5 SQL: QUERIES, PROGRAMMING, TRIGGERS

> What men or gods are these? What maidens loth?
> What mad pursuit? What struggle to escape?
> What pipes and timbrels? What wild ecstasy?
>
> —John Keats, *Ode on a Grecian Urn*

> What is the average salary in the Toy department?
>
> —Anonymous SQL user

Structured Query Language (SQL) is the most widely used commercial relational database language. It was originally developed at IBM in the SEQUEL-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. Our presentation follows the current ANSI/ISO standard for SQL, which is called SQL-92. We also discuss some important extensions in the new standard, SQL:1999. While not all DBMS products support the full SQL-92 standard yet, vendors are working toward this goal and most products already support the core features. The SQL language has several aspects to it:

- **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. The DDL also provides commands for specifying *access rights* or *privileges* to tables and views. Although the standard does not discuss indexes, commercial implementations also provide commands for creating and deleting indexes. We covered the DDL features of SQL in Chapter 3.
- **The Data Manipulation Language (DML):** This subset of SQL allows users to pose queries and to insert, delete, and modify rows. We covered DML commands to insert, delete, and modify rows in Chapter 3.
- **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.
- **Triggers:** 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.

- **Security:** SQL provides mechanisms to control users' access to data objects such as tables and views.
- **Transaction management:** Various commands allow a user to explicitly control aspects of how a transaction is to be executed.
- **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.

This chapter covers the query language features which are the core of SQL's DML, embedded and dynamic SQL, and triggers. We also briefly discuss some integrity constraint specifications that rely upon the use of the query language features of SQL. The ease of expressing queries in SQL has played a major role in the success of relational database systems. Although this material can be read independently of the preceding chapters, relational algebra and calculus (which we covered in Chapter 4) provide a formal foundation for a large subset of the SQL query language. Much of the power and elegance of SQL can be attributed to this foundation.

We will continue our presentation of SQL in Chapter 17, where we discuss aspects of SQL that are related to security. We discuss SQL's support for the *transaction* concept in Chapter 18.

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. We discuss *null* values, which are special values used to indicate unknown or nonexistent field values, in Section 5.6. We consider how SQL commands can be embedded in a host language in Section 5.7 and in Section 5.8, where we discuss how relations can be accessed one tuple at a time through the use of *cursors*. In Section 5.9 we describe how queries can be constructed at run-time using dynamic SQL, and in Section 5.10, we discuss two standard interfaces to a DBMS, called ODBC and JDBC. We discuss complex integrity constraints that can be specified using the SQL DDL in Section 5.11, 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 5.12 and 5.13. 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 monitors the database, detects these situations, and invokes the trigger. Several current relational DBMS products support some form of triggers, and the current draft of the SQL:1999 standard requires support for triggers.

Levels of SQL-92: SQL is a continously evolving standard with the current standard being SQL-92. When the standard is updated, DMBS vendors are usually not able to immediately conform to the new standard in their next product releases because they also have to address issues such as performance improvements and better system management. Therefore, three SQL-92 levels have been defined: *Entry SQL, Intermediate SQL*, and *Full SQL*. Of these, Entry SQL is closest to the previous standard, SQL-89, and therefore the easiest for a vendor to support. Intermediate SQL includes about half of the new features of SQL-92. Full SQL is the complete language.

The idea is to make it possible for vendors to achieve full compliance with the standard in steps and for customers to get an idea of how complete a vendor's support for SQL-92 really is, at each of these steps. In reality, while IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all support several features from Intermediate and Full SQL—and many of these products support features in the new SQL:1999 standard as well—they can claim full support only for Entry SQL.

5.1 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 will 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.

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.

---

[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 $S3$ 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

The basic form of an SQL query is as follows:

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

Such a query intuitively corresponds to a relational algebra expression involving selections, projections, and cross-products. 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. Let us consider a simple query.

(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 $\langle$*sname, age*$\rangle$. 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 $\langle s,a \rangle$ 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 $S3$ 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.

sname	*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.

(Q11) 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.

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 don't 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. Compute the cross-product of the tables in the **from-list**.
2. Delete those 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.

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 DBMS actually evaluates queries in Chapters 12 and 13; for now,

---

[2] Expressions with `NOT` can always be replaced by equivalent expressions without `NOT` given the set of comparison operators listed above.

our purpose is simply to explain the meaning of a query. We illustrate the conceptual evaluation strategy using the following query:

(Q1) Find the names of sailors who have reserved 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 $S4$ of Sailors shown in Figures 5.6 and 5.7, since the computation on our usual example instances ($R2$ and $S3$) 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*	*rating*	*age*
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Figure 5.7 Instance $S4$ of Sailors

The first step is to construct the cross-product $S4 \times R3$, which is shown in Figure 5.8.

sid	*sname*	*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	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

Figure 5.8 $S4 \times R3$

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 on R3 and S4

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 S, 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 sname
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 reserved a red boat.

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

This query contains a join of two tables, followed by a selection on the color of boats. We can think of B and R as rows in the corresponding tables that 'prove' that a sailor with sid R.sid reserved a red boat B.bid. On our example instances $R2$ and $S3$ (Figures

---

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

5.1 and 5.2), the answer consists of the *sid*s 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 red 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'
```

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 B.

(Q3) Find the colors of boats reserved by Lubber.

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

This query is very similar to the previous one. Notice 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 reserved 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 columns. 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. It can also contain *aggregates* such as *sum* and *count*, which we will discuss in Section 5.5. The SQL-92 standard also includes expressions over date

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.

and time values, which we will not discuss. Although not part of the SQL-92 standard, many implementations also support the use of built-in functions such as *sqrt*, *sin*, and *mod*.

(Q17) Compute increments for the ratings of persons 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 *expression1* = *expression2*.

```
SELECT S1.sname AS name1, S2.sname AS name2
FROM   Sailors S1, 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-92 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 that will match 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 '* could be 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 whose name begins and ends with B and has at least three characters.

```
SELECT S.age
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’)
```

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

---

[4] Note that although the SQL-92 standard includes these operations, many systems currently support only `UNION`. Also, many systems recognize the keyword `MINUS` for `EXCEPT`.

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 R1, Boats B1, Reserves R2, Boats B2
WHERE  S.sid = R1.sid AND R1.bid = B1.bid
       AND S.sid = R2.sid AND R2.bid = B2.bid
       AND B1.color='red' AND B2.color = 'green'
```

We can think of R1 and B1 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, R1, B1, 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.sid = 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 = R2.sid AND R2.bid = B2.bid 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.sname
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 $S3$, 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 *sid*s 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.

(Q19) Find the sid*s of all sailors 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. Thus, 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.bid = B.bid AND B.color = ‘red’
EXCEPT
SELECT R2.sid
FROM   Boats B2, Reserves R2
WHERE  R2.bid = B2.bid AND B2.color = ‘green’
```

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 have 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 *sid*s 58 and 71. The second part returns 22 and 31. The answer is, therefore, the set of *sid*s 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 $m + n$, where m and n are the numbers of times that the row appears in the two parts of the union. Similarly, one version of `INTERSECT` retains duplicates—the number of copies of a row in the result is $min(m, n)$—and one version of `EXCEPT` 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 **subquery**. When writing a query, we sometimes 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). This section discusses only subqueries that appear in the `WHERE` clause. The treatment of subqueries appearing elsewhere is quite similar. Examples of subqueries that appear in the `FROM` clause are discussed 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 *sid*s for sailors who have reserved boat 103 (the set contains 22, 31, and 74 on instances $R2$ and $S3$), 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. Notice 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 qualification 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 *bid*s of red boats (102 and 104 on instance $B1$). The subquery one level above finds the set of *sid*s of sailors who have reserved one of these boats. On instances $B1$, $R2$, and $S3$, this set of *sid*s contains 22, 31, and 64. The top-level query finds the names of sailors whose *sid* is in this set of *sid*s. For the example instances, we get Dustin, Lubber, and Horatio.

To find the names of sailors who have not reserved a red boat, we replace the outermost occurrence of `IN` by `NOT IN`:

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

---

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

```
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 query 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 `IN` with `NOT IN`. This modified query would compute the names of sailors who have reserved a boat that is not red, i.e., 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 *bid*s of red boats). The outer query then finds the *sid*s in Reserves tuples where the *bid* is not 102 or 104. On instances $B1$, $R2$, and $S3$, the outer query computes the set of *sid*s 22, 31, 64, and 74. Finally, we find the names of sailors whose *sid* is in this set.

We 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, i.e., who have only reserved red boats (if they've reserved any boats at all). Proceeding as in the previous paragraph, on instances $B1$, $R2$, and $S3$, the outer query computes the set of *sid*s (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 that we have seen thus far, the inner subquery has been completely independent of the outer query. In general the inner subquery could depend on the row that is 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. Thus, for each Sailor row S, we test whether the set of Reserves rows R such that *R.bid = 103* `AND` *S.sid = R.sid* is nonempty. If so, sailor S has reserved boat 103, and we retrieve the name. The subquery clearly depends on the current row S and must 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 *correlation*, 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 don't 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 will 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, it 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 $S3$, this computes the *sid*s 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 above query returns an empty answer set. To understand comparisons involving `ANY`, it is useful to think of the comparison being carried out repeatedly. In the example above, *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.rating* > `ANY` ... to return `true`.

(Q23) Find sailors whose rating is better than every sailor called Horatio.

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 *sid*s 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 in order for *S.rating* > `ALL` ... to return `true`.

As another illustration of `ALL`, consider the following query:

(Q24) 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, i.e., when it is the largest rating value. In the instance $S3$, the condition is only satisfied for *rating* 10, and the answer includes the *sid*s of sailors with this rating, i.e., 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 *sid*s that are included in the set of *sid*s 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 *sid*s 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 *sid*s to identify sailors (while intersecting) and have to return sailor names:

```
SELECT  S3.sname
FROM    Sailors S3
WHERE   S3.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 reserved 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 ))
```

Notice that this query is correlated—for each sailor S, we check to see that the set of boats reserved by S includes all boats. 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.

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-92 does not preclude this).

(Q25) Find the average age of all sailors.

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

On instance $S3$, the average age is 37.4. Of course, the `WHERE` clause can be used to restrict the sailors who are 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.) Thus, 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-92 standard but 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 we 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 $S3$, 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. Thus, without `DISTINCT` Q29 is equivalent to Q28. However, the use of `COUNT` (*) is better querying style when it is applicable.

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 $S3$, 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 prone—one could easily (and incorrectly!) use `ANY` instead of `ALL`, and retrieve sailors who are older than *some* sailor with 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 10, we could write 10 queries of the form:

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

where $i = 1, 2, \ldots, 10$. Writing 10 such queries is tedious. More importantly, 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 qualifications over groups (for example, we may only be interested 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*. The optional `AS` *new-name* term gives this column a name in the table that is the result of the query. Any of the aggregation operators can be used for **aggop**.

  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. If a column appears in list (1), but not in **grouping-list**, it is not clear what value should be assigned to it in an answer row.

- 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. Therefore, a column appearing in the **group-qualification** must appear as the argument to an aggregation operator, or it must also appear in **grouping-list**.

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

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

(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 $S3$ 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

Figure 5.10 Instance $S3$ of Sailors

The second step is to apply the qualification in the `WHERE` clause, $S.age >= 18$. This step eliminates the row $\langle 71, zorba, 10, 16\rangle$. 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. 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.

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

Figure 5.11 After Evaluation Step 3

rating	*age*
1	33.0
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

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 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.

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 sailorcount
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⟩.

It is interesting to observe that the following version of the above query is illegal:

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

Even though the group-qualification *B.color = 'red'* is single-valued per group, since the grouping attribute *bid* is a key for Boats (and therefore determines *color*), SQL disallows this query. 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 average age of sailors for each rating level that has at least two sailors.

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

After identifying groups based on *rating*, we retain only groups with at least two sailors. The answer to this query on instance $S3$ is shown in Figure 5.14.

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

Figure 5.14 Q34 Answer

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

Figure 5.15 Q35 Answer

rating	*avgage*
3	45.5
7	40.0
8	40.5

Figure 5.16 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 )
```

(Q35) Find the average age of sailors who are 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 2 sailors, it is discarded. For each remaining group, we output the average age. The answer to this query on instance $S3$ is shown in Figure 5.15. Notice 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 of sailors who are of voting age (i.e., at least 18 years old) for 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 )
```

The above formulation of the query reflects the fact that it is a variant of Q35. The answer to Q36 on instance $S3$ is shown in Figure 5.16. 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-92 standard.[6] 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 minimum 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!

---

[6] Not all systems currently support nested queries in the `FROM` clause.

A correct version of the above 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.

```
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 $S3$ is $\langle 10, 25.5 \rangle$.

As an exercise, the reader should consider whether the following query computes the same answer, and if not, why:

```
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 also include a *maiden-name* column. However, only married women who take their husband's last name have a maiden name. For single women 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 $\langle 98, Dan, null, 39 \rangle$ to represent Dan. The presence of *null* values complicates many 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 *rating* $= 8$ `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, it 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`, it 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 to `unknown` is eliminated. Eliminating rows that evaluate to `unknown` has a subtle but significant impact 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 hand, 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 earlier in Chapter 4.)

SQL-92 allows the desired type of join to be specified in the `FROM` clause. For example, the following query lists ⟨*sid,bid*⟩ pairs corresponding to sailors and boats they have reserved:

```
SELECT Sailors.sid, Reserves.bid
FROM   Sailors 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 5.17.

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

Figure 5.17 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 EMBEDDED SQL *

We have 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 commands. 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 may want to integrate a database application with a nice graphical user interface, or we may want to ask a query that cannot be expressed in SQL. (See Chapter 27 for examples of such queries.)

To deal with such situations, the SQL standard defines 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.

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). Of course, 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 from SQL commands. (SQL, like C and other programming languages, provides an operator to cast values of one type into values of another type.) The second complication has to do with the fact that SQL is **set-oriented**; commands operate on and produce tables, which are sets (or multisets) of rows. Programming languages do not typically have a data type that corresponds to sets or multisets of rows. Thus, although SQL commands deal with tables, the interface to the host language is constrained to be one row at a time. The *cursor* mechanism is introduced to deal with this problem; we discuss cursors in Section 5.8.

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

5.7.1 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 must 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_sname* 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`.

An important point to consider is that SQL needs some way 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[6]`, that is, a character string that is five characters long. (Recall the null-terminator in C strings!) In this chapter, we will assume that `SQLSTATE` is declared.

5.7.2 Embedding SQL Statements

All SQL statements that are 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` *stmt*]

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

5.8 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.

5.8.1 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:

- We 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 will see shortly.
- `INSERT`, `DELETE`, and `UPDATE` statements typically don't require a cursor, although some variants of `DELETE` and `UPDATE` do 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*. Thus, 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 command with `EXEC SQL`, how can the answers be bound to host language variables? The `INTO` clause is not adequate 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 will 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.

5.8.2 Properties of Cursors

The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR 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 updatable, 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 *sinfo* is an updatable cursor and is 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 *sinfo*; 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 *sinfo* cursor, we might modify *rating* values in Sailor rows by concurrently executing the command:

```
UPDATE  Sailors S
SET     S.rating = S.rating - 1
```

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 *sinfo*, 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 *sinfo* if it is executed after *sinfo* is opened. If `INSENSITIVE` is not specified, the behavior is implementation dependent in this situation.

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 5.18.

rating	*minage*
8	25.5
3	25.5
7	35.0

Figure 5.18 Order in which Tuples Are Fetched

5.9 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**. There are two main commands, `PREPARE` and `EXECUTE`, which we illustrate 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 a run-time overhead. Interactive and embedded SQL commands can be prepared once at compile time and then re-executed 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 can we pass parameters from the host langugage program to the SQL statement being prepared, for example?—but we will not discuss it further; readers interested in using dynamic SQL should consult one of the many good books devoted to SQL.

5.10 ODBC AND JDBC *

Embedded SQL enables the integration of SQL with a general-purpose programming language. As described in Section 5.7, a DBMS-specific preprocessor transforms the embedded SQL statements into function calls in the host language. The details of this translation vary across DBMS, and therefore even though the source code can be compiled 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 only one DBMS, but several different DBMSs 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. Since it is only known at run-time which DBMSs the application is going to access, drivers are loaded dynamically on demand. Existing drivers are registered with a **driver manager**, which manages the set of existing drivers.

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, we will refer in the remainder of this section 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 performs the following steps. A data source is selected, the corresponding driver is dynamically loaded, and a connection with the data source is established. 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 only visible to other connections after the connection has committed its changes. While a connection is open, 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.

5.10.1 Architecture

The architecture of ODBC/JDBC has four main components: the *application*, the *driver manager*, several data source specific *drivers*, and the corresponding *data sources*. Each component has different roles, as explained in the next paragraph.

The *application* initiates and terminates the connection with the data source. It sets transaction boundaries, submits SQL statements, and retrieves the results—all through a well-defined interface as specified by the ODBC/JDBC API. The primary goal of the *driver manager* is to load ODBC/JDBC drivers and to pass ODBC/JDBC function calls from the application to the correct driver. The driver manager also handles ODBC/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 ODBC/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. For example, drivers in JDBC are classified into four types depending on the architectural relationship between the application and the data source:

1. **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 an ODBC-JDBC bridge. In this case the application loads only one driver, namely the bridge.
2. **Type II (direct translation to the native API)** This driver translates JDBC function calls directly into method invocations of the API of one specific data source. The driver is dynamically linked, and is specific to the data source.
3. **Type III (network bridges)** The driver talks over a network to a middle-ware server that translates the JDBC requests into DBMS-specific method invocations. In this case, the driver on the client site (i.e., the network bridge) is not DBMS-specific.
4. **Type IV (direct translation over sockets)** 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 DBMS-specific.

5.10.2 An Example Using JDBC

JDBC is a collection of Java classes and interfaces that enables database access from programs written in the Java programming language. The classes and interfaces are part of the `java.sql` package. In this section, we illustrate the individual steps that are required to submit a database query to a data source and to retrieve the results.

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.

A session with a DBMS is started through creation of a `Connection` object. 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` method of the `Connection` class. The `Connection` class has methods to set the autocommit mode (`setAutoCommit`) and to retrieve the current autocommit mode (`getAutoCommit`). A transaction can be aborted using the `rollback` method.

The following Java example code dynamically loads a data source driver and establishes a connection:

```
Class.forName("oracle/jdbc.driver.OracleDriver");
Connection connection = DriverManager.getConnection(url,uid,password);
```

In considering the interaction of an application with a data source, the issues that 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. JDBC allows the creation of SQL statements that refer to variables in the Java host program. Similar to the `SQLSTATE` variable, JDBC throws an `SQLException` if an error occurs. The information includes `SQLState`, a string describing the error. As in embedded SQL, JDBC provides the concept of a cursor through the `ResultSet` class.

While a complete discussion of the actual implementation of these concepts is beyond the scope of the discussion here, we complete this section by considering two illustrative JDBC code fragments.

In our first example, we show how JDBC refers to Java variables inside an SQL statement. During a session, all interactions with a data source are encapsulated into objects that are created by the `Connection` object. SQL statements that refer to variables in the host program are objects of the class `PreparedStatement`. Whereas in embedded SQL the actual names of the host language variables appear in the SQL query text, JDBC replaces each parameter with a "?" and then sets values of each parameter at run-time through `settype` methods, where `type` is the type of the parameter. These points are illustrated in the following Java program fragment, which inserts one row into the Sailors relation:

```
connection.setAutoCommit(false);
PreparedStatement pstmt =
      connection.prepareStatement("INSERT INTO Sailors VALUES ?,?,?,?");
pstmt.setString(1, j_name); pstmt.setInt(2, j_id);
pstmt.setInt(3, j_rating); pstmt.setInt(4, j_age);
pstmt.execute();
pstmt.close();
connection.commit();
```

Our second example shows how the `ResultSet` class provides the functionality of a cursor. After the SQL statement *stmt* is executed, *result* is positioned right before the first row. The method `next` fetches the next row and enables reading of its values through `gettype` methods, where `type` is the type of the field.

```
Statement stmt = connection.createStatement();
ResultSet res = stmt.executeQuery("SELECT S.name, S.age FROM Sailors S");
while (result.next()) {
      String name = res.getString(1);
      int age = res.getInt(2);
      // process result row
}
stmt.close();
```

5.11 COMPLEX INTEGRITY CONSTRAINTS IN SQL-92 *

In this section we discuss the specification of complex integrity constraints in SQL-92, utilizing the full power of SQL query constructs. The features discussed in this section complement the integrity constraint features of SQL presented in Chapter 3.

5.11.1 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.

5.11.2 Domain Constraints

A user can define a new domain using the `CREATE DOMAIN` statement, which makes use of `CHECK` constraints.

```
CREATE DOMAIN ratingval INTEGER DEFAULT 0
                        CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

`INTEGER` is the **base 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 0 associated with `ratingval` is used. (If a default value is specified for the column as part of the table definition, this takes precedence over the default value associated with the domain.) This feature can be used to minimize data entry errors; common default values are automatically filled in rather than being typed in.

SQL-92's support for the concept of a domain is limited in an important respect. For example, we can define two domains called `Sailorid` and `Boatclass`, each using

`INTEGER` as a base type. The intent is to force a comparison of a `Sailorid` value with a `Boatclass` 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-92. This problem is addressed through the introduction of *distinct types* in SQL:1999 (see Section 3.4).

5.11.3 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 associated 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 any one 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 might 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 very 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 )
```

5.12 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 new 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 *after*: a trigger that initializes a variable used to count the number 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).

5.12.1 Examples of Triggers in SQL

The examples shown in Figure 5.19, written using Oracle 7 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 $age < 18$.

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

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

Figure 5.19 Examples Illustrating Triggers

One of the example triggers in Figure 5.19 executes before the activating statement, and the other example executes after. 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 5.19 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 triggering 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 *init_count* 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 5.19, 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. The SQL:1999 draft 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 5.20 and is an alternative to the triggers shown in Figure 5.19.

The definition in Figure 5.20 uses the syntax of the SQL:1999 draft, 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 modifications 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 5.19 only computes the count; an additional trigger is required to insert the appropriate tuple into the statistics table.)

```
CREATE TRIGGER set_count 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 I.age < 18
```

Figure 5.20 Set-Oriented Trigger

5.13 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.

5.13.1 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 some 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 same 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.

5.13.2 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) will achieve 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 (e.g., an insert or delete statement). 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 *itemid*, *quantity*, *customerid*, 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 the above 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.

5.13.3 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 can tell the customer, and possibly generate additional sales! We 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 5.12 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 21.10.1). Our list of potential uses of triggers is not exhaustive; for example, triggers have also been considered for workflow management and enforcing business rules.

5.14 POINTS TO REVIEW

- A basic SQL query has a `SELECT`, a `FROM`, and a `WHERE` clause. The query answer is a *multiset* of tuples. Duplicates in the query result can be removed by using `DISTINCT` in the `SELECT` clause. Relation names in the `WHERE` clause can be followed by a *range variable*. The output can involve arithmetic or string expressions over column names and constants and the output columns can be renamed using `AS`. SQL provides string pattern matching capabilities through the `LIKE` operator. **(Section 5.2)**

- SQL provides the following (multi)set operations: `UNION`, `INTERSECT`, and `EXCEPT`. **(Section 5.3)**
- Queries that have (sub-)queries are called *nested queries*. Nested queries allow us to express conditions that refer to tuples that are results of a query themselves. Nested queries are often *correlated*, i.e., the subquery contains variables that are bound to values in the outer (main) query. In the `WHERE` clause of an SQL query, complex expressions using nested queries can be formed using `IN`, `EXISTS`, `UNIQUE`, `ANY`, and `ALL`. Using nested queries, we can express division in SQL. **(Section 5.4)**
- SQL supports the aggregate operators `COUNT`, `SUM`, `AVERAGE`, `MAX`, and `MIN`. **(Section 5.5)**
- Grouping in SQL extends the basic query form by the `GROUP BY` and `HAVING` clauses. **(Section 5.5.1)**
- A special column value named *null* denotes unknown values. The treatment of *null* values is based upon a three-valued logic involving `true`, `false`, and `unknown`. **(Section 5.6)**
- SQL commands can be executed from within a host language such as C. Conceptually, the main issue is that of data type mismatches between SQL and the host language. **(Section 5.7)**
- Typical programming languages do not have a data type that corresponds to a collection of records (i.e., tables). Embedded SQL provides the *cursor* mechanism to address this problem by allowing us to retrieve rows one at a time. **(Section 5.8)**
- Dynamic SQL enables interaction with a DBMS from a host language without having the SQL commands fixed at compile time in the source code. **(Section 5.9)**
- ODBC and JDBC are application programming interfaces that introduce a layer of indirection between the application and the DBMS. This layer enables abstraction from the DBMS at the level of the executable. **(Section 5.10)**
- The query capabilities of SQL can be used to specify a rich class of integrity constraints, including domain constraints, `CHECK` constraints, and assertions. **(Section 5.11)**
- A *trigger* is a procedure that is automatically invoked by the DBMS in response to specified changes to the database. A trigger has three parts. The *event* describes the change that activates the trigger. The *condition* is a query that is run whenever the trigger is activated. The *action* is the procedure that is executed if the trigger is activated and the condition is true. A *row-level trigger* is activated for each modified record, a *statement-level trigger* is activated only once per `INSERT` command. **(Section 5.12)**

- What triggers are activated in what order can be hard to understand because a statement can activate more than one trigger and the action of one trigger can activate other triggers. Triggers are more flexible than integrity constraints and the potential uses of triggers go beyond maintaining database integrity. **(Section 5.13)**

EXERCISES

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`, *fname:* `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 answers.

1. Find the names of all Juniors (Level = JR) who are enrolled in a class taught by I. Teach.
2. Find the age of the oldest student who is either a History major or is 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 names of all students who are enrolled in two classes that meet at the same time.
5. Find the names of faculty members who teach in every room in which some class is taught.
6. Find the names of faculty members for whom 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. Find the names of students who are enrolled in the maximum number of classes.
10. Find the names of students who are not enrolled in any class.
11. 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 *pname*s of parts for which there is some supplier.
2. Find the *sname*s of suppliers who supply every part.
3. Find the *sname*s of suppliers who supply every red part.
4. Find the *pname*s of parts supplied by Acme Widget Suppliers and by no one else.
5. Find the *sid*s 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 *sid*s of suppliers who supply only red parts.
8. Find the *sid*s of suppliers who supply a red part and a green part.
9. Find the *sid*s of suppliers who supply a red part or a green part.

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

Flights(*flno:* `integer`, *from:* `string`, *to:* `string`, *distance:* `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 that he (or she) is certified for.
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 1,000 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 *aid*s of all aircraft that can be used on routes from Los Angeles to Chicago.
7. Identify the flights that can be piloted by every pilot who makes more than $100,000. (*Hint:* The pilot must be certified for at least one plane with a sufficiently large cruising range.)
8. Print the *ename*s of pilots who can operate planes with *cruisingrange* greater than 3,000 miles, but are not certified on any Boeing aircraft.

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 5.21 An Instance of Sailors

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.

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`, *pct_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.
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 *managerid*s of managers who manage only departments with budgets greater than $1,000,000.
5. Find the *ename*s of managers who manage the departments with the largest budget.
6. If a manager manages more than one department, he or she *controls* the sum of all the budgets for those departments. Find the *managerid*s of managers who control more than $5,000,000.
7. Find the *managerid*s of managers who control the largest amount.

Exercise 5.5 Consider the instance of the Sailors relation shown in Figure 5.21.

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 computed above by the count, would the result be the same as the average? How would your answer change if the above 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 5.21. 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.
   (a) 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 *sid=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 S1 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 language 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 SQL relation Edges(*from, to*), using embedded SQL commands. (You don't have to write the program; just explain the main points to be dealt with.)
10. Explain the following terms with respect to cursors: *updatability, sensitivity*, 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 SQL, 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`)
Works(*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`, *major:* `string`,
level: `string`, *age:* `integer`)
Class(*name:* `string`, *meets_at:* `time`, *room:* `string`, *fid:* `integer`)
Enrolled(*snum:* `integer`, *cname:* `string`)
Faculty(*fid*: `integer`, *fname:* `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 the above 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 class 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 Math101.

(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.

(m) The total enrollment in courses taught by faculty in the department with *deptid=33* is greater than the number of Math majors.

(n) There must be at least one CS major if there are any students whatsoever.

(o) 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`, *managerid:* `integer`)

Write SQL-92 integrity constraints (domain, key, foreign key, or `CHECK` constraints; or assertions) or SQL:1999 triggers to ensure each of the following requirements, considered independently.

1. Employees must make a minimum salary of $1,000.
2. Every manager must be also be an employee.
3. The total percentage of all 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 all employees in the department.

PROJECT-BASED EXERCISES

Exercise 5.11 Identify the subset of SQL-92 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 [90, 130]. 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 [455]; we refer readers to this book and to [170] for a more detailed treatment. Date offers an insightful critique of SQL in [167]. Although some of the problems have been addressed in SQL-92, others remain. A formal semantics for a large subset of SQL queries is presented in [489]. SQL-92 is the current International Standards Organization (ISO) and American National Standards Institute (ANSI) standard. Melton is the editor of the ANSI document on the SQL-92 standard, document X3.135-1992. The corresponding ISO document is ISO/IEC 9075:1992. A successor, called SQL:1999, builds on SQL-92 and includes procedural language extensions, user-defined types, row ids, a call-level interface, multimedia data types, recursive queries, and other enhancements; SQL:1999 is close to ratification (as of June 1999). Drafts of the SQL:1999 (previously called SQL3) deliberations are available at the following URL:

`ftp://jerry.ece.umassd.edu/isowg3/`

The SQL:1999 standard is discussed in [200].

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 JavaSoft web page (`java.sun.com/products/jdbc`). There exist many books on ODBC, for example, Sander's ODBC Developer's Guide [567] and the Microsoft ODBC SDK [463]. Books on JDBC include works by Hamilton et al. [304], Reese [541], and White et al. [678].

[679] contains a collection of papers that cover the active database field. [695] includes a good in-depth introduction to active rules, covering semantics, applications and design issues. [213] discusses SQL extensions for specifying integrity constraint checks through triggers. [104] also discusses a procedural mechanism, called an *alerter*, for monitoring a database. [154] is a recent paper that suggests how triggers might be incorporated into SQL extensions. Influential active database prototypes include Ariel [309], HiPAC [448], ODE [14], Postgres [632], RDL [601], and Sentinel [29]. [126] compares various architectures for active database systems.

[28] 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 [244] and [693]. Designing and managing complex rule systems is discussed in [50, 190]. [121] discusses rule management using Chimera, a data model and language for active database systems.