a dynamically-generated HTML file, a movie, the output of a program, etc.

A URI has three parts:

- The (name of the) protocol used to access the resource.
- The host computer where the resource is located.
- The path name of the resource itself on the host computer.

Consider an example URI, such as http://www.bookstore.com/index.html. This URI can be interpreted as follows. Use the HTTP protocol (explained in the next section) to retrieve the document index.html located at the computer www.bookstore.com.This example URI is an instance of a Universal Resource Locator (URL), a subset of the more general URI naming scheme; the distinction is not important for our purposes. As another example, the following HTML fragment shows a URI that is an email address:

```
<a href=lImailto:webmaster@bookstore.com">Email the webmaster.</A>
```

7.2.2 The Hypertext Transfer Protocol (HTTP)

A communication protocol is a set of standards that defines the structure of messages between two communicating parties so that they can understand each other's messages. The Hypertext Transfer Protocol (HTTP) is the most common communication protocol used over the Internet. It is a client-server protocol in which a client (usually a Web browser) sends a request to an HTTP server, which sends a response back to the client. When a user requests a webpage (e.g., clicks on a hyperlink), the browser sends **HTTP** request messages for the objects in the page to the server. The server receives the requests and responds with **HTTP** response messages, which include the objects. It is important to recognize that HTTP is used to transmit all kinds of resources, not just files, but most resources on the Internet today are either static files or files output from server-side scripts.

A variant of the HTTP protocol called the Secure Sockets Layer (SSL) protocol uses encryption to exchange information securely between client and server. We postpone a discussion of SSL to Section 21.5.2 and present the basic HTTP protocol in this chapter.

As an example, consider what happens if a user clicks on the following link: http://www.bookstore.com/index.html. We first explain the structure of an HTTP request message and then the structure of an HTTP response message.

HTTP Requests

The client (Web browser) establishes a connection with the webserver that hosts the resource and sends a HTTP request message. The following example shows a sample HTTP request message:

```
GET index.html HTTP/1.1
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
```

The general structure of an HTTP request consists of several lines of ASCII text, with an empty line at the end. The first line, the request line, has three fields: the HTTP method field, the URI field, and the HTTP version field. The method field can take on values GET and POST; in the example the message requests the object index.html. (We discuss the differences between HTTP GET and HTTP POST in detail in Section 7.11.) The version field indicates which version of HTTP is used by the client and can be used for future extensions of the protocol. The user agent indicates the type of the client (e.g., versions of Netscape or Internet Explorer); we do not discuss this option further. The third line, starting with Accept, indicates what types of files the client is willing to accept. For example, if the page index.html contains a movie file with the extension .mpg, the server will not send this file to the client, as the client is not ready to accept it.

HTTP Responses

The server responds with an HTTP response message. It retrieves the page index.html, uses it to assemble the HTTP response message, and sends the message to the client. A sample HTTP response looks like this:

```
HTTP/1.1 200 OK
Date: Mon, 04 Mar 2002 12:00:00 GMT
Content-Length: 1024
Content-Type: text/html
Last-Modified: Mall, 22 Jun 1998 09:23:24 GMT
<HTML>
<HEAD>
</HEAD>
<BODY>
```

<H1>Barns and Nobble Internet Bookstore</H1>
Our inventory:
<H3>Science</H3>
<B>The Character of Physical Law</B>

The HTTP response message has three parts: a status line, several header lines, and the body of the message (which contains the actual object that the client requested). The status line has three fields (analogous to the request line of the HTTP request message): the HTTP version (HTTP/1.1), a status code (200), and an associated server message (OK). Common status codes and associated messages are:

- 200 OK: The request succeeded and the object is contained in the body of the response message";
- 400 Bad Request: A generic error code indicating that the request could not be fulfilled by the server.
- 404 Not Found: The requested object does not exist on the server.
- 505 HTTP Version Not Supported: The HTTP protocol version that the client uses is not supported by the server. (Recall that the HTTP protocol version sent in the client's request.)

Our example has three header lines: The date header line indicates the time and date when the HTTP response was created (not that this is not the object creation time). The Last-Modified header line indicates when the object was created. The Content-Length header line indicates the number of bytes in the object being sent after the last header line. The Content-Type header line indicates that the object in the entity body is HTML text.

The client (the Web browser) receives the response message, extracts the HTML file, parses it, and displays it. In doing so, it might find additional URIs in the file, and it then uses the HTTP protocol to retrieve each of these resources, establishing a new connection each time.

One important issue is that the HTTP protocol is a stateless protocol. Every message----from, the client to the HTTP server and vice-versa-is self-contained, and the connection established with a request is maintained only until the response message is sent. The protocol provides no mechanism to automatically 'remember' previous interactions between client and server.

The stateless nature of the HTTP protocol has a major impact on how Internet applications are written. Consider a user who interacts with our exalIlple

bookstore application. Assume that the bookstore permits users to log into the site and then carry out several actions, such as ordering books or changing their address, without logging in again (until the login expires or the user logs out). How do we keep track of whether a user is logged in or not? Since HTTP is stateless, we cannot switch to a different state (say the 'logged in' state) at the protocol level. Instead, for every request that the user (more precisely, his or her Web browser) sends to the server, we must encode any *state* information required by the application, such as the user's login status. Alternatively, the server-side application code must maintain this state information and look it up on a per-request basis. This issue is explored further in Section 7.7.5.

Note that the statelessness of HTTP is a tradeoff between ease of implementation of the HTTP protocol and ease of application development. The designers of HTTP chose to keep the protocol itself simple, and deferred any functionality beyond the request of objects to application layers above the HTTP protocol.

7.3 HTML DOCUMENTS

In this section and the next, we focus on introducing HTML and XML. In Section 7.6, we consider how applications can use HTML and XML to create forms that capture user input, communicate with an HTTP server, and convert the results produced by the data management layer into one of these formats.

HTML is a simple language used to describe a document. It is also called a **markup language** because HTML works by augmenting regular text with 'marks' that hold special meaning for a Web browser. Commands in the language, called tags, consist (usually) of a start tag and an **end** tag of the form <TAG> and </TAG>, respectively. For example, consider the HTML fragment shown in Figure 7.1. It describes a webpage that shows a list of books. The document is enclosed by the tags <HTML> and </HTML>, marking it as an HTML document. The remainder of the document-enclosed in <BODY> ... </BoDY>-contains information about three books. Data about each book is represented as an unordered list (UL) whose entries are marked with the LI tag. HTML defines the set of valid tags as well as the meaning of the tags. For example, HTML specifies that the tag <TITLE> is a valid tag that denotes the title of the document. As another example, the tag <UL> always denotes an unordered list.

Audio, video, and even programs (written in Java, a highly portable language) can be included in HTML documents. When a user retrieves such a document using a suitable browser, images in the document arc displayed, audio and video clips are played, and embedded programs are executed at the user's machine; the result is a rich multimedia presentation. The case with which HTML docu-

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<Hl>Barns and Nobble Internet Bookstore</Hl>
Our inventory:
<H3>Science</H3>
    <B>The Character of Physical Law</B>
    <UL>
        <LI>Author: Richard Feynman</LI>
        <LI>Published 1980</LI>
        <Ll>Hardcover</LI>
    </UL>
<H3>Fiction</H3>
    <B>Waiting for the Mahatma</B>
    <UL>
        <LI>Author: R.K. Narayan</LI>
        <LI>Published 1981</Ll>
    </UL>
    <B>The English Teacher</B>
    <UL>
        <LI>Author: R.K. Narayan</LI>
        <LI>Published 1980</LI>
        <LI>Paperback</LI>
    </UL>
</BODY>
</HTML>
```

Figure 7.1 Book Listing in HTML

ments can be created—there are now visual editors that automatically generate HTML----and accessed using Internet browsers has fueled the explosive growth of the Web.

7.4 XML DOCUMENTS

In this section, we introduce XML as a document format, and consider how applications can utilize XML. Managing XML documents in a DBMS poses several new challenges; we discuss this aspect of XML in Chapter 27.

vVhile HTML can be used to mark up documents for display purposes, it is not adequate to describe the structure of the content for more general applications. For example, we can send the HTML document shown in Figure 7.1 to another application that displays it, but the second application cannot distinguish the first names of authors from their last names. (The application can try to recover such information by looking at the text inside the tags, but this defeats the purpose of using tags to describe document structure.) Therefore, HTML is unsuitable for the exchange of complex documents containing product specifications or bids, for example.

Extensible Markup Language (XML) is a markup language developed to remedy the shortcomings of HTML. In contrast to a fixed set of tags whose meaning is specified by the language (as in HTML), XML allows users to define new collections of tags that can be used to structure any type of data or document the user wishes to transmit. XML is an important bridge between the document-oriented view of data implicit in HTML and the schema-oriented view of data that is central to a DBMS. It has the potential to make database systems more tightly integrated into Web applications than ever before.

XML emerged from the confluence of two technologies, SGML and HTML. The Standard Generalized Markup Language (SGML) is a metalanguage that allows the definition of data and document interchange languages such as HTML. The SGML standard was published in 1988, and many organizations that rnanage a large number of complex documents have adopted it. Due to its generality, SGML is complex and requires sophisticated programs to harness its full potential. XML was developed to have much of the power of SGML while remaining relatively simple. Nonetheless, XML, like SGML, allows the definition of new document markup languages.

Although XML does not prevent a user from designing tags that encode the display of the data in a Web browser, there is a style language for XML called Extensible Style Language (XSL). XSL is a standard way of describing how an XML docmnent that adheres to a certain vocabulary of tags should be displayed.

7.4.1 Introduction to XML

We use the small XML docmnent shown in Figure 7.2 as an example.

- Elements: Elements, also called tags, arc the primary building blocks of an XML docmnent. The start of the content of an element ELM is marked with <ELM>, which is called the start tag, and the end of the content end is marked with </ELM>, called the end tag. In our example document.

The Design Goals of XML: XML was developed starting in 1996 by a working group under guidance of the World Wide Web Consortium (W3C) XML Special Interest Group. The design goals for XML included the following:

1. XML should be compatible with SGML.
2. It should be easy to write programs that process XML documents.
3. The design of XML should be formal and concise.

the element BOOKLIST encloses all information in the sample document. The element BOOK demarcates all data associated with a single book. XML elements are case sensitive: the element BOOK is different from Book. Elements must be properly nested. Start tags that appear inside the content of other tags must have a corresponding end tag. For example, consider the following XML fragment:

```
<BOOK>
    <AUTHOR>
        <FIRSTNAME>Richard</FIRSTNAME>
        <LASTNAME>Feynluan</LASTNAME>
    </AUTHOR>
</BOOK>
```

The element AUTHOR is completely nested inside the element BOOK, and both the elements LASTNAME and FIRSTNAME are nested inside the element AUTHOR.

- Attributes: An element can have descriptive attributes that provide additional information about the element. The values of attributes are set inside the start tag of an element. For example, let ELM denote an element with the attribute att. We can set the value of att to value through the following expression: <ELM att="value">. All attribute values must be enclosed in quotes. In Figure 7.2, the element BOOK has two attributes. The attribute GENRE indicates the genre of the book (science or fiction) and the attribute FORMAT indicates whether the book is a hardcover or a paperback.

- Entity References: Entities are shortcuts for portions of common text or the content of external files, and we call the usage of an entity in the XML document an entity reference. Wherever an entity reference appears in the document, it is textually replaced by its content. Entity references start with a '&' and end with a ';'. Five predefined entities in XML are placeholders for chara.cters with special meaning in XML. For example, the

```
<?xml version=11.0" encoding="UTF-S" standalone=llyesll?>
<BOOKLIST>
<BOOK GENRE=" Science" FORMAT=" Hardcover" >
    <AUTHOR>
        <FIRSTNAME>Richard</FIRSTNAME>
        <LASTNAME>Feynman</LASTNAME>
    </AUTHOR>
    <TITLE>The Character of Physical Law</TITLE>
    <PUBLISHED>1980</PUBLISHED>
</BOOK>
<BOOK> GENRE=" Fiction" >
    <AUTHOR>
        <FIRSTNAME>R.K.</FIRSTNAME>
        <LASTNAME>Narayan</LASTNAME>
    </AUTHOR>
    <TITLE>Waiting for the Mahatma</TITLE>
    <PUBLISHED>1981</PUBLISHED>
</BOOK>
<BOOK GENRE=" Fiction" >
    <AUTHOR>
        <FIRSTNAME>R.K.</FIRSTNAME>
        <LASTNAME>Narayan</LASTNAME>
    </AUTHOR>
    <TITLE>The English Teacher</TITLE>
    <PUBLISHED> 1980</PUBLISHED>
</BOOK>
</BOOKLIST>
```

Figure 7.2 Book Information in XML

< character that marks the beginning of an XML command is reserved and has to be represented by the entity It. The other four reserved characters are &, >, ", and '; they are represented by the entities amp, gt, quot, and apos. For example, the text '1 < 5' has to be encoded in an XML document as follows: ' 1&1t ;5'. We can also use entities to insert arbitrary Unicode characters into the text. Unicode is a standard for character representations, similar to ASCII. For example, we can display the Japanese Hiragana character *a* using the entity reference &#x3042.

- Comments: We can insert comments anywhere in an XML document. Comments start with <! - and end with ->. Comments can contain arbitrary text except the string --.

- Document Type Declarations (DTDs): In XML, we can define our own markup language. A DTD is a set of rules that allows us to specify our own set of elements, attributes, and entities. Thus, a DTD is basically a grammar that indicates what tags are allowed, in what order they can appear, and how they can be nested. We discuss DTDs in detail in the next section.

We call an XML document well-formed if it has no associated DTD but follows these structural guidelines:

- The document starts with an XML declaration. An example of an XML declaration is the first line of the XML document shown in Figure 7.2.
- A root element contains all the other elements. In our example, the root element is the element BOOKLIST.
- All elements must be properly nested. This requirement states that start and end tags of an element must appear within the same enclosing element.

7.4.2 XML DTDs

A DTD is a set of rules that allows us to specify our own set of elements, attributes, and entities. A DTD specifies which elements we can use and constraints on these elements, for example, how elements can be nested and where elements can appear in the document. We call a document valid if a DTD is associated with it and the document is structured according to the rules set by the DTD. In the remainder of this section, we use the example DTD shown in Figure 7.3 to illustrate how to construct DTDs.

```
<!DOCTYPE BOOKLIST [
<! ELEMENT BOOKLIST (BOOK)*>
    <! ELEMENT BOOK (AUTHOR,TITLE,PUBLISHED?»
        <!ELEMENT AUTHOR (FIRSTNAME,LASTNAME»
            <! ELEMENT FIRSTNAME (#PCDATA»
            <! ELEMENT LASTNAME (#PCDATA»
        <! ELEMENT TITLE (#PCDATA»
        <!ELEMENT PUBLISHED (#PCDATA»
    <! ATTLIST BOOK GENRE (ScienceIFiction) #REQUIRED>
    <!ATTLIST BOOK FORMAT (PaperbackIHardcover) "Paperback">
]>
```

Figure 7.3 Bookstore XML DTD

A DTD is enclosed in <!DOCTYPE name [DTDdeclarationJ >, where name is the name of the outermost enclosing tag, and DTDdeclaration is the text of the rules of the DTD. The DTD starts with the outermost element—the *root element*—which is BOOKLIST in our example. Consider the next rule:

```
<!ELEMENT BOOKLIST (BOOK)*>
```

This rule tells us that the element BOOKLIST consists of zero or more BOOK elements. The * after BOOK indicates how many BOOK elements can appear inside the BOOKLIST element. A * denotes zero or more occurrences, a + denotes one or more occurrences, and a? denotes zero or one occurrence. For example, if we want to ensure that a BOOKLIST has at least one book, we could change the rule as follows:

```
<!ELEMENT BOOKLIST (BOOK)+>
```

Let us look at the next rule:

```
<!ELEMENT BOOK (AUTHOR,TITLE,PUBLISHED?»
```

This rule states that a BOOK element contains a AUTHOR element, a TITLE element, and an optional PUBLISHED clement. Note the use of the? to indicate that the information is optional by having zero or one occurrence of the element. Let us move ahead to the following rule:

```
< !ELEMENT LASTNAME (#PCDATA»
```

Until now we considered only elements that contained other elements. This rule states that LASTNAME is an element that does not contain other elements, but contains actual text. Elements that only contain other elements are said to have element content, whereas elements that also contain #PCDATA are said to have mixed content. In general, an element type declaration has the following structure:

```
<!ELEMENT (contentType»
```

Five possible content types are:

- Other elements.
- The special syrnbol #PCDATA, which indicates (parsed) character data.
- The special symbol EMPTY, which indicates that the element has no content. Elements that have no content are not required to have an end tag.
- The special symbol ANY, which indicates that any content is permitted. This content should be avoided whenever possible since it disables all checking of the document structure inside the element.

- A regular expression constructed from the preceding four choices. A regular expression is one of the following:
  - exp1, exp2, exp3: A list of regular expressions.
  - exp*: An optional expression (zero or more occurrences).
  - exp?: An optional expression (zero or one occurrences).
  - exp+: A mandatory expression (one or more occurrences).
  - expl | exp2: expl or exp2.

Attributes of elements are declared outside the element. For example, consider the following attribute declaration from Figure 7.3:

```
<! ATTLIST BOOK GENRE (ScienceIFiction) #REQUIRED»
```

This XML DTD fragment specifies the attribute GENRE, which is an attribute of the element BOOK. The attribute can take two values: Science or Fiction. Each BOOK element must be described in its start tag by a GENRE attribute since the attribute is required as indicated by #REQUIRED. Let us look at the general structure of a DTD attribute declaration:

```
<! ATTLIST elementName (attName attType default)+>
```

The keyword ATTLIST indicates the beginning of an attribute declaration. The string elementName is the name of the element with which the following attribute dcfinition is associated. What follows is the declaration of one or more attributes. Each attribute has a name, as indicated by attName, and a type, as indicated by attType. XML defines several possible types for an attribute. We discuss only string types and enumerated types here. An attribute of type string can take any string as a value. We can declare such an attribute by setting its type field to CDATA. F'or example, we can declare a third attribute of type string of the elernent BOOK as follows:

```
<!ATTLIST BOOK edition CDATA "1">
```

If an attribute has an enumerated type, we list all its possible values in the attribute declaration. In our example, the attribute GENRE is an enumerated attribute type; its possible attribute values are 'Science' and 'Fiction'.

The last part of an attribute declaration is called its default specification. The DTD in Figure 7.3 shows two different default specifications: #REQUIRED and the string 'Paperback'. The default specification #REQUIRED indicates that the attribute is required and whenever its associated element appears somewhere in the XML document a value for the attribute must be specified. The default specification indicated by the string 'Paperback' indicates that the attribute is not required; whenever its associated element appears without setting

```
<?xml version=11.0" encoding=IUTF-8" standalone="no"?>
<!DOCTYPE BOOKLIST SYSTEM" books.dtd" >
<BOOKLIST>
    <BOOK GENRE=" Science" FORMAT=" Hardcover" >
        <AUTHOR>
```

Figure 7.4 Book Information in XML

XML Schema: The DTD mechanism has several limitations, in spite of its widespread use. For example, elements and attributes cannot be assigned types in a flexible way, and elements are always ordered, even if the application does not require this. XML Schema is a new W3C proposal that provides a more powerful way to describe document structure than DTDs; it is a superset of DTDs, allowing legacy data to be handled easily. An interesting aspect is that it supports uniqueness and foreign key constraints.

a value for the attribute, the attribute automatically takes the value 'Paperback'. For example, we can make the attribute value 'Science' the default value for the GENRE attribute as follows:

```
<! ATTLIST BOOK GENRE (ScienceIFiction) "Science" >
```

In our bookstore example, the XML document with a reference to the DTD is shown in Figure 7.4.

7.4.3 Domain-Specific DTDs

Recently, DTDs have been developed for several specialized domains-including a wide range of commercial, engineering, financial, industrial, and scientific domains----and a lot of the excitement about XML has its origins in the belief that more and more standardized DTDs will be developed. Standardized DTDs would enable seamless data exchange among heterogeneous sources, a problem solved today either by implementing specialized protocols such as Electronic Data Interchange (EDI) or by implementing ad hoc solutions.

Even in an environment where all XML data is valid, it is not possible to straightforwardly integrate several XML documents by matching elements in their DTDs, because even when two elements have identical names in two different DTDs, the meaning of the elements could be completely different. If both documents use a single, standard DTD, we avoid this problem. The

development of standardized DTDs is more a social process than a research problem, since the major players in a given domain or industry segment have to collaborate.

For example, the mathematical markup language (MathML) has been developed for encoding mathematical material on the Web. There are two types of MathML elements. The 28 presentation elements describe the layout structure of a document; examples are the **mrow** element, which indicates a horizontal row of characters, and the msup element, which indicates a base and a subscript. The 75 content elements describe mathematical concepts. An example is the plus element, which denotes the addition operator. (A third type of element, the math element, is used to pass parameters to the MathML processor.) MathML allows us to encode mathematical objects in both notations since the requirements of the user of the objects might be different. Content elements encode the precise mathematical meaning of an object without ambiguity, and the description can be used by applications such as computer algebra systems. On the other hand, good notation can suggest the logical structure to a human and emphasize key aspects of an object; presentation elements allow us to describe mathematical objects at this level.

For example, consider the following simple equation:

$$x^2 - 4x - 32 = 0$$

Using presentation elements, the equation is represented as follows:

```
<mrow>
    <mrow> <msup><mi>x</mi><mn>2</mn></msup>
        <mo>-</mo>
        <mrow><mn>4</mn>
            <mo>&invisibletimes;</mo>
            <mi>x</mi>
        </mrow>
        <mo>-</mo><mn>32</mn>
    </mrow><mo>=</mo><mn>O</nm>
</mrow>
```

Using content elements, the equation is described as follows:

```
<reln><eq/>
    <apply>
        <minus/>
        <apply> <power/> <ci>x</ci> <cn>2</cn> </apply>
        <apply> <times/> <cn>4</cn> <ci>x</ci> </apply>
        <cn>32</cn>
```

```
        </apply> <cn>O</cn>
    </reln>
```

Note the additional power that we gain from using MathML instead of encoding the formula in HTML. The common way of displaying mathematical objects inside an HTML object is to include images that display the objects, for example, as in the following code fragment:

```
<IMG SRC=lIimages/equation.gifu ALI=II x**2 - 4x - 32 = 10 ll >
```

The equation is encoded inside an IMG tag with an alternative display format specified in the ALI tag. Using this encoding of a mathematical object leads to the following presentation problems. First, the image is usually sized to match a certain font size, and on systems with other font sizes the image is either too small or too large. Second, on systems with a different background color, the picture does not blend into the background and the resolution of the image is usually inferior when printing the document. Apart from problems with changing presentations, we cannot easily search for a formula or formula fragments on a page, since there is no specific markup tag.

7.5 THE THREE-TIER APPLICATION ARCHITECTURE

In this section, we discuss the overall architecture of data-intensive Internet applications. Data-intensive Internet applications can be understood in terms of three different functional components: *data management, application logic,* and *pTesentation.* The component that handles data mallgement usually utilizes a DBMS for data storage, but application logic and presentation involve much more than just the DBMS itself.

We start with a short overview of the history of database-backed application architectures, and introduce single-tier and client-server architectures in Section 7.5.1. We explain the three-tier architecture in detail in Section 7.5.2, and show its advantages in Section 7.5.3.

7.5.1 Single-Tier and Client-Server Architectures

In this section, we provide some perspective on the three-tier architecture by discussing single-tier and client-server architectures, the predecessors of the three-tier architecture. Initially, data-intensive applications were combined into a single tier, including the DBMS, application logic, and user interface, as illustrated in Figure 7.5. The application typically ran on a mainframe, and users accessed it through *dumb teT'minals* that could perform only data input and display. This approach has the benefit of being easily maintained by a central administrator.

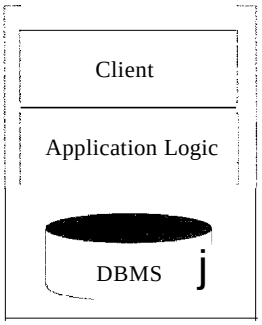


Figure 7.5 A Single-Tier Architecture

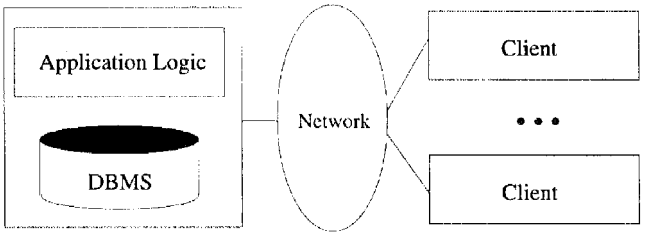


Figure 7.6 A Two-Server Architecture: Thin Clients

Single-tier architectures have a,n important drawback: Users expect graphical interfaces that require much more computational power than simple dumb terminals. Centralized computation of the graphical displays of such interfaces requires much more computational power than a single server has available, and thus single-tier architectures do not scale to thousands of users. The commoditization of the PC and the availability of cheap client computers led to the developlnent of the two-tier architecture.

Two-tier architectures, often also referred to as client-server architectures, consist of a client computer and a server computer, which interact through a well-defined protocol. What part of the functionality the client implements, and what part is left to the server, can vary. In the traditional client-server architecture, the client implements just the graphical user interface, and the server. implements both the business logic and the data management; such clients are often called thin clients, and this architecture is illustra,ted in Figure 7.6.

Other divisions are possible, such as more powerful clients that implement both user interface and business logic, or clients that implement user interface and part of the business logic, with the remaining part being implemented at the

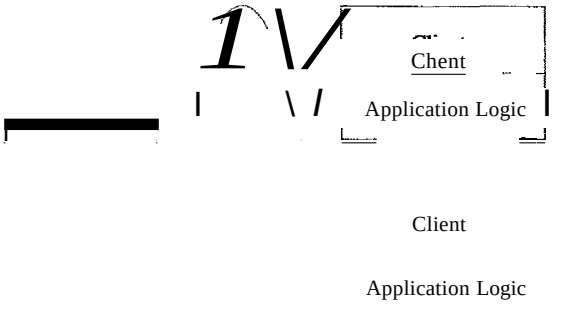


Figure 7.7 A Two-Tier Architecture: Thick Clients

server level; such clients are often called **thick** clients, and this architecture is illustrated in Figure 7.7.

Compared to the single-tier architecture, two-tier architectures physically separate the user interface from the data management layer. To implement two-tier architectures, we can no longer have dumb terminals on the client side; we require computers that run sophisticated presentation code (and possibly, application logic).

Over the last ten years, a large number of client-server development tools such Microsoft Visual Basic and Sybase Powerbuilder have been developed. These tools permit rapid development of client-server software, contributing to the success of the client-server model, especially the thin-client version.

The thick-client model has several disadvantages when compared to the thin-client model. First, there is no central place to update and maintain the business logic, since the application code runs at many client sites. Second, a large amount of trust is required between the server and the clients. As an exam-- pIe, the DBMS of a bank has to trust the (application executing at an) ATM machine to leave the database in a consistent state. (One way to address this problem is through *stored procedures*, trusted application code that is registered with the DBMS and can be called from SQL statelnents. We discuss stored procedures in detail in Section 6.5.)

A third disadvantage of the thick-client architecture is that it does not scale with the number of clients; it typically cannot handle more than a few hundred clients. The application logic at the client issues SQL queries to the server and the server returns the query result to the client, where further processing takes place. Large query results might be transferred between client and server.

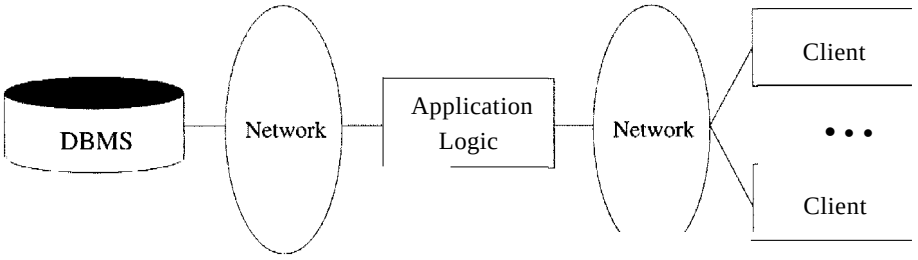


Figure 7.8 A Standard Three-Tier Architecture

(Stored procedures can mitigate this bottleneck.) Fourth, thick-client systems do not scale as the application accesses more and more database systems. Assume there are x different database systems that are accessed by y clients, then there are $x \cdot y$ different connections open at any time, clearly not a scalable solution.

These disadvantages of thick-client systems and the widespread adoption of standard, very thin clients—notably, Web browsers—have led to the widespread use thin-client architectures.

7.5.2 Three-Tier Architectures

The thin-client two-tier architecture essentially separates presentation issues from the rest of the application. The three-tier architecture goes one step further, and also separates application logic from data management:

- Presentation Tier: Users require a natural interface to make requests, provide input, and to see results. The widespread use of the Internet has made Web-based interfaces increasingly popular.
- Middle Tier: The application logic executes here. An enterprise-class application reflects complex business processes, and is coded in a general purpose language such as C++ or Java.
- Data Management Tier: Data-intensive Web applications involve DBMSs, which are the subject of this book.

Figure 7.8 shows a basic three-tier architecture. Different technologies have been developed to enable distribution of the three tiers of an application across multiple hardware platforms and different physical sites. Figure 7.9 shows the technologies relevant to each tier.

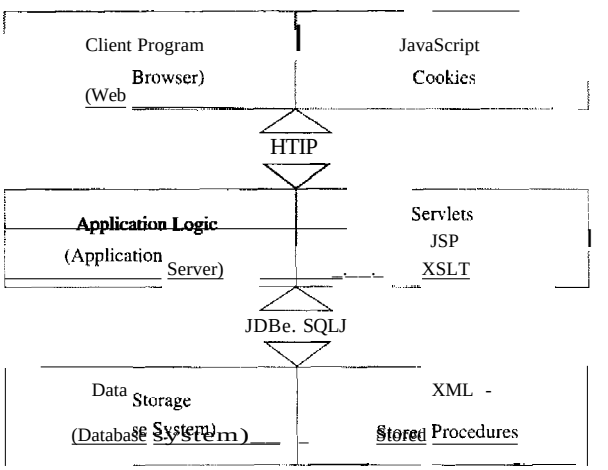


Figure 7.9 Technologies for the Three Tiers

Overview of the Presentation Tier

At the presentation layer, we need to provide forms through which the user can issue requests, and display responses that the middle tier generates. The hypertext markup language (HTML) discussed in Section 7.3 is the basic data presentation language.

It is important that this layer of code be easy to adapt to different display devices and formats; for example, regular desktops versus handheld devices versus cell phones. This adaptivity can be achieved either at the middle tier through generation of different pages for different types of client, or directly at the client through style sheets that specify how the data should be presented. In the latter case, the middle tier is responsible for producing the appropriate data in response to user requests, whereas the presentation layer decides *how* to display that information.

We cover presentation tier technologies, including style sheets, in Section 7.6.

Overview of the Middle Tier

The middle layer runs code that implements the business logic of the application: It controls what data needs to be input before an action can be executed, determines the control flow between multi-action steps, controls access to the database layer, and often assembles dynamically generated HTML pages from database query results.

The middle tier code is responsible for supporting all the different roles involved in the application. For example, in an Internet shopping site implementation, we would like customers to be able to browse the catalog and make purchases, administrators to be able to inspect current inventory, and possibly data analysts to ask summary queries about purchase histories. Each of these roles can require support for several complex actions.

For example, consider the a customer who wants to buy an item (after browsing or searching the site to find it). Before a sale can happen, the customer has to go through a series of steps: She has to add items to her shopping basket, she has to provide her shipping address and credit card number (unless she has an account at the site), and she has to finally confirm the sale with tax and shipping costs added. Controlling the flow among these steps and remembering already executed steps is done at the middle tier of the application. The data carried along during this series of steps might involve database accesses, but usually it is not yet permanent (for example, a shopping basket is not stored in the database until the sale is confirmed).

We cover the middle tier in detail in Section 7.7.

7.5.3 Advantages of the Three-Tier Architecture

The three-tier architecture has the following advantages:

- Heterogeneous Systems: Applications can utilize the strengths of different platforms and different software components at the different tiers. It is easy to modify or replace the code at any tier without affecting the other tiers.
- **Thin** Clients: Clients only need enough computation power for the presentation layer. Typically, clients are Web browsers.
- Integrated Data Access: In many applications, the data must be accessed from several sources. This can be handled transparently at the middle tier, where we can centrally manage connections to all database systems involved.
- Scalabilit,y to Many Clients: Each client is lightweight and all access to the system is through the middle tier. The middle tier can share database connections across clients, and if the middle tier becomes the bottle-neck, we can deploy several servers executing the middle tier code; clients can connect to anyone of these servers, if the logic is designed appropriately. This is illustrated in Figure 7.10, which also shows how the middle tier accesses multiple data sources. Of course, we rely upon the DBMS for each

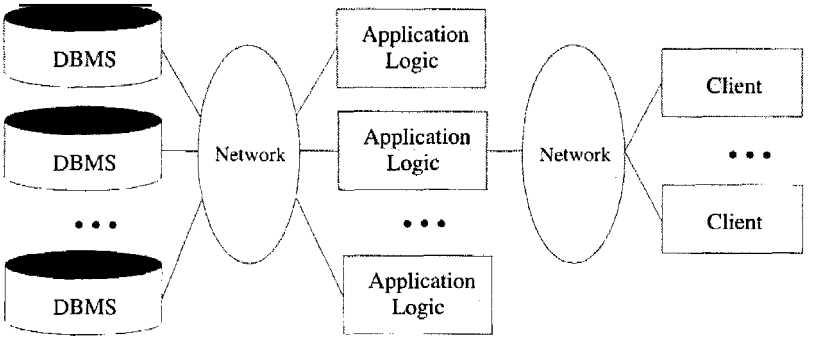


Figure 7.10 Middle-Tier Replication and Access to Multiple Data Sources

data source to be scalable (and this might involve additional parallelization or replication, as discussed in Chapter 22).

- Software Development Benefits: By dividing the application cleanly into parts that address presentation, data access, and business logic, we gain many advantages. The business logic is centralized, and is therefore easy to maintain, debug, and change. Interaction between tiers occurs through well-defined, standardized APIs. Therefore, each application tier can be built out of reusable components that can be individually developed, debugged, and tested.

7.6 THE PRESENTATION LAYER

In this section, we describe technologies for the client side of the three-tier architecture. We discuss HTML forms as a special means of passing arguments from the client to the middle tier (i.e., from the presentation tier to the middle tier) in Section 7.6.1. In Section 7.6.2, we introduce JavaScript, a Java-based scripting language that can be used for light-weight computation in the client tier (e.g., for simple animations). We conclude our discussion of client-side technologies by presenting style sheets in Section 7.6.3. Style sheets are languages that allow us to present the same webpage with different formatting for clients with different presentation capabilities; for example, Web browsers versus cell phones, or even a Netscape browser versus Microsoft's Internet Explorer.

7.6.1 HTML Forms

HTML forms are a common way of communicating data from the client tier to the middle tier. The general format of a form is the following:

```
<FORM ACTION="page.jsp" METHOD="GET" NAME="LoginForm">
```

```
</FORM>
```

A single HTML document can contain more than one form. Inside an HTML form, we can have any HTML tags except another FORM element.

The FORM tag has three important attributes:

- ACTION: Specifies the URI of the page to which the form contents are submitted; if the ACTION attribute is absent, then the URI of the current page is used. In the sample above, the form input would be submited to the page named page. j sp, which should provide logic for processing the input from the form. (We will explain methods for reading form data at the middle tier in Section 7.7.)
- METHOD: The HTTP/1.0 method used to submit the user input from the filled-out form to the webserver. There are two choices, GET and POST; we postpone their discussion to the next section.
- NAME: This attribute gives the form a name. Although not necessary, naming forms is good style. In Section 7.6.2, we discuss how to write client-side programs in JavaScript that refer to forms by name and perform checks on form fields.

Inside HTML forms, the INPUT, SELECT, and TEXTAREA tags are used to specify user input elements; a form can have many elements of each type. The simplest user input element is an INPUT field, a standalone tag with no terminating tag. An example of an INPUT tag is the following:

```
<INPUT TYPE=ltext" NAME="title">
```

The INPUT tag has several attributes. The three most important ones are TYPE, NAME, and VALUE. The TYPE attribute determines the type of the input field. If the TYPE attribute has value **text**, then the field is a text input field. If the TYPE attribute has value password, then the input field is a text field where the entered characters are displayed as stars on the screen. If the TYPE attribute has value **reset**, it is a simple button that resets all input fields within the form to their default values. If the TYPE attribute has value submit, then it is a button that sends the values of the different input fields in the form to the server. Note that **reset** and submit input fields affect the entire form.

The NAME attribute of the INPUT tag specifies the symbolic name for this field and is used to identify the value of this input fi.eld when it is sent to the server. NAME has to be set for INPUT tags of all types except submit and **reset**. In the preceding example, we specified **title** as the NAME of the input field.

The VALUE attribute of an input tag can be used for text or password fields to specify the default contents of the field. For submit or reset buttons, VALUE determines the label of the button.

The form in Figure 7.11 shows two text fields, one regular text input field and one password field. It also contains two buttons, a reset button labeled 'Reset Values' and a submit button labeled 'Log on.' Note that the two input fields are named, whereas the reset and submit button have no NAME attributes.

```
<FORM ACTION="page.jsp" METHoD="GET" NAME="LoginForm">
    <INPUT TYPE="text" NAME="username" VALUE=" Joe"><P>
    <INPUT TYPE="password" NAME="p&ssword"><P>
    <INPUT TYPE="reset" VALUE="Reset Values"><P>
    <INPUT TYPE="submit" VALUE="Log on">
</FoRM>
```

Figure 7.11 HTl'vlL Form with Two Text Fields and Two Buttons

HTML forms have other ways of specifying user input, such as the aforementioned TEXTAREA and SELECT tags; we do not discuss them.

Passing Arguments to Server-Side Scripts

As mentioned at the beginning of Section 7.6.1, there are two different ways to submit HTML Form data to the webserver. If the method GET is used, then the contents of the form are assembled into a query URI (as discussed next) and sent to the server. If the method POST is used, then the contents of the form are encoded as in the GET method, but the contents are sent in a separate data block instead of appending them directly to the URI. Thus, in the GET method the form contents are directly visible to the user as the constructed URI, whereas in the POST method, the form contents are sent inside the HTTP request message body and are not visible to the user.

Using the GET method gives users the opportunity to bookmark the page with the constructed URI and thus directly jump to it in subsequent sessions; this is not possible with the POST method. The choice of GET versus POST should be determined' by the application and its requirements.

Let us look at the encoding of the URI when the GET method is used. The encoded URI has the following form:

action?name1=value1&name2=value2&name3=value3

The action is the URI specified in the ACTION attribute to the FORM tag, or the current document URI if no ACTION attribute was specified. The 'name=value' pairs are the user inputs from the INPUT fields in the form. For form INPUT fields where the user did not input anything, the name is stil present with an empty value (name=). As a concrete example, consider the password submission form at the end of the previous section. Assume that the user inputs 'John Doe' as username, and 'secret' as password. Then the request URI is:

page.jsp?username=J01111+Doe&password=secret

The user input from forms can contain general ASCII characters, such as the space character, but URIs have to be single, consecutive strings with no spaces. Therefore, special characters such as spaces, '=', and other unprintable characters are encoded in a special way. To create a URI that has form fields encoded, we perform the following three steps:

1. Convert all special characters in the names and values to '%xyz,' where 'xyz' is the ASCII value of the character in hexadecimal. Special characters include =, &, %, +, and other unprintable characters. Note that we could encode *all* characters by their ASCII value.
2. Convert all space characters to the '+' character.
3. Glue corresponding names and values from an individual HTML INPUT tag together with '=' and then paste name-value pairs from different HTML INPUT tags together using'&' to create a request URI of the form: action?namel=value1&name2=value2&name3=value3

Note that in order to process the input elements from the HTML form at the middle tier, we need the ACTION attribute of the FORM tag to point to a page, script, or program that will process the values of the form fields the user entered. We discuss ways of receiving values from form fields in Sections 7.7.1 and 7.7.3.

7.6.2 JavaScript

JavaScript is a scripting language at the client tier with which we can add programs to webpages that run directly at the client (Le., at the machine running the Web browser). JavaScript is often used for the following types of computation at the client:

- Browser Detection: JavaScript can be used to detect the browser type and load a browser-specific page.
- Form Validation: JavaScript is used to perform simple consistency checks on form fields. For example, a JavaScript program might check whether a

form input that asks for an email address contains the character '@,' or if all required fields have been input by the user.

- **Browser Control:** This includes opening pages in customized windows; examples include the annoying pop-up advertisements that you see at many websites, which are programmed using JavaScript.

JavaScript is usually embedded into an HTML document with a special tag, the SCRIPT tag. The SCRIPT tag has the attribute LANGUAGE, which indicates the language in which the script is written. For JavaScript, we set the language attribute to JavaScript. Another attribute of the SCRIPT tag is the SRC attribute, which specifies an external file with JavaScript code that is automatically embedded into the HTML document. Usually JavaScript source code files use a '.js' extension. The following fragment shows a JavaScript file included in an HTML document:

```
<SCRIPT LANGUAGE=" JavaScript" SRC="validateForm.js"> </SCRIPT>
```

The SCRIPT tag can be placed inside HTML comments so that the JavaScript code is not displayed verbatim in Web browsers that do not recognize the SCRIPT tag. Here is another JavaScipt code example that creates a pop-up box with a welcoming message. We enclose the JavaScipt code inside HTML comments for the reasons just mentioned.

```
<SCRIPT LANGUAGE=" JavaScript" >
<!--
    alert("Welcome to our bookstore");
//-->
</SCRIPT>
```

JavaScript provides two different commenting styles: single-line comments that start with the '//' character, and multi-line comments starting with '/*' and ending with ,*/' characters.[1]

JavaScript has variables that can be numbers, boolean values (true or false), strings, and some other data types that we do not discuss. Global variables have to be declared in advance of their usage with the keyword var, and they can be used anywhere inside the HTML documents. Variables local to a JavaScript function (explained next) need not be declared. Variables do not have a fixed type, but implicitly have the type of the data to which they have been assigned.

---

1 Actually, '<!--' also marks the start of a single-line comment, which is why we did not have to mark the HTML starting cormnent '<!--' in the preceding example using JavaScript comment notation. In contrast, the HTML closing comment "-->" has to be commented out in JavaScript as it is interpreted otherwise.

JavaScript has the usual assignment operators (=, + =, etc.), the usual arithmetic operators (+, -, *, /, %), the usual comparison operators (==, ! =, >=, etc.), and the usual boolean operators (&& for logical AND, || for logical OR, and! for negation). Strings can be concatenated using the '+' character. The type of an object determines the behavior of operators; for example 1+1 is 2, since we are adding numbers, whereas "1"+"1" is "11," since we are concatenating strings. JavaScript contains the usual types of statements, such as assignments, conditional statements (if Ccondition) {statements;} else {statements; }), and loops (for-loop, do-while, and while-loop).

JavaScript allows us to create functions using the function keyword: function f Cargl, arg2) {statements;}. We can call functions from JavaScript code, and functions can return values using the keyword return.

We conclude this introduction to JavaScript with a larger example of a JavaScript function that tests whether the login and password fields of a HTML form are not empty. Figure 7.12 shows the JavaScript function and the HTML form. The JavaScript code is a function called testLoginEmptyO that tests whether either of the two input fields in the form named LoginForm is empty. In the function testLoginEmpty, we first use variable loginForm to refer to the form LoginForm using the implicitly defined variable document, which refers to the current HTML page. (JavaScript has a library of objects that are implicitly defined.) We then check whether either of the strings loginForm. userif. value or loginForm. password. value is empty.

The function testLoginEmpty is checked within a form event handler. An event **handler** is a function that is called if an event happens on an object in a webpage. The event handler we use is onSubmit, which is called if the submit button is pressed (or if the user presses return in a text field in the form). If the event handler returns true, then the form contents are submitted to the server, otherwise the form contents are not submitted to the server.

JavaScript has functionality that goes beyond the basics that we explained in this section; the interested reader is referred to the bibliographic notes at the end of this chapter.

7.6.3 Style Sheets

Different clients have different displays, and we need correspondingly different ways of displaying the same information. For example, in the simplest case, we might need to use different font sizes or colors that provide high-contrast on a black-and-white screen. As a more sophisticated example, we might need to re-arrange objects on the page to accommodate small screens in personal

```
<SCRIPT LANGUAGE="JavaScript">
<!--
function testLoginEmpty()
{
    10ginForm = document.LoginForm
    if ((loginForm.userid.value == "") ||
            (loginFonn.password.value == "")) {
        alert(,Please enter values for userid and password.');
        return false;
    }
    else
        return true;
}
//-->
</SCRIPT>
<Hi ALIGN = "CENTER">Barns and Nobble Internet Bookstore</Hi>
<H3 ALIGN = "CENTER">Please enter your userid and password:</H3>
<FORM NAME = "LoginForm" METHOD="POST"
        ACTION="TableOfContents.jsp"
        onSubmit="return testLoginEmptyO">
    Userid: <INPUT TYPE="TEXT" NAME="userid"><P>
    Password: <INPUT TYPE="PASSWORD" NAME="password"><P>
    <INPUT TYPE="SUBMIT" VALUE="Login" NAME="SUBMIT">
    <INPUT TYPE="RESET" VALUE="Clear Input" NAME="RESET">
</FORM>
```

Figure 7.12 Form Validation with JavaScript

digital assistants (PDAs). As another example, we might highlight different infonnation to focus on some important part of the page. A style sheet is a method to adapt the same document contents to different presentation formats. A style sheet contains instructions that tell a Web browser (or whatever the client uses to display the webpage) how to translate the data of a document into a presentation that is suitable for the client's display.

Style sheets separate the transformative aspect of the page from the rendering aspects of the page. During transformation, the objects in the XML document are rearranged to form a different structure, to omit parts of the XML document, or to merge two different XML documents into a single document. During rendering, we take the existing hierarchical structure of the XML document and format the document according to the user's display device.

```
BODY {BACKGROUND-COLOR: yellow}
Hi {FONT-SIZE: 36pt}
H3 {COLOR: blue}
P {MARGIN-LEFT: 50px; COLOR: red}
```

Figure 7.13 An Example Style sheet

The use of style sheets has many advantages. First, we can reuse the same document many times and display it differently depending on the context. Second, we can tailor the display to the reader's preference such as font size, color style, and even level of detail. Third, we can deal with different output formats, such as different output devices (laptops versus cell phones), different display sizes (letter versus legal paper), and different display media (paper versus digital display). Fourth, we can standardize the display format within a corporation and thus apply style sheet conventions to documents at any time. Further, changes and improvements to these display conventions can be managed at a central place.

There are two style sheet languages: XSL and ess. ess was created for HTML with the goal of separating the display characteristics of different formatting tags from the tags themselves. XSL is an extension of ess to arbitrary XML docurnents; besides allowing us to define ways of formatting objects, XSL contains a transformation language that enables us to rearrange objects. The target files for ess are HTML files, whereas the target files for XSL are XML files.

Cascading Style Sheets

A Cascading Style Sheet (CSS) defines how to display HTML elements. (In Section 7.13, we introduce a more general style sheet language designed for XML documents.) Styles are normally stored in style sheets, which are files that contain style definitions. Many different HTML documents, such as all documents in a website, can refer to the same ess. Thus, we can change the format of a website by changing a single file. This is a very convenient way of changing the layout of many webpages at the same time, and a first step toward the separation of content from presentation.

An example style sheet is shown in Figure 7.13. It is included into an HTML file with the following line:

```
<LINK REL="style sheet" TYPE="text/css" HREF="books.css" />
```

Each line in a CSS sheet consists of three parts; a selector, a property, and a value. They are syntactically arranged in the following way:

selector {property: value}

The selector is the element or tag whose format we are defining. The property indicates the tag's attribute whose value we want to set in the style sheet, and the property is the actual value of the attribute. As an example, consider the first line of the example style sheet shown in Figure 7.13:

BODY {BACKGROUND-COLOR: yellow}

This line has the same effect as changing the HTML code to the following:

<BODY BACKGROUND-COLOR="yellow">.

The value should always be quoted, as it could consist of several words. More than one property for the same selector can be separated by semicolons as shown in the last line of the example in Figure 7.13:

P {MARGIN-LEFT: 50px; COLOR: red}

Cascading style sheets have an extensive syntax; the bibliographic notes at the end of the chapter point to books and online resources on CSSs.

XSL

XSL is a language for expressing style sheets. An XSL style sheet is, like CSS, a file that describes how to display an XML document of a given type. XSL shares the functionality of CSS and is compatible with it (although it uses a different syntax).

The capabilities of XSL vastly exceed the functionality of CSS. XSL contains the XSL Transformation language, or XSLT, a language that allows us to transform the input XML document into a XML document with another structure. For example, with XSLT we can change the order of elements that we are displaying (e.g.; by sorting them), process elements more than once, suppress elements in one place and present them in another, and add generated text to the presentation.

XSL also contains the XML Path Language (XPath), a language that allows us to refer to parts of an XML document. We discuss XPath in Section

27. XSL also contains XSL Formatting Object, a way of formatting the output of an XSL transformation.

7.7 THE MIDDLE TIER

In this section, we discuss technologies for the middle tier. The first generation of middle-tier applications were stand-alone programs written in a general-purpose programming language such as C, C++, and Perl. Programmers quickly realized that interaction with a stand-alone application was quite costly; the overheads include starting the application every time it is invoked and switching processes between the webserver and the application. Therefore, such interactions do not scale to large numbers of concurrent users. This led to the development of the application server, which provides the run-time environment for several technologies that can be used to program middle-tier application components. Most of today's large-scale websites use an application server to run application code at the middle tier.

Our coverage of technologies for the middle tier mirrors this evolution. We start in Section 7.7.1 with the Common Gateway Interface, a protocol that is used to transmit arguments from HTML forms to application programs running at the middle tier. We introduce application servers in Section 7.7.2. We then describe technologies for writing application logic at the middle tier: Java servlets (Section 7.7.3) and Java Server Pages (Section 7.7.4). Another important functionality is the maintenance of state in the middle tier component of the application as the client component goes through a series of steps to complete a transaction (for example, the purchase of a market basket of items or the reservation of a flight). In Section 7.7.5, we discuss Cookies, one approach to maintaining state.

7.7.1 CGI: The Common Gateway Interface

The Common Gateway Interface connects HTML forms with application programs. It is a protocol that defines how arguments from forms are passed to programs at the server side. We do not go into the details of the actual CGI protocol since libraries enable application programs to get arguments from the HTML fonn; we shortly see an example in a CGI program. Programs that communicate with the webserver via CGI are often called CGI scripts, since many such application programs were written in a scripting language such as Perl.

As an example of a program that interfaces with an HTML form via CGI, consider the sample page shown in Figure 7.14. This webpage contains a form where a user can fill in the name of an author. If the user presses the 'Send

```
<HTML><HEAD><TITLE>The Database Bookstore</TITLE></HEAD>
<BODY>
<FORM ACTION="find_books.cgi" METHOD=POST>
    Type an author name:
    <INPUT TYPE="text" NAME=lauthorName"
           SIZE=30 MAXLENGTH=50>
    <INPUT TYPE="submitil value="Send it">
    <INPUT TYPE=lreset" VALUE="Clear form">
</FORM>
</BODY></HTML>
```

Figure 7.14 A Sample Web Page Where Form Input Is Sent to a CGI Script

it' button, the Perl script 'findBooks.cgi' shown in Figure 7.14 is executed as a separate process. The CGl protocol defines how the communication between the form and the script is performed. Figure 7.15 illustrates the processes created when using the CGl protocol.

Figure 7.16 shows the example CGl script, written in Perl. We omit error-checking code for simplicity. Perl is an interpreted language that is often used for CGl scripting and many Perl libraries, called **modules,** provide high-level interfaces to the CGl protocol. \Ve use one such library, called the **DBI library,** in our example. The CGI module is a convenient collection of functions for creating CGl scripts. In part 1 of the sample script, we extract the argument of the HTML form that is passed along from the client as follows:

```
$authorName = $dataIn- >paramCauthorName');
```

Note that the parameter name authorName was used in the form in Figure 7.14 to name the first input field. Conveniently, the CGl protocol abstracts the actual implementation of how the webpage is returned to the Web browser; the webpage consists simply of the output of our program, and we start assembling the output HTML page in part 2. Everything the script writes in print-statements is part of the dynamically constructed webpage returned to the browser. We finish in part 3 by appending the closing format tags to the resulting page.

7.7.2 Application Servers

Application logic can be enforced through server-side programs that are invoked using the CGl protocol. However, since each page request results in the creation of a new process, this solution does not scale well to a large number of simultaneous requests. This performance problem led to the development of

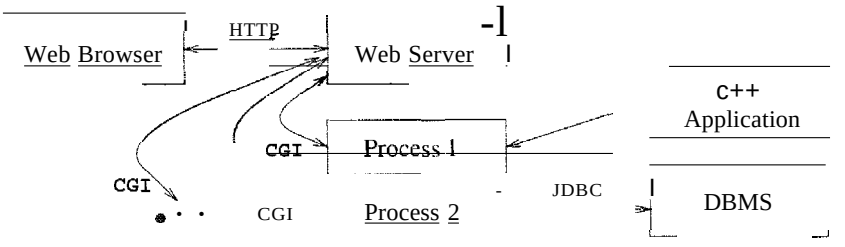


Figure 7.15 Process Structure with eGI Scripts

```
#!/usr/bin/perl
use CGI;

### part 1
$dataln = new CGI;
$dataIn-¿header();
$authorName = $dataln-l,param('authorName');

### part 2
print (II<HTML><TITLE>Argument passing test</TITLE> II) ;
print (II The user passed the following argument: II) ;
print (II authorName: ", $authorName);

### part 3
print ("</HTML>");
exit;
```

Figure 7.16 A Simple Perl Script

specialized programs called application servers. An application server maintains a pool of threads or processes and uses these to execute requests. Thus, it avoids the startup cost of creating a new process for each request.

Application servers have evolved into flexible middle-tier packages that provide many functions in addition to eliminating the process-creation overhead. They facilitate concurrent access to several heterogeneous data sources (e.g., by providing JDBC drivers), and provide session management services. Often, business processes involve several steps. Users expect the system to maintain continuity during such a multistep session. Several session identifiers such as cookies, URI extensions, and hidden fields in HTML forms can be used to identify a session. Application servers provide functionality to detect when a session starts and ends and keep track of the sessions of individual users. They

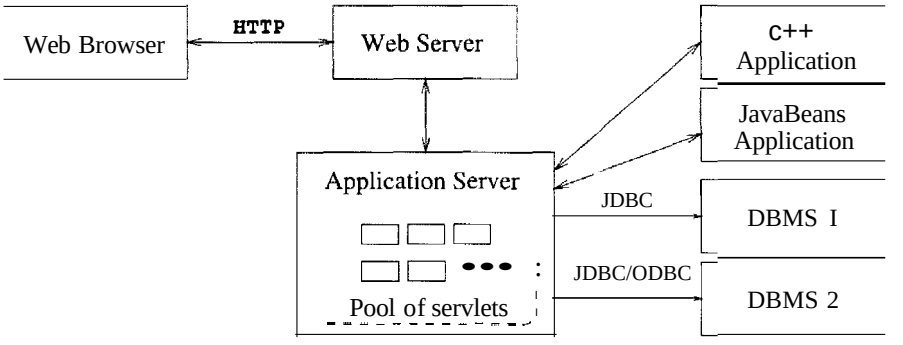


Figure 7.17 Process Structure in the Application Server Architecture

also help to ensure secure database access by supporting a general user-id mechanism. (For more on security, see Chapter 21.)

A possible architecture for a website with an application server is shown in Figure 7.17. The client (a Web browser) interacts with the webserver through the HTTP protocol. The webserver delivers static HTML or XML pages directly to the client. To assemble dynamic pages, the webserver sends a request to the application server. The application server contacts one or more data sources to retrieve necessary data or sends update requests to the data sources. After the interaction with the data sources is completed, the application server assembles the webpage and reports the result to the webserver, which retrieves the page and delivers it to the client.

The execution of business logic at the webserver's site, server-side processing, has become a standard model for implementing more complicated business processes on the Internet. There are many different technologies for server-side processing and we only mention a few in this section; the interested reader is referred to the bibliographic notes at the end of the chapter.

7.7.3 Servlets

Java servlets are pieces of Java code that run on the middle tier, in either webservers or application servers. There are special conventions on how to read the input from the user request and how to write output generated by the servlet. Servlets are truly platform-independent, and so they have become very popular with Web developers.

Since servlets are Java programs, they are very versatile. For example, servlets can build webpages, access databases, and maintain state. Servlets have access

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

pUblic class ServletTemplate extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        PrintWriter out = response.getWriter();
        // Use 'out' to send content to browser
        out.println("Hello World");
    }
}
```

Figure 7.18 Servlet Template

to all Java APIs, including JDBC. All servlets must implement the Servlet interface. In most cases, servlets extend the specific HttpServlet class for servers that communicate with clients via HTTP. The HttpServlet class provides methods such as doGet and doPost to receive arguments from HTML forms, and it sends its output back to the elient via HTTP. Servlets that communicate through other protocols (such as ftp) need to extend the class GenericServlet.

Servlets are compiled Java classes executed and maintained by a servlet container. The servlet container manages the lifespan of individual servlets by creating and destroying them. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by webservers. For such applications, there is a useful library of HTTP-specific servlet classes.

Servlets usually handle requests from HTML forms and maintain state between the client and the server. We discuss how to maintain state in Section 7.7.5. A template of a generic servlet structure is shown in Figure 7.18. This simple servlet just outputs the two words "Hello World," but it shows the general structure of a full-fledged servlet. The request object is used to read HTML form data. The response object is used to specify the HTTP response status code and headers of the HTTP response. The object out is used to compose the content that is returned to the client.

Recall that HTTP sends back the status line, a header, a blank line, and then the context. Right now our servlet just returns plain text. We can extend our servlet by setting the content type to HTML, generating HTML as follows:

```
PrinfWriter out = response.get\Vriter();
String docType =
    "<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 " +
    "Transitional//EN"> \n";
out.println(docType +
    "<HTML>\n" +
    "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
    "<BODY>\n" +
    "<Hl>Hello WWW</Hl>\n" +
    "</BODY></HTML>");
```

What happens during the life of a servlet? Several methods are called at different stages in the development of a servlet. When a requested page is a servlet, the webserver forwards the request to the servlet container, which creates an instance of the servlet if necessary. At servlet creation time, the servlet container calls the init () method, and before deallocating the servlet, the servlet container calls the servlet's destroyO method.

When a servlet container calls a servlet because of a requested page, it starts with the service () method, whose default behavior is to call one of the following methods based on the HTTP transfer method: service () calls doGet 0 for a HTTP GET request, and it calls doPost () for a HTTP POST request. This automatic dispatching allows the servlet to perform different tasks on the request data depending on the HTTP transfer method. Usually, we do not override the service () method, unless we want to program a servlet that handles both HTTP POST and HTTP GET requests identically.

We conclude our discussion of servlets with an example, shown in Figure 7.19, that illustrates how to pass arguments from an HTML form to a servlet.

7.7.4 JavaServer Pages

In the previous section, we saw how to use Java programs in the middle tier to encode application logic and dynamically generate webpages. If we needed to generate HTML output, we wrote it to the out object. Thus, we can think about servlets as Java code embodying application logic, with embedded HTML for output.

JavaServer pages (JSPs) interchange the roles of output and application logic. JavaServer pages are written in HTML with servlet-like code embedded in special HT1VIL tags. Thus, in comparison to servlets, JavaServer pages are better suited to quickly building interfaces that have some logic inside, whereas servlets are better suited for complex application logic.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ReadUserName extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType('j textjhtml'j);
        PrintWriter out = response.getWriter();

        out.println("<BODY>\n" +
            "<Hi ALIGN=CENTER> Username: </Hi>\n" +
            "<UL>\n" +
            " <LI>title: "
            + request.getParameter("userid") + "\n" +
            + request.getParameter("password'j) + "\n" +
            1</UL>\n" +
            1</BODY></HTML>")j
    }

    public void doPost(HttpServletRequest request,
            HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Figure 7.19 Extracting the User Name and Password From a Form

While there is a big difference for the programmer, the middle tier handles JavaServer pages in a very simple way: They are usually compiled into a servlet, which is then handled by a servlet container analogous to other servlets.

The code fragment in Figure 7.20 shows a simple JSP example. In the middle of the HTML code, we access information that was passed from a form.

```
<!DOCTYPE HTML PUBLIC 11_//W3C//DTD HTML 4.0
        Transitional//EN"">
<HTML>
<HEAD><TITLE>Welcome to Barnes and Nobble</TITLE></HEAD>
<BODY>
    <Hl>Welcome back!</Hl>
    <% String name="NewUser"";
        if (request.getParameter(lIusernamell) != null) {
            name=request.getParameter("username");
        }
    %>
    You are logged on as user <%=name%>
    <P>
    Regular HTML for all the rest of the on-line store's webpage.
</BODY>
</HTML>
```

Figure 7.20 Reading Form Parameters in JSP

7.7.5 **Maintaining State**

As discussed in previous sections, there is a need to maintain a user's state across different pages. As an example, consider a user who wants to make a purchase at the Barnes and Nobble website. The user must first add items into her shopping basket, which persists while she navigates through the site. Thus, we use the notion of state mainly to remember information as the user navigates through the site.

The HTTP protocol is stateless. We call an interaction with a webserver stateless if no inforination is retained from one request to the next request. We call an interaction with a webserver stateful, or we say that state is maintained, if some memory is stored between requests to the server, and different actions are taken depending on the contents stored.

In our example of Barnes and Nobble, we need to maintain the shopping basket of a user. Since state is not encapsulated in the HTTP protocol, it has to be maintained either at the server or at the client. Since the HTTP protocol is stateless by design, let us review the advantages and disadvantages of this design decision. First, a stateless protocol is easy to program and use, and it is great for applications that require just retrieval of static information. In addition, no extra memory is used to maintain state, and thus the protocol itself is very efficient. On the other hand, without some additional mechanism at the presentation tier and the middle tier, we have no record of previous requests, and we cannot program shopping baskets or user logins.

Since we cannot maintain state in the HTTP protocol, where should we mtain-tain state? There are basically two choices. We can maintain state in the middle tier, by storing information in the local main memory of the application logic, or even in a database system. Alternatively, we can maintain state on the client side by storing data in the form of a *cookie*. We discuss these two ways of maintaining state in the next two sections.

Maintaining State at the Middle Tier

At the middle tier, we have several choices as to *where* we maintain state. First, we could store the state at the bottom tier, in the database server. The state survives crashes of the system, but a database access is required to query or update the state, a potential performance bottleneck. An alternative is to store state in main memory at the middle tier. The drawbacks are that this information is volatile and that it might take up a lot of main memory. We can also store state in local files at the middle tier, as a compromise between the first two approaches.

A rule of thumb is to use state maintenance at the middle tier or database tier only for data that needs to persist over many different user sessions. Examples of such data are past customer orders, click-stream data recording a user's movement through the website, or other permanent choices that a user makes, such as decisions about personalized site layout, types of messages the user is willing to receive, and so on. As these examples illustrate, state information is often centered around users who interact with the website.

Maintaining State at the Presentation Tier: Cookies

Another possibility is to store state at the presentation tier and pass it to the middle tier with every HTTP request. We essentially work around around the statelessness of the HTTP protocol by sending additional information with every request. Such information is called a cookie.

A **cookie** is a collection of *(name, value)*-pairs that can be manipulated at the presentation and middle tiers. Cookies are easy to use in Java servlets and Java8erver Pages and provide a simple way to make non-essential data persistent at the client. They survive several client sessions because they persist in the browser cache even after the browser is closed.

One disadvantage of cookies is that they are often perceived as as being invasive, and many users disable cookies in their Web browser; browsers allow users to prevent cookies from being saved on their machines. Another disadvantage is that the data in a cookie is currently limited to 4KB, but for most applications this is not a bad limit.

We can use cookies to store information such as the user's shopping basket, login information, and other non-permanent choices made in the current session.

Next, we discuss how cookies can be manipulated from servlets at the middle tier.

The Servlet Cookie API

A cookie is stored. in a small text file at the client and. contains *(name, value)*-pairs, where both name and value are strings. We create a new cookie through the Java Cookie class in the middle tier application code:

```
Cookie cookie = new Cookie("username","guest");
cookie.setDomain("www.bookstore.com");
cookie.set8ecure(false);                    // no 88L required
cookie.setMaxAge(60*60*24*7*31);      // one month lifetime
response.addCookie(cookie);
```

Let us look at each part of this code. First, we create a new Cookie object with the specified *(name, value)*-pair. Then we set attributes of the cookie; we list some of the most common attributes below:

- setDomain and getDomain: The domain specifies the website that will receive the cookie. The default value for this attribute is the domain that created the cookie.
- setSecure and getSecure: If this flag is true, then the cookie is sent only if we are llsing a secure version of the HTTP protocol, such as 88L.
- setMaxAge and getMaxAge: The MaxAge attribute determines the lifetime of the cookie in seconds. If the value of MaxAge is less than or equal to zero, the cookie is deleted when the browser is closed.

- setName and getName: We did not use these functions in our code fragment; they allow us to Ilame the cookie.
- setValue and getValue: These functions allow us to set and read the value of the cookie.

The cookie is added to the request object within the Java servlet to be sent to the client. Once a cookie is received from a site (www.bookstore.comin this example), the client's Web browser appends it to all HTTP requests it sends to this site, until the cookie expires.

We can access the contents of a cookie in the middle-tier code through the request object getCookies 0 method, which returns an array of Cookie objects. The following code fragment reads the array and looks for the cookie with name 'username.'

```
Cookie[] cookies = request.getCookiesO;
String theUser;
for(int i=O; i < cookies.length; i++) {
    Cookie cookie = cookies[i];
    if (cookie.getNameO.equals("username"))
        theUser = cookie.getValueO;
}
```

A simple test can be used to check whether the user has turned off cookies: Send a cookie to the user, and then check whether the request object that is returned still contains the cookie. Note that a cookie should never contain an unencrypted password or other private, unencrypted data, as the user can easily inspect, modify, and erase any cookie at any time, including in the middle of a session. The application logic needs to have sufficient consistency checks to ensure that the data in the cookie is valid.

7.8 CASE STUDY: THE INTERNET BOOK SHOP

DBDudes now moves on to the implementation of the application layer and considers alternatives for connecting the DBMS to the World Wide Web.

DBDudes begifls by considering session management. For example, users who log in to the site, browse the catalog, and select books to buy do not want to re-enter their cllstomer identification numbers. Session management has to extend to the whole process of selecting books, adding them to a shopping cart, possibly removing books from the cart, and checking out and paying for the books.

DBDudes then considers whether webpages for books should be static or dynamic. If there is a static webpage for each book, then we need an extra database field in the Books relation that points to the location of the file. Even though this enables special page designs for different books, it is a very labor-intensive solution. DBDudes convinces B&N to dynamically assemble the webpage for a book from a standard template instantiated with information about the book in the Books relation. Thus, DBDudes do not use static HTML pages, such as the one shown in Figure 7.1, to display the inventory.

DBDudes considers the use of XML as a data exchange format between the database server and the middle tier, or the middle tier and the client tier. Representation of the data in XML at the middle tier as shown in Figures 7.2 and 7.3 would allow easier integration of other data sources in the future, but B&N decides that they do not anticipate a need for such integration, and so DBDudes decide not to use XML data exchange at this time.

DBDudes designs the application logic as follows. They think that there will be four different webpages:

- index.jsp: The home page of Barns and Nobble. This is the main entry point for the shop. This page has search text fields and buttons that allow the user to search by author name, ISBN, or title of the book. There is also a link to the page that shows the shopping cart, cart.jsp.
- login.jsp: Allows registered users to log in. Here DBDudes use an HTML form similar to the one displayed in Figure 7.11. At the middle tier, they use a code fragment similar to the piece shown in Figure 7.19 and JavaServerPages as shown in Figure 7.20.
- search.jsp: Lists all books in the database that match the search condition specified by the user. The user can add listed items to the shopping basket; each book has a button next to it that adds it. (If the item is already in the shopping basket, it increments the quantity by one.) There is also a counter that shows the total number of items currently in the shopping basket. (DBDucles makes a note that that a quantity of five for a single item in the shopping basket should indicate a total purchase quantity of five as well.) The search.jsp page also contains a button that directs the user to cart.jsp.
- cart.jsp: Lists all the books currently in the shopping basket. The listing should include all items in the shopping basket with the product name, price, a text box for the quantity (which the user can use to change quantities of items), and a button to remove the item from the shopping basket. This page has three other buttons: one button to continue shopping (which returns the user to page index.jsp), a second button to update the shop-

ping basket with the altered quantities from the text boxes, and a third button to place the order, which directs the user to the page confirm.jsp.

- coni irm. j sp: Lists the complete order so far and allows the user to enter his or her contact information or customer ID. There are two buttons on this page: one button to cancel the order and a second button to submit the final order. The cancel button ernpties the shopping basket and returns the user to the home page. The submit button updates the database with the new order, empties the shopping basket, and returns the user to the home page.

DBDudes also considers the use of JavaScript at the presentation tier to check user input before it is sent to the middle tier. For example, in the page login. j sp, DBDudes is likely to write JavaScript code similar to that shown in Figure 7.12.

This leaves DBDudes with one final decision: how to connect applications to the DBMS. They consider the two main alternatives presented in Section 7.7: CGI scripts versus using an application server infrastructure. If they use CGI scripts, they would have to encode session management logic-not an easy task. If they use an application server, they can make use of all the functionality that the application server provides. Therefore, they recommend that B&N implement server-side processing using an application server.

B&N accepts the decision to use an application server, but decides that no code should be specific to any particular application server, since B&N does not want to lock itself into one vendor. DBDudes agrees proceeds to build the following pieces:

- DBDudes designs top level pages that allow customers to navigate the website as well as various search forms and result presentations.
- Assuming that DBDudes selects a Java-based application server, they have to write Java servlets to process form-generated requests. Potentially, they could reuse existing (possibly commercially available) JavaBeans. They can use JDBC as a database interface; exarnples of JDBC code can be found in Section 6.2. Instead of prograrnming servlets, they could resort to Java Server Pages and annotate pages with special .JSP markup tags.
- DBDudes select an application server that uses proprietary markup tags, but due to their arrangement with B&N, they are not allowed to use such tags in their code.

For completeness, we remark that if DBDudes and B&N had agreed to use CGr scripts, DBDucles would have had the following tasks:

- Create the top level HTML pages that allow users to navigate the site and various forms that allow users to search the catalog by ISBN, author name, or title. An example page containing a search form is shown in Figure 7.1. In addition to the input forms, DBDudes must develop appropriate presentations for the results.
- Develop the logic to track a customer session. Relevant information must be stored either at the server side or in the customer's browser using cookies.
- Write the scripts that process user requests. For example, a customer can use a form called 'Search books by title' to type in a title and search for books with that title. The CGI interface communicates with a script that processes the request. An example of such a script written in Perl using the DBI library for data access is shown in Figure 7.16.

Our discussion thus far covers only the customer interface, the part of the website that is exposed to B&N's customers. DBDudes also needs to add applications that allow the employees and the shop owner to query and access the database and to generate summary reports of business activities.

Complete files for the case study can be found on the webpage for this book.

7.9 REVIEW QUESTIONS

Answers to the review questions can be found in the listed sections.

- What are URIs and URLs? (Section 7.2.1)
- How does the HTTP protocol work? What is a stateless protocol? (Section 7.2.2)
- Explain the main concepts of HTML. Why is it used only for data presentation and not data exchange? (Section 7.3)
- What are some shortcomings of HTML, and how does XML address them? (Section 7.4)
- What are the main components of an XML document? (Section 7.4.1)
- Why do we have XML DTDs? What is a well-formed XML document? What is a valid XML document? Give an example of an XML document that is valid but not well-formed, and vice versa. (Section 7.4.2)
- 'What is the role of domain-specific DTDs? (Section 7.4.3)
- What is a three-tier architecture? 'What advantages does it offer over single-tier and two-tier architectures? Give a short overview of the functionality at each of the three tiers. (Section 7.5)

- Explain how three-tier architectures address each of the following issues of database-backed Internet applications: heterogeneity, thin clients, data integration, scalability, software development. (Section 7.5.3)
- Write an HTML form. Describe all the components of an HTML form. (Section 7.6.1)
- What is the difference between the HTML GET and POST methods? How does URI encoding of an HTML form work? (Section 7.11)
- What is JavaScript used for? Write a JavaScipt function that checks whether an HTML form element contains a syntactically valid email address. (Section 7.6.2)
- What problem do style sheets address? What are the advantages of using style sheets? (Section 7.6.3)
- What are Cascading Style Sheets? Explain the components of Cascading Style Sheets. What is XSL and how it is different from CSS? (Sections 7.6.3 and 7.13)
- What is CGl and what problem does it address? (Section 7.7.1)
- What are application servers and how are they different from webservers? (Section 7.7.2)
- What are servlets? How do servlets handle data from HTML forms? Explain what happens during the lifetime of a servlet. (Section 7.7.3)
- What is the difference between servlets and JSP? When should we use servlets and when should we use JSP? (Section 7.7.4)
- Why do we need to maintain state at the middle tier? What are cookies? How does a browser handle cookies? How can we access the data in cookies from servlets? (Section 7.7.5)

EXERCISES

Exercise 7.1 Briefly answer the following questions:

1. Explain the following terms and describe what they are used for: HTML, URL, XML, Java, JSP, XSL, XSLT, servlet, cookie, HTTP, ess, DTD.
2. What is eGl? Why was eGI introduced? What are the disadvantages of an architecture using eel scripts?
3. \Vhat is the difference between a webserver and an application server? What fUl1cionality do typical application servers provide?
4. When is an XML document well-formed? When is an XML document valid?

Exercise 7.2 Briefly answer the following questions about the HTTP protocol: