CHAPTER

7

Advanced SQL

Visit www.pearsonhighered.com/hoffer to view the accompanying video for this chapter.

Learning Objectives

After studying this chapter, you should be able to:

- Concisely define each of the following key terms: **join, equi-join, natural join, outer join, correlated subquery, user-defined data type, Persistent Stored Modules (SQL/PSM), trigger, function, procedure, embedded SQL,** and **dynamic SQL.**
- Write single- and multiple-table queries using SQL commands.
- Define three types of join commands and use SQL to write these commands.
- Write noncorrelated and correlated subqueries and know when to write each.
- Establish referential integrity using SQL.
- Understand common uses of database triggers and stored procedures.
- Discuss the SQL:200n standard and explain its enhancements and extensions.

INTRODUCTION

The previous chapter introduced SQL and explored its capabilities for querying one table. The real power of the relational model derives from its storage of data in many related entities. Taking advantage of this approach to data storage requires establishing relationships and constructing queries that use data from multiple tables. This chapter examines multiple-table queries in some detail. Different approaches to getting results from more than one table are demonstrated, including the use of subqueries, inner and outer joins, and union joins.

Once an understanding of basic SQL syntax is gained, it is important to understand how SQL is used in the creation of applications. Triggers, small modules of code that include SQL, execute automatically when a particular condition, defined in the trigger, exists. Procedures are similar modules of code but must be called before they execute. SQL commands are often embedded within modules written in a host language, such as C, PHP, .NET, or Java. Dynamic SQL creates SQL statements on the fly, inserting parameter values as needed, and is essential to Web applications. Brief introductions and examples of each of these methods are included in this chapter. Some of the enhancements and extensions to SQL included in SQL:200n are also covered. Oracle, a leading RDBMS vendor, is SQL:1999 compliant.

Completion of this chapter gives the student an overview of SQL and some of the ways in which it may be used. Many additional features, often referred to as "obscure" in more detailed SQL texts, will be needed in particular situations. Practice with the syntax included in this chapter will give you a good start toward mastery of SQL.

PROCESSING MULTIPLE TABLES

Now that we have explored some of the possibilities for working with a single table, it's time to bring out the light sabers, jet packs, and tools for heavy lifting: We will work with multiple tables simultaneously. The power of RDBMSs is realized when working with multiple tables. When relationships exist among tables, the tables can be linked together in queries. Remember from Chapter 4 that these relationships are established by including a common column(s) in each table where a relationship is needed. Often this is accomplished by setting up a primary key–foreign key relationship, where the foreign key in one table references the primary key in another, and the values in both come from a common domain. We can use these columns to establish a link between two tables by finding common values in the columns. Figure 7-1 carries forward two relations from Figure 6-3, depicting part of the Pine Valley Furniture Company database. Notice that CustomerID values in Order_T correspond to CustomerID values in Customer_T. Using this correspondence, we can deduce that Contemporary Casuals placed orders 1001 and 1010 because Contemporary Casuals's CustomerID is 1, and Order_T shows that OrderID 1001 and 1010 were placed by customer 1. In a relational system, data from related tables are combined into one result table or view and then displayed or used as input to a form or report definition.

The linking of related tables varies among different types of relational systems. In SQL, the WHERE clause of the SELECT command is also used for multiple-table operations. In fact, SELECT can include references to two, three, or more tables in the same command. As illustrated next, SQL has two ways to use SELECT for combining data from related tables.

Join
A relational operation that causes two tables with a common domain to be combined into a single table or view.

The most frequently used relational operation, which brings together data from two or more related tables into one resultant table, is called a **join**. Originally, SQL specified a join implicitly by referring in a WHERE clause to the matching of common columns over which tables were joined. Since SQL-92, joins may also be specified in the FROM clause. In either case, two tables may be joined when each contains a column that shares a common domain with the other. As mentioned previously, a primary key from one table and a foreign key that references the table with the primary key will share a common domain and are frequently used to establish a join. Occasionally, joins will be established using columns that share a common domain but not the primary-foreign key relationship, and that also works (e.g., we might join customers and salespersons based on common postal codes, for which there is no relationship in the data model for the database). The result of a join operation is a single table. Selected columns from all the tables are included. Each row returned contains data from rows in the different input tables where values for the common columns match.

Explicit JOIN . . . ON commands are included in the FROM clause. The following join operations are included in the standard, though each RDBMS product is likely to support only a subset of the keywords: INNER, OUTER, FULL, LEFT, RIGHT, CROSS, and UNION. (We'll explain these in a following section.) NATURAL is an optional keyword. No matter what form of join you are using, *there should be one ON or WHERE specification for each pair of tables being joined.* Thus, if two tables are to be combined, one ON or WHERE condition would be necessary, but if three tables (A, B, and C) are to be combined, then two

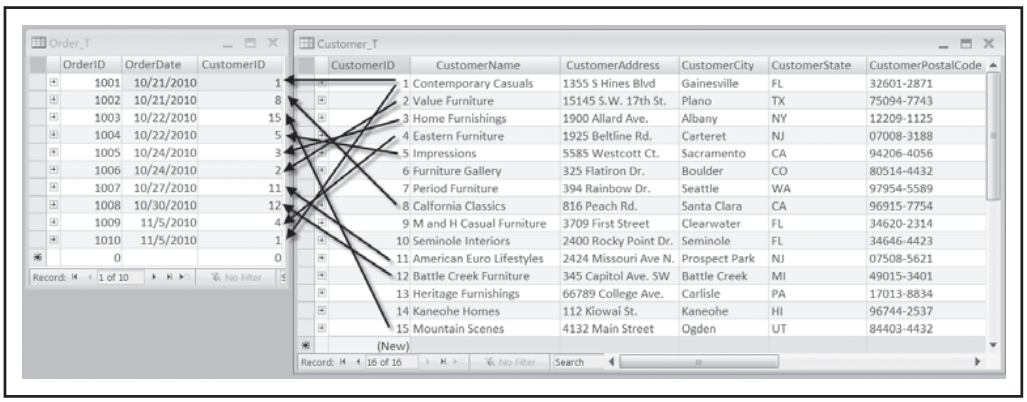


FIGURE 7-1 Pine Valley Furniture Company Customer_T and Order_T tables, with pointers from customers to their orders

ON or WHERE conditions would be necessary because there are 2 pairs of tables (A-B and B-C), and so forth. Most systems support up to 10 pairs of tables within one SQL command. At this time, core SQL does not support CROSS JOIN, UNION JOIN, FULL [OUTER] JOIN, or the keyword NATURAL. Knowing this should help you understand why you may not find these implemented in the RDBMS you are using. Because they are included in the SQL:200n standard and are useful, expect to find them becoming more widely available.

The various types of joins are described in the following sections.

Equi-join

Equi-join
A join in which the joining condition is based on equality between values in the common columns. Common columns appear (redundantly) in the result table.

With an **equi-join**, the joining condition is based on *equality* between values in the common columns. For example, if we want to know data about customers who have placed orders, that information is kept in two tables, Customer_T and Order_T. It is necessary to match customers with their orders and then collect the information about, for example, customer name and order number in one table in order to answer our question. We call the table created by the query the result or *answer table*.

Query: What are the customer IDs and names of all customers, along with the order IDs for all the orders they have placed?

```
SELECT Customer_T.CustomerID, Order_T.CustomerID,
   CustomerName, OrderID
      FROM Customer_T, Order_T
         WHERE Customer_T.CustomerID = Order_T. CustomerID
         ORDER BY OrderID
```

Result:

CUSTOMERID	CUSTOMERID	CUSTOMERNAME	ORDERID
1	1	Contemporary Casuals	1001
8	8	California Classics	1002
15	15	Mountain Scenes	1003
5	5	Impressions	1004
3	3	Home Furnishings	1005
2	2	Value Furniture	1006
11	11	American Euro Lifestyles	1007
12	12	Battle Creek Furniture	1008
4	4	Eastern Furniture	1009
1	1	Contemporary Casuals	1010

10 rows selected.

The redundant CustomerID columns, one from each table, demonstrate that the customer IDs have been matched and that matching gives one row for each order placed. We prefixed the CustomerID columns with the names of their respective tables so SQL knows which CustomerID column we referenced in each element of the SELECT list; we did not have to prefix CustomerName nor OrderID with their associated table names because each of these columns is found in only one table in the FROM list.

The importance of achieving the match between tables can be seen if the WHERE clause is omitted. That query will return all combinations of customers and orders, or 150 rows, and includes all possible combinations of the rows from the two tables (i.e., an order will be matched with every customer, not just the customer who placed that order). In this case, this join does not reflect the relationships that exist between the

tables and is not a useful or meaningful result. The number of rows is equal to the number of rows in each table, multiplied together (10 orders × 15 customers = 150 rows). This is called a *Cartesian join*. Cartesian joins with spurious results will occur when any joining component of a WHERE clause with multiple conditions is missing or erroneous. In the rare case that a Cartesian join is desired, omit the pairings in the WHERE clause. A Cartesian join may be explicitly created by using the phrase CROSS JOIN in the FROM statement. FROM Customer_T CROSS JOIN Order_T would create a Cartesian product of all customers with all orders. (Use this query only if you really mean to because a cross join against a production database can produce hundreds of thousands of rows and can consume significant computer time—plenty of time to receive a pizza delivery!)

The keywords INNER JOIN . . . ON are used to establish an equi-join in the FROM clause. While the syntax demonstrated here is Microsoft Access SQL syntax, note that some systems, such as Oracle and Microsoft SQL Server, treat the keyword JOIN by itself without the word INNER to establish an equi-join:

Query: What are the customer IDs and names of all customers, along with the order IDs for all the orders they have placed?

```
SELECT Customer_T.CustomerID, Order_T.CustomerID,
  CustomerName, OrderID
FROM Customer_T INNER JOIN Order_T ON
  Customer_T.CustomerID = Order_T.CustomerID
ORDER BY OrderID;
```

Result: Same as the previous query.

Simplest of all would be to use the JOIN . . . USING syntax, if this is supported by the RDBMS you are using. If the database designer thought ahead and used identical column names for the primary and foreign keys, as has been done with CustomerID in the Customer_T and Order_T tables, the following query could be used:

```
SELECT Customer_T.CustomerID, Order_T.CustomerID,
  CustomerName, OrderID
FROM Customer_T INNER JOIN Order_T USING CustomerID
ORDER BY OrderID ;
```

Notice that the WHERE clause now functions only in its traditional role as a filter as needed. Since the FROM clause is generally evaluated prior to the WHERE clause, some users prefer using the newer syntax of ON or USING in the FROM clause. A smaller record set that meets the join conditions is all that must be evaluated by the remaining clauses, and performance may improve. All DBMS products support the traditional method of defining joins within the WHERE clause. Microsoft SQL Server supports the INNER JOIN . . . ON syntax, Oracle has supported it since 9i, and MySQL has supported it since 3.23.17.

We again emphasize that SQL is a set-oriented language. Thus, this join example is produced by taking the customer table and the order table as two sets and appending together those rows from Customer_T with rows from Order_T that have equal CustomerID values. This is a set intersection operation, which is followed by appending the selected columns from the matching rows. Figure 7-2 uses set diagrams to display the most common types of two-table joins.

Natural Join

Natural join
A join that is the same as an equi-join except that one of the duplicate columns is eliminated in the result table.

A **natural join** is the same as an equi-join, except that it is performed over matching columns, and one of the duplicate columns is eliminated in the result table. The natural join is the most commonly used form of join operation. (No, a "natural" join is not a more healthy join with more fiber, and there is no un-natural join; but you will find it a natural and essential function with relational databases.) Notice in the command below that CustomerID must still be qualified because there is still ambiguity; CustomerID

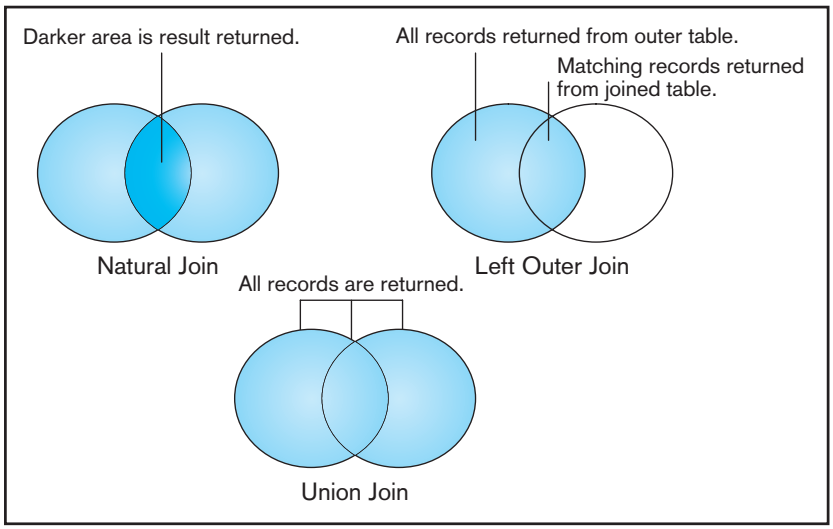


FIGURE 7-2 **Visualization of different join types, with the results returned in the shaded area**

exists in both Customer_T and Order_T, and therefore it must be specified from which table CustomerID should be displayed. NATURAL is an optional keyword when the join is defined in the FROM clause.

> ***Query:*** For each customer who has placed an order, what is the customer's ID, name, and order number?

```
SELECT Customer_T.CustomerID, CustomerName, OrderID
FROM Customer_T NATURAL JOIN Order_T ON
Customer_T.CustomerID = Order_T.CustomerID;
```

Note that the order of table names in the FROM clause is immaterial. The query optimizer of the DBMS will decide in which sequence to process each table. Whether indexes exist on common columns will influence the sequence in which tables are processed, as will which table is on the 1 and which is on the M side of 1:*M* relationship. If a query takes significantly different amounts of time, depending on the order in which tables are listed in the FROM clause, the DBMS does not have a very good query optimizer.

Outer Join

In joining two tables, we often find that a row in one table does not have a matching row in the other table. For example, several CustomerID numbers do not appear in the Order_T table. In Figure 7-1 pointers have been drawn from customers to their orders. Contemporary Casuals has placed two orders. Furniture Gallery, Period Furniture, M & H Casual Furniture, Seminole Interiors, Heritage Furnishings, and Kaneohe Homes have not placed orders in this small example. We can assume that this is because those customers have not placed orders since 10/21/2010, or their orders are not included in our very short sample Order_T table. As a result, the equi-join and natural join shown previously do not include all the customers shown in Customer_T.

Of course, the organization may be very interested in identifying those customers who have not placed orders. It might want to contact them to encourage new orders, or it might be interested in analyzing these customers to discern why they are not ordering. Using an **outer join** produces this information: Rows that do not have matching values in common columns are also included in the result table. Null values appear in columns where there is not a match between tables.

Outer join
A join in which rows that do not have matching values in common columns are nevertheless included in the result table.

Outer joins can be handled by the major RDBMS vendors, but the syntax used to accomplish an outer join varies across vendors. The example given here uses ANSI standard syntax. When an outer join is not available explicitly, use UNION and NOT EXISTS (discussed later in this chapter) to carry out an outer join. Here is an outer join.

Query: List customer name, identification number, and order number for all customers listed in the Customer table. Include the customer identification number and name even if there is no order available for that customer.

```
SELECT Customer_T.CustomerID, CustomerName, OrderID
  FROM Customer_T LEFT OUTER JOIN Order_T
  WHERE Customer_T.CustomerID = Order_T. CustomerID;
```

The syntax LEFT OUTER JOIN was selected because the Customer_T table was named first, and it is the table from which we want all rows returned, regardless of whether there is a matching order in the Order_T table. Had we reversed the order in which the tables were listed, the same results would be obtained by requesting a RIGHT OUTER JOIN. It is also possible to request a FULL OUTER JOIN. In that case, all rows from both tables would be returned and matched, if possible, including any rows that do not have a match in the other table. INNER JOINs are much more common than OUTER JOINs because outer joins are necessary only when the user needs to see data from all rows, even those that have no matching row in another table.

It should also be noted that the OUTER JOIN syntax does not apply easily to a join condition of more than two tables. The results returned will vary according to the vendor, so be sure to test any outer join syntax that involves more than two tables until you understand how it will be interpreted by the DBMS being used.

Also, the result table from an outer join may indicate NULL (or a symbol, such as ??) as the values for columns in the second table where no match was achieved. If those columns could have NULL as a data value, you cannot know whether the row returned is a matched row or an unmatched row unless you run another query that checks for null values in the base table or view. Also, a column that is defined as NOT NULL may be assigned a NULL value in the result table of an OUTER JOIN. In the following result, NULL values are shown by an empty value (i.e., a customer without any orders is listed with no value for OrderID).

Result:

CUSTOMERID	CUSTOMERNAME	ORDERID
1	Contemporary Casuals	1001
1	Contemporary Casuals	1010
2	Value Furniture	1006
3	Home Furnishings	1005
4	Eastern Furniture	1009
5	Impressions	1004
6	Furniture Gallery	
7	Period Furniture	
8	California Classics	1002
9	M & H Casual Furniture	
10	Seminole Interiors	
11	American Euro Lifestyles	1007
12	Battle Creek Furniture	1008
13	Heritage Furnishings	
14	Kaneohe Homes	
15	Mountain Scenes	1003

16 rows selected.

It may help you to glance back at Figures 7-1 and 7-2. In Figure 7-2, customers are represented by the left circle and orders are represented by the right. With an INNER JOIN of Customer_T and Order_T, only the 10 rows that have arrows drawn in Figure 7-1 will be returned. The LEFT OUTER JOIN on Customer_T, returns all of the customers along with the orders they have placed, and customers are returned even if they have not placed orders. Because Customer 1, Contemporary Casuals, has placed two orders, a total of 16 rows are returned because rows are returned for both orders placed by Contemporary Casuals.

The advantage of an outer join is that information is not lost. Here, all customer names were returned, whether or not they had placed orders. Requesting a RIGHT OUTER join would return all orders. (Because referential integrity requires that every order be associated with a valid customer ID, this right outer join would only ensure that referential integrity is being enforced.) Customers who had not placed orders would not be included in the result.

> ***Query:*** List customer name, identification number, and order number for all orders listed in the Order table. Include the order number, even if there is no customer name and identification number available.

```
SELECT Customer_T.CustomerID,CustomerName, OrderID
  FROM Customer_T RIGHT OUTER JOIN Order_T ON
    Customer_T.CustomerID = Order_T.CustomerID;
```

Union Join

SQL:1999 and, by extension, SQL:200n also allow for the use of UNION JOIN, which has not yet been implemented in all DBMS products. The results of a UNION JOIN will be a table that includes all data from each table that is joined. The result table will contain all columns from each table and will contain an instance for each row of data included from each table. Thus, a UNION JOIN of the Customer_T table (15 customers and 6 attributes) and the Order_T table (10 orders and 3 attributes) will return a result table of 25 rows (15 + 10) and 9 columns (6 + 3). Assuming that each original table contained no nulls, each customer row in the result table will contain three attributes with assigned null values, and each order row will contain six attributes with assigned null values.

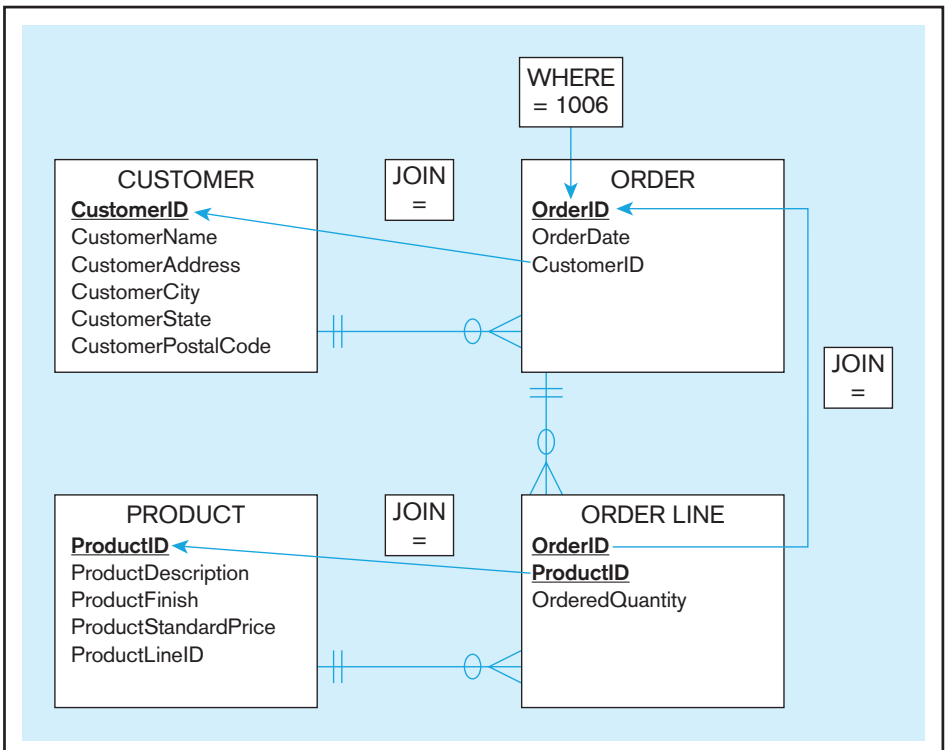
UNION JOINs may not include the keyword NATURAL, an ON clause, or a USING clause. Each of these implies an equivalence that would conflict with the UNION JOIN's inclusion of all the data from each table that is joined. Do not confuse this command with the UNION command that joins multiple SELECT statements and is covered later in this chapter.

Sample Join Involving Four Tables

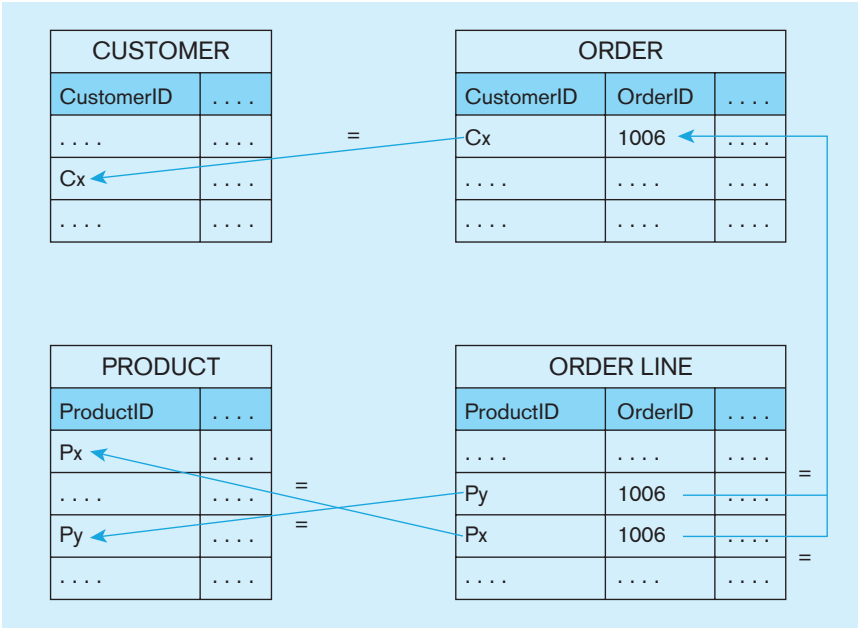
Much of the power of the relational model comes from its ability to work with the relationships among the objects in the database. Designing a database so that data about each object are kept in separate tables simplifies maintenance and data integrity. The capability to relate the objects to each other by joining the tables provides critical business information and reports to employees. Although the examples provided in Chapters 6 and 7 are simple and constructed only to provide a basic understanding of SQL, it is important to realize that these commands can be and often are built into much more complex queries that provide exactly the information needed for a report or process.

Here is a sample join query that involves a four-table join. This query produces a result table that includes the information needed to create an invoice for order number 1006. We want the customer information, the order and order line information, and the product information, so we will need to join four tables. Figure 7-3a shows an annotated ERD of the four tables involved in constructing this query; Figure 7-3b shows an abstract instance diagram of the four tables with order 1006 hypothetically having two line items for products Px and Py, respectively. We encourage you to draw such diagrams to help conceive the data involved in a query and how you might then construct the corresponding SQL command with joins.

FIGURE 7-3 Diagrams depicting a four-table join
(a) Annotated ERD with relations used in a four-table join



(b) Annotated instance diagram of relations used in a four-table join



Query: Assemble all information necessary to create an invoice for order number 1006.

```
SELECT Customer_T.CustomerID, CustomerName, CustomerAddress,
   CustomerCity, CustomerState, CustomerPostalCode, Order_T.OrderID,
   OrderDate, OrderedQuantity, ProductDescription, StandardPrice,
   (OrderedQuantity * ProductStandardPrice)
FROM Customer_T, Order_T, OrderLine_T, Product_T
   WHERE Order_T.CustomerID = Customer_T.CustomerID
      AND Order_T.OrderID = OrderLine_T.OrderID
      AND OrderLine_T.ProductID = Product_T.ProductID
      AND Order_T.OrderID = 1006;
```

FIGURE 7-4 Results from a four-table join (edited for readability)

CUSTOMERID	CUSTOMERNAME	CUSTOMERADDRESS	CUSTOMER CITY	CUSTOMER STATE	CUSTOMER POSTALCODE
2	Value Furniture	15145 S. W. 17th St.	Plano	TX	75094 7743
2	Value Furniture	15145 S. W. 17th St.	Plano	TX	75094 7743
2	Value Furniture	15145 S. W. 17th St.	Plano	TX	75094 7743

ORDERID	ORDERDATE	ORDERED QUANTITY	PRODUCTNAME	PRODUCT STANDARDPRICE	(QUANTITY* STANDARDPRICE)
1006	24-OCT -10	1	Entertainment Center	650	650
1006	24-OCT -10	2	Writer's Desk	325	650
1006	24-OCT -10	2	Dining Table	800	1600

The results of the query are shown in Figure 7-4. Remember, because the join involves four tables, there are three column join conditions, as follows:

1. Order_T.CustomerID = Customer_T.CustomerID = links an order with its associated customer.
2. Order_T.OrderID = OrderLine_T.OrderID links each order with the details of the items ordered.
3. OrderLine_T.ProductID = Product_T.ProductID links each order detail record with the product description for that order line.

Self-Join

There are times when a join requires matching rows in a table with other rows in that same table—that is, joining a table with itself. There is no special command in SQL to do this, but people generally call this operation a *self-join*. Self-joins arise for several reasons, the most common of which is a unary relationship, such as the Supervises relationship in the Pine Valley Furniture database in Figure 2-22. This relationship is implemented by placing in the EmployeeSupervisor column the EmployeeID (foreign key) of the employee's supervisor, another employee. With this recursive foreign key column, we can ask the following question:

Query: What are the employee ID and name of each employee and the name of his or her supervisor (label the supervisor's name Manager)?

```
SELECT E.EmployeeID, E.EmployeeName, M.EmployeeName AS Manager
  FROM Employee_T E, Employee_T M
  WHERE E.EmployeeSupervisor = M.EmployeeID;
```

Result:

EMPLOYEEID	EMPLOYEENAME	MANAGER
123-44-347	Jim Jason	Robert Lewis

There are two things to note in this query. First, the Employee table is, in a sense, serving two roles: It contains a list of employees and a list of managers. Thus, the FROM clause refers to the Employee_T table twice, once for each of these roles. However, to distinguish these roles in the rest of the query, we give the Employee_T table an alias for each role (in this case, E for employee and M for manager roles, respectively). Then the columns from the SELECT list are clear: first the ID and name of an employee (with prefix E) and then the name of a manager (with prefix M). Which manager? That then is the second point: The WHERE clause joins the "employee" and "manager" tables based on the foreign key from employee (EmployeeSupervisor) to manager (EmployeeID). As far

as SQL is concerned, it considers the E and M tables to be two different tables that have identical column names, so the column names must have a suffix to clarify from which table a column is to be chosen each time it is referenced.

It turns out that there are various interesting queries that can be written using self-joins following unary relationships. For example, which employees have a salary greater than the salary of their manager (not uncommon in professional baseball, but generally frowned on in business or government organizations), or (if we had this data in our database) is anyone married to his or her manager (not uncommon in a family-run business but possibly prohibited in many organizations)? Several of the Problems and Exercises at the end of this chapter require queries with a self-join.

As with any other join, it is not necessary that a self-join be based on a foreign key and a specified unary relationship. For example, when a salesperson is scheduled to visit a particular customer, maybe she would want to know who are all the other customers in the same postal code as the customer she is scheduled to visit. Remember, it is possible to join rows on columns from different (or the same) tables as long as those columns come from the same domain of values and the linkage of values from those columns makes sense. For example, even though ProductFinish and EmployeeCity may have the identical data type, they don't come from the same domain of values, and there is no conceivable business reason to link products and employees on these columns. However, one might conceive of some reason to understand the sales booked by a salesperson by looking at order dates of the person's sales relative to his or her hire date. It is amazing what questions SQL can answer (although we have limited control on how SQL displays the results).

Subqueries

The preceding SQL examples illustrate one of the two basic approaches for joining two tables: the joining technique. SQL also provides the subquery technique, which involves placing an inner query (SELECT . . . FROM . . . WHERE) within a WHERE or HAVING clause of another (outer) query. The inner query provides a set of one or more values for the search condition of the outer query. Such queries are referred to as subqueries or nested subqueries. Subqueries can be nested multiple times. Subqueries are prime examples of why SQL is a set-oriented language.

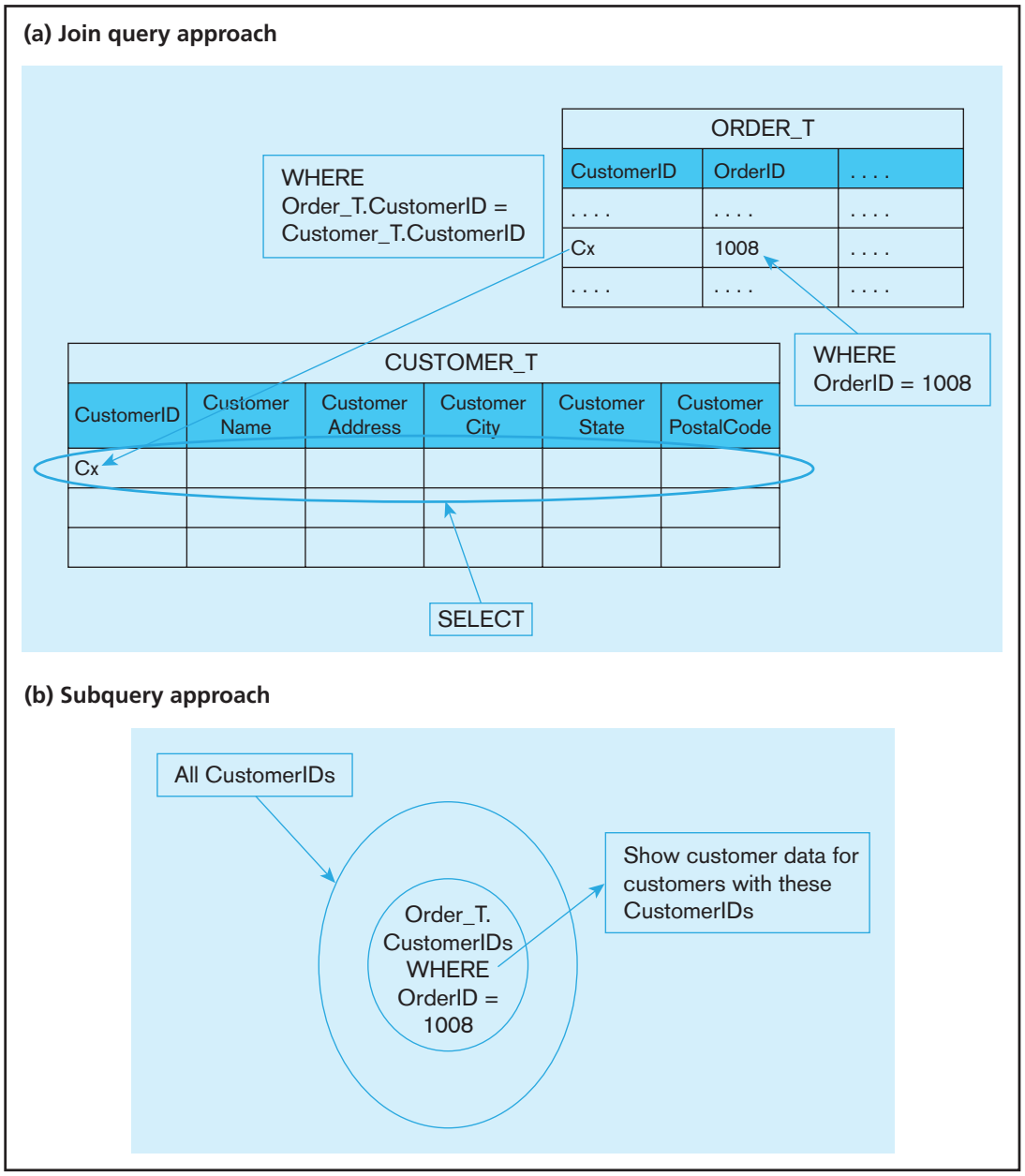
Sometimes, either the joining or the subquery technique can be used to accomplish the same result, and different people will have different preferences about which technique to use. Other times, only a join or only a subquery will work. The joining technique is useful when data from *several relations* are to be retrieved and displayed, and the relationships are not necessarily nested, whereas the subquery technique allows you to display data from only the tables mentioned in the outer query. Let's compare two queries that return the same result. Both answer the question, what is the name and address of the customer who placed order number 1008? First, we will use a join query, which is graphically depicted in Figure 7-5a.

> ***Query:*** What are the name and address of the customer who placed order number 1008?

```
SELECT CustomerName, CustomerAddress, CustomerCity,
  CustomerState, CustomerPostalCode
FROM Customer_T, Order_T
WHERE Customer_T.CustomerID = Order_T. CustomerID
  AND OrderID = 1008;
```

In set-processing terms, this query finds the subset of the Order_T table for OrderID = 1008 and then matches the row(s) in that subset with the rows in the Customer_T table that have the same CustomerID values. In this approach, it is not necessary that only one order have the OrderID value 1008. Now, look at the equivalent query using the subquery technique, which is graphically depicted in Figure 7-5b.

FIGURE 7-5 Graphical depiction of two ways to answer a query with different types of joins



Query: What are the name and address of the customer who placed order number 1008?

```
SELECT CustomerName, CustomerAddress, CustomerCity,
CustomerState, CustomerPostalCode
  FROM Customer_T
    WHERE Customer_T.CustomerID =
        (SELECT Order_T.CustomerID
          FROM Order_T
            WHERE OrderID = 1008);
```

Notice that the subquery, shaded in blue and enclosed in parentheses, follows the form learned for constructing SQL queries and could stand on its own as an independent query. That is, the result of the subquery, as with any other query, is a set of rows—in this case, a set of CustomerID values. We know that only one value will be in the result. (There is only one CustomerID for the order with OrderID 1008.) To be safe, we

can, and probably should, use the IN operator rather than = when writing subqueries. *The subquery approach may be used for this query because we need to display data from only the table in the outer query*. The value for OrderID does not appear in the query result; it is used as the selection criterion in the inner query. To include data from the subquery in the result, use the join technique, because data from a subquery cannot be included in the final results.

As noted previously, we know in advance that the preceding subquery will return at most one value, the CustomerID associated with OrderID 1008. The result will be empty if an order with that ID does not exist. (It is advisable to check that your query will work if a subquery returns zero, one, or many values.) A subquery can also return a list (set) of values (with zero, one, or many entries) if it includes the keyword IN. *Because the result of the subquery is used to compare with one attribute (CustomerID, in this query), the select list of a subquery may include only one attribute*. For example, which customers have placed orders? Here is a query that will answer that question.

Query: What are the names of customers who have placed orders?

```
SELECT CustomerName
  FROM Customer_T
    WHERE CustomerID IN
       (SELECT DISTINCT CustomerID
       FROM Order_T);
```

This query produces the following result. As required, the subquery select list contains only the one attribute, CustomerID, needed in the WHERE clause of the outer query. Distinct is used in the subquery because we do not care how many orders a customer has placed, as long as they have placed an order. For each customer identified in the Order_T table, that customer's name has been returned from Customer_T. (You will study this query again in Figure 7-7a.)

Result:

CUSTOMERNAME
Contemporary Casuals
Value Furniture
Home Furnishings
Eastern Furniture
Impressions
California Classics
American Euro Lifestyles
Battle Creek Furniture
Mountain Scenes
9 rows selected.

The qualifiers NOT, ANY, and ALL may be used in front of IN or with logical operators such as = , >, and <. Because IN works with zero, one, or many values from the inner query, many programmers simply use IN instead of = for all queries, even if the equal sign would work. The next example shows the use of NOT, and it also demonstrates that a join can be used in an inner query.

Query: Which customers have not placed any orders for computer desks?

```
SELECT CustomerName
  FROM Customer_T
  WHERE CustomerID NOT IN
  (SELECT CustomerID
    FROM Order_T, OrderLine_T, Product_T
      WHERE Order_T.OrderID = OrderLine_T.OrderID
        AND OrderLine_T.ProductID = Product_T.ProductID
        AND ProductDescription = 'Computer Desk');
```

Result:

CUSTOMERNAME
Value Furniture
Home Furnishings
Eastern Furniture
Furniture Gallery
Period Furniture
M & H Casual Furniture
Seminole Interiors
American Euro Lifestyles
Heritage Furnishings
Kaneohe Homes
10 rows selected.

The result shows that 10 customers have not yet ordered computer desks. The inner query returned a list (set) of all customers who had ordered computer desks. The outer query listed the names of those customers who were not in the list returned by the inner query. Figure 7-6 graphically breaks out the results of the subquery and main query.

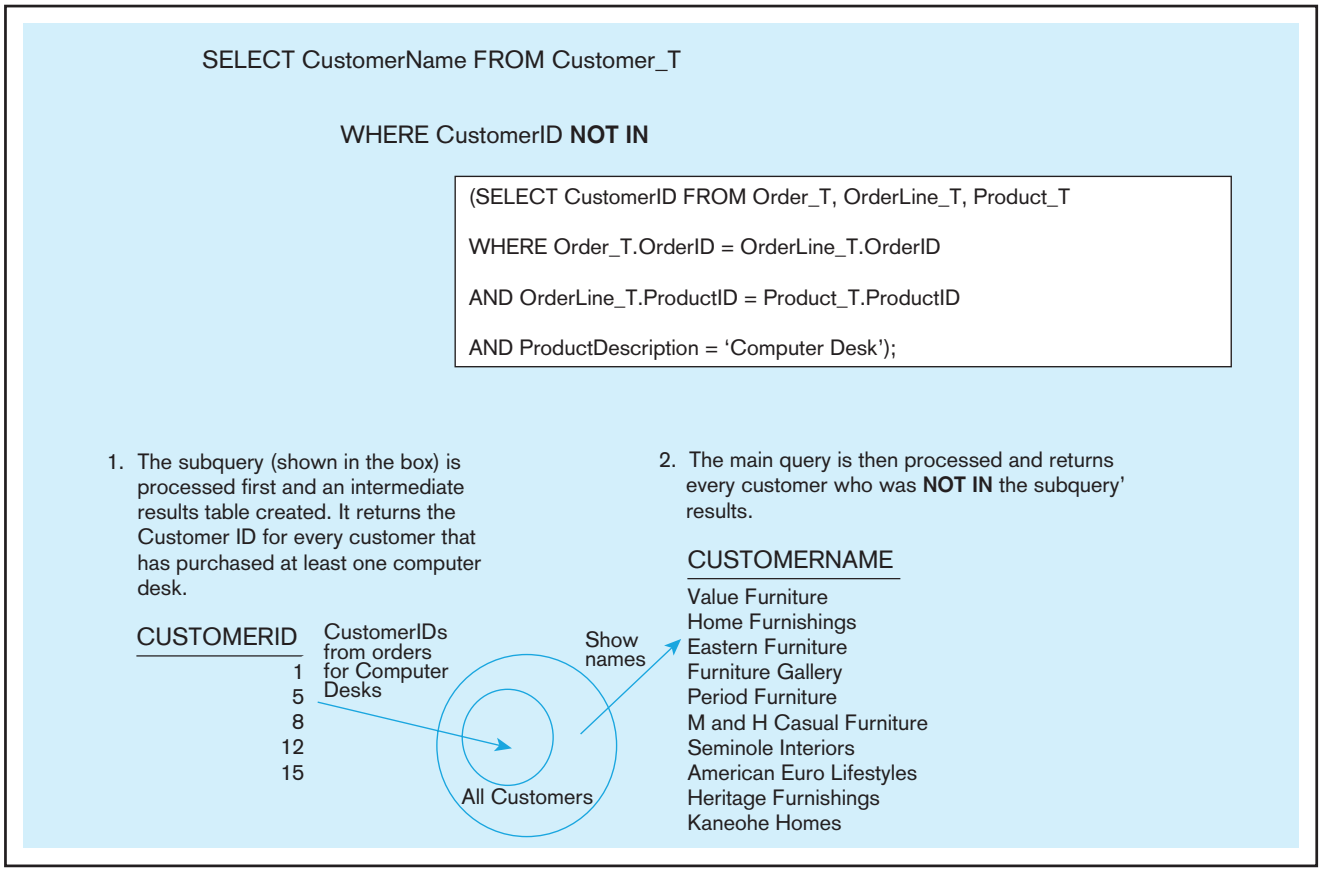
Qualifications such as < ANY or >= ALL instead of IN are also useful. For example, the qualification >= ALL can be used to match with the maximum value in a set. But be careful: Some combinations of qualifications may not make sense, such as = ALL (which makes sense only when the all the elements of the set have the same value).

Two other conditions associated with using subqueries are EXISTS and NOT EXISTS. These keywords are included in an SQL query at the same location where IN would be, just prior to the beginning of the subquery. EXISTS will take a value of *true* if the subquery returns an intermediate result table that contains one or more rows (i.e., a nonempty set) and *false* if no rows are returned (i.e., an empty set). NOT EXISTS will take a value of *true* if no rows are returned and *false* if one or more rows are returned.

So, when do you use EXISTS versus IN, and when do you use NOT EXISTS versus NOT IN? You use EXISTS (NOT EXISTS) when your only interest is whether the subquery returns a nonempty (empty) set (i.e., you don't care what is in the set, just whether it is empty), and you use IN (NOT IN) when you need to know what values are (are not) in the set. Remember, IN and NOT IN return a set of values from only one column, which can then be compared to one column in the outer query. EXISTS and NOT EXISTS returns only a true or false value depending on whether there are any rows in the answer table of the inner query or subquery.

Consider the following SQL statement, which includes EXISTS.

FIGURE 7-6 **Using the NOT IN qualifier**



Query: What are the order IDs for all orders that have included furniture finished in natural ash?

```
SELECT DISTINCT OrderID FROM OrderLine_T
WHERE EXISTS
  (SELECT *
    FROM Product_T
      WHERE ProductID = OrderLine_T.ProductID
      AND ProductFinish = 'Natural Ash');
```

The subquery is executed for each order line in the outer query. The subquery checks for each order line to see if the finish for the product on that order line is natural ash (indicated by the arrow added to the query above). If this is true (EXISTS), the outer query displays the order ID for that order. The outer query checks this one row at a time for every row in the set of referenced rows (the OrderLine_T table). There have been seven such orders, as the result shows. (We discuss this query further in Figure 7-7b.)

Result:

ORDERID
1001
1002
1003
1006
1007
1008
1009

7 rows selected.

When EXISTS or NOT EXISTS is used in a subquery, the select list of the subquery will usually just select all columns (SELECT *) as a placeholder because it does not matter which columns are returned. The purpose of the subquery is to test whether any rows fit the conditions, not to return values from particular columns for comparison purposes in the outer query. The columns that will be displayed are determined strictly by the outer query. The EXISTS subquery illustrated previously, like almost all other EXISTS subqueries, is a correlated subquery, which is described next. Queries containing the keyword NOT EXISTS will return a result table when no rows are found that satisfy the subquery.

In summary, use the subquery approach when qualifications are nested or when qualifications are easily understood in a nested way. Most systems allow pairwise joining of *one and only one column* in an inner query with one column in an outer query. An exception to this is when a subquery is used with the EXISTS keyword. Data can be displayed only from the table(s) referenced in the outer query. Up to 16 levels of nesting are typically supported. Queries are processed from the inside out, although another type of subquery, a correlated subquery, is processed from the outside in.

Correlated Subqueries

In the first subquery example in the prior section, it was necessary to examine the inner query before considering the outer query. That is, the result of the inner query was used to limit the processing of the outer query. In contrast, **correlated subqueries** use the result of the outer query to determine the processing of the inner query. That is, the inner query is somewhat different for each row referenced in the outer query. In this case, the inner query must be computed for *each* outer row, whereas in the earlier examples, the inner query was computed *only once* for all rows processed in the outer query. The EXISTS subquery example in the prior section had this characteristic, in which the inner query was executed for each OrderLine_T row, and each time it was executed, the inner query was for a different ProductID value—the one from the OrderLine_T row in the outer query. Figures 7-7a and 7-7b depict the different processing order for each of the examples from the previous section on subqueries.

Correlated subquery
In SQL, a subquery in which processing the inner query depends on data from the outer query.

Let's consider another example query that requires composing a correlated subquery.

Query: List the details about the product with the highest standard price.

```
SELECT ProductDescription, ProductFinish, ProductStandardPrice
FROM Product_T PA
  WHERE PA.ProductStandardPrice > ALL
    (SELECT ProductStandardPrice FROM Product_T PB
      WHERE PB.ProductID ! = PA.ProductID);
```

As you can see in the following result, the dining table has a higher unit price than any other product.

Result:

PRODUCTDESCRIPTION	PRODUCTFINISH	PRODUCTSTANDARDPRICE
Dining Table	Natural Ash	800

The arrow added to the query above illustrates the cross-reference for a value in the inner query to be taken from a table in the outer query. The logic of this SQL statement is that the subquery will be executed once for each product to be sure that no other product has a higher standard price. Notice that we are comparing rows in a table to themselves and that we are able to do this by giving the table two aliases, PA and PB; you'll recall we identified this earlier as a self-join. First, ProductID 1, the end table, will be considered. When the subquery is executed, it will return a set of values, which are the standard prices of every product except the one being considered in the outer query (product 1, for the first time it is executed). Then, the outer query will check to see if the standard price for the product being considered is greater than all of the standard prices returned by the subquery. If it is, it will be returned as the result of the query. If not, the next standard price value in the outer query will be considered, and the inner query

FIGURE 7-7 Subquery processing

(a) Processing a noncorrelated subquery

What are the names of customers who have placed orders?

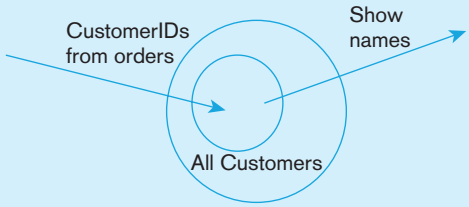
```
SELECT CustomerName
    FROM Customer_T
        WHERE CustomerID IN
```

```
(SELECT DISTINCT CustomerID
 FROM Order_T);
```

1. The subquery (shown in the box) is processed first and an intermediate results table created:

CUSTOMERID
1
8
15
5
3
2
11
12
4

9 rows selected.



2. The outer query returns the requested customer information for each customer included in the intermediate results table:

CUSTOMERNAME
Contemporary Casuals
Value Furniture
Home Furnishings
Eastern Furniture
Impressions
California Classics
American Euro Lifestyles
Battle Creek Furniture
Mountain Scenes

9 rows selected.

(continued)

will return a list of all the standard prices for the other products. The list returned by the inner query changes as each product in the outer query changes; that makes it a correlated subquery. Can you identify a special set of standard prices for which this query will not yield the desired result (see Problem and Exercise 38)?

Using Derived Tables

Subqueries are not limited to inclusion in the WHERE clause. As we saw in Chapter 6, they may also be used in the FROM clause to create a temporary derived table (or set) that is used in the query. Creating a derived table that has an aggregate value in it, such as MAX, AVG, or MIN, allows the aggregate to be used in the WHERE clause. Here, pieces of furniture that exceed the average standard price are listed.

Query: Show the product description, product standard price, and overall average standard price for all products that have a standard price that is higher than the average standard price.

```
SELECT ProductDescription, ProductStandardPrice, AvgPrice
  FROM
     (SELECT AVG(ProductStandardPrice) AvgPrice FROM Product_T),
        Product_T
  WHERE ProductStandardPrice > AvgPrice;
```

Result:

PRODUCTDESCRIPTION	PRODUCTSTANDARDPRICE	AVGPRICE
Entertainment Center	650	440.625
8-Drawer Dresser	750	440.625
Dining Table	800	440.625

FIGURE 7-7 ***(continued)***

(b) Processing a correlated subquery

What are the order IDs for all orders that have included furniture finished in natural ash?

```
SELECT DISTINCT OrderID FROM OrderLine_T
WHERE EXISTS
        (SELECT *
          FROM Product _T
                WHERE ProductID = OrderLine_T.ProductID
                    AND Productfinish = 'Natural Ash');
```

	OrderID	ProductID	OrderedQuantity
1 →	1001	1	1
	1001	2	2
	1001	4	1
3 →	1002	3	5
	1003	3	3
	1004	6	2
	1004	8	2
	1005	4	4
	1006	4	1
	1006	5	2
	1007	1	3
	1007	2	2
	1008	3	3
	1008	8	3
	1009	4	2
	1009	7	3
	1010	8	10
*	0	0	0

	ProductID	ProductDescription	ProductFinish	ProductStandardPrice	ProductLineID
▶ ⊞	1	End Table	Cherry	$175.00	10001
⊞	2 → 2	Coffee Table	Natural Ash	$200.00	20001
⊞	4 → 3	Computer Desk	Natural Ash	$375.00	20001
⊞	4	Entertainment Center	Natural Maple	$650.00	30001
⊞	5	Writer's Desk	Cherry	$325.00	10001
⊞	6	8-Drawer Dresser	White Ash	$750.00	20001
⊞	7	Dining Table	Natural Ash	$800.00	20001
⊞	8	Computer Desk	Walnut	$250.00	30001
*	(AutoNumber)			$0.00	

1. The first order ID is selected from OrderLine_T: OrderID =1001.
2. The subquery is evaluated to see if any product in that order has a natural ash finish. Product 2 does, and is part of the order. EXISTS is valued as *true* and the order ID is added to the result table.
3. The next order ID is selected from OrderLine_T: OrderID =1002.
4. The subquery is evaluated to see if the product ordered has a natural ash finish. It does. EXISTS is valued as *true* and the order ID is added to the result table.
5. Processing continues through each order ID. Orders 1004, 1005, and 1010 are not included in the result table because they do not include any furniture with a natural ash finish. The final result table is shown in the text on page 302.

So, why did this query require a derived table rather than, say, a subquery? The reason is we want to display both the standard price and the average standard price for each of the selected products. The similar query in the prior section on correlated subqueries worked fine to show data from only the table in the outer query, the product table. However, to show both standard price and the average standard price in each displayed row, we have to get both values into the "outer" query, as is done in the query above.

Combining Queries

Sometimes, no matter how clever you are, you can't get all the rows you want into the single answer table using one SELECT statement. Fortunately, you have a lifeline! The UNION clause is used to combine the output (i.e., union the set of rows) from multiple queries together into a single result table. To use the UNION clause, each query involved must output the same number of columns, and they must be UNION compatible. This means that the output from each query for each column should be of compatible data types. Acceptance as a compatible data type varies among the DBMS products. When performing a union where output for a column will merge two different data types, it is safest to use the CAST command to control the data type conversion

yourself. For example, the DATE data type in Order_T might need to be converted into a text data type. The following SQL command would accomplish this:

```
SELECT CAST (OrderDate AS CHAR) FROM Order_T;
```

The following query determines the customer(s) who has in a given line item purchased the largest quantity of any Pine Valley product and the customer(s) who has in a given line item purchased the smallest quantity and returns the results in one table.

Query:

```
SELECT C1.CustomerID, CustomerName, OrderedQuantity,
'Largest Quantity' AS Quantity
FROM Customer_T C1,Order_T O1, OrderLine_T Q1
     WHERE C1.CustomerID = O1.CustomerID
     AND O1.OrderID = Q1.OrderID
     AND OrderedQuantity =
     (SELECT MAX(OrderedQuantity)
     FROM OrderLine_T)
UNION
SELECT C1.CustomerID, CustomerName, OrderedQuantity,
'Smallest Quantity'
FROM Customer_T C1, Order_T O1, OrderLine_T Q1
     WHERE C1.CustomerID = O1.CustomerID
     AND O1.OrderID = Q1.OrderID
     AND OrderedQuantity =
          (SELECT MIN(OrderedQuantity)
          FROM OrderLine_T)
ORDER BY 3;
```

Notice that an expression Quantity has been created in which the strings 'Smallest Quantity' and 'Largest Quantity' have been inserted for readability. The ORDER BY clause has been used to organize the order in which the rows of output are listed. Figure 7-8 breaks the query into parts to help you understand how it processes.

Result:

CUSTOMERID	CUSTOMERNAME	ORDEREDQUANTITY	QUANTITY
1	Contemporary Casuals	1	Smallest Quantity
2	Value Furniture	1	Smallest Quantity
1	Contemporary Casuals	10	Largest Quantity

Did we have to answer this question by using UNION? Could we instead have answered it using one SELECT and a complex, compound WHERE clause with many ANDs and ORs? In general, the answer is sometimes (another good academic answer, like "it depends"). Often, it is simply easiest to conceive of and write a query using several simply SELECTs and a UNION. Or, if it is a query you frequently run, maybe one way will process more efficiently than another. You will learn from experience which approach is most natural for you and best for a given situation.

Now that you remember the union set operation from finite mathematics, you may also remember that there are other set operations—intersect (to find the elements in common between two sets) and minus (to find the elements in one set that are not in

FIGURE 7-8 Combining queries using UNION

```
SELECT C1.CustomerID, CustomerName, OrderedQuantity, 'Largest Quantity' AS Quantity
    FROM Customer_T C1,Order_T O1, OrderLine_T Q1
    WHERE C1.CustomerID = O1.CustomerID
              AND O1.OrderID = Q1.OrderID
              AND OrderedQuantity =
                              (SELECT MAX(OrderedQuantity)
                              FROM OrderLine_T)
```

1. In the above query, the subquery is processed first and an intermediate results table created. It contains the maximum quantity ordered from OrderLine_T and has a value of 10.
2. Next the main query selects customer information for the customer or customers who ordered 10 of any item. Contemporary Casuals has ordered 10 of some unspecified item.

```
SELECT C1.CustomerID, CustomerName, OrderedQuantity, 'Smallest Quantity'
    FROM Customer_T C1, Order_T O1, OrderLine_T Q1
    WHERE C1.CustomerID = O1.CustomerID
              AND O1.OrderID = Q1.OrderID
              AND OrderedQuantity =
                              (SELECT MIN(OrderedQuantity)
                               FROM OrderLine_T)
ORDER BY 3;
```

1. In the second main query, the same process is followed but the result returned is for the minimum order quantity.
2. The results of the two queries are joined together using the UNION command.
3. The results are then ordered according to the value in OrderedQuantity. The default is ascending value, so the orders with the smallest quantity, 1, are listed first.

another set). These operations—INTERSECT and MINUS—are also available in SQL, and they are used just as UNION was above to manipulate the result sets created by two SELECT statements.

Conditional Expressions

Establishing IF-THEN-ELSE logical processing within an SQL statement can now be accomplished by using the CASE keyword in a statement. Figure 7-9 gives the CASE syntax, which actually has four forms. The CASE form can be constructed using either an expression that equates to a value or a predicate. The predicate form is based on three-value logic (true, false, don't know) but allows for more complex operations. The value-expression form requires a match to the value expression. NULLIF and COALESCE are the keywords associated with the other two forms of the CASE expression.

```
{CASE expression
{WHEN expression
THEN {expression     |  NULL}}  . . .
|  {WHEN predicate
THEN {expression     |  NULL}}  . . .
[ELSE {expression      NULL}]
END  }
|  ( NULLIF (expression, expression)      }
|  ( COALESCE (expression      . . .) }
```

FIGURE 7-9 CASE conditional syntax

CASE could be used in constructing a query that asks "What products are included in Product Line 1?" In this example, the query displays the product description for each product in the specified product line and a special text, '####' for all other products, thus displaying a sense of the relative proportion of products in the specified product line.

Query:

```
SELECT CASE
    WHEN ProductLine = 1 THEN ProductDescription
    ELSE '####'
END AS ProductDescription
FROM Product_T;
```

Result:

PRODUCTDESCRIPTION
End Table
####
####
####
Writers Desk
####
####
####

Gulutzan & Pelzer (1999, p. 573) indicate that "It's possible to use CASE expressions this way as retrieval substitutes, but the more common applications are (a) to make up for SQL's lack of an enumerated <data type>, (b) to perform complicated if/then calculations, (c) for translation, and (d) to avoid exceptions. We find CASE expressions to be indispensable, and it amazes us that in pre SQL-92 DBMSs they didn't exist."

More Complicated SQL Queries

We have kept the examples used in Chapter 6 and this chapter very simple in order to make it easier for you to concentrate on the piece of SQL syntax being introduced. It is important to understand that production databases may contain hundreds and even thousands of tables, and many of those contain hundreds of columns. While it is difficult to come up with complicated queries from the four tables used in Chapter 6 and this chapter, the text comes with a larger version of the Pine Valley Furniture Company database, which allows for somewhat more complex queries. This version is available at **www.prenhall.com/hoffer** and at **www.teradatastudentnetwork.com**; here are two samples drawn from that database:

Question 1: For each salesperson, list his or her biggest-selling product.

Query: First, we will define a view called TSales, which computes the total sales of each product sold by each salesperson. We create this view to simplify answering this query by breaking it into several easier-to-write queries.

```
CREATE VIEW TSales AS
SELECT SalespersonName,
    ProductDescription,
    SUM(OrderedQuantity) AS Totorders
```

```
FROM Salesperson_T, OrderLine_T, Product_T, Order_T
    WHERE Salesperson_T.SalespersonID=Order_T.SalespersonID
    AND Order_T.OrderID=OrderLine_T.OrderID
    AND OrderLine_T.ProductID=Product_T.ProductID
    GROUP BY SalespersonName, ProductDescription;
```

Next we write a correlated subquery using the view:

```
SELECT SalespersonName, ProductDescription
    FROM TSales AS A
        WHERE Totorders = (SELECT MAX(Totorders) FROM TSales B
        WHERE B.SalesperssonName = A.SalespersonName);
```

Notice that once we had the TSales view, the correlated subquery was rather simple to write. Also, it was simple to conceive of the final query once all the data needed to display were all in the set created by the virtual table (set) of the view. Our thought process was if we could create a set of information about the total sales for each salesperson, we could then find the maximum value of total sales in this set. Then it is simply a matter of scanning that set to see which salesperson(s) has total sales equal to that maximum value. There are likely other ways to write SQL statements to answer this question, so use whatever approach works and is most natural for you. We suggest that you draw diagrams, like those you have seen in figures in this chapter, to represent the sets you think you could manipulate to answer the question you face.

Question 2: Write an SQL query to list all salespersons who work in the territory where the most end tables have been sold.

Query: First, we will create a query called TopTerritory, using the following SQL statement:

```
SELECT TOP 1 Territory_T.TerritoryID,
SUM(OrderedQuantity) AS TopSales
    FROM Territory_T INNER JOIN (Product_T INNER JOIN
      (((Customer_T INNER JOIN DoesBusinessIn_T ON
      Customer_T.CustomerID = DoesBusinessIn_T.CustomerID)
      INNER JOIN Order_T ON Customer_T.CustomerID =
      Order_T.CustomerID) INNER JOIN OrderLine_T ON
      Order_T.OrderID = OrderLine_T.OrderID) ON
      Product_T.ProductID = OrderLine_T.ProductID) ON
      Territory_T.TerritoryID = DoesBusinessIn_T.TerritoryID
      WHERE ((ProductDescription)='End Table')
      GROUP BY Territory_T.TerritoryID
      ORDER BY TotSales DESC;
```

This will give us the territory number of the top-producing territory for sales of end tables.

Next, we will write a query using this query as a derived table. (To save space, we simply insert the name we used for the above query, but SQL requires that the above query be inserted as a derived table where its name appears in the query below. Alternatively, TopTerritory could have been created as a view.)

```
SELECT Salesperson_T.SalespersonID, SalesperspmName
    FROM Territory_T INNER JOIN Salesperson_T ON
      Territory_T.TerritoryID = Salesperson_T.TerritoryID
    WHERE Salesperson_T.TerritoryID IN
      (SELECT TerritoryID FROM TopTerritory);
```

You probably noticed the use of the TOP operator in the TopTerritory query above. TOP, which is compliant with the SQL:2003 standard, specifies a given number or percentage of the rows (with or without ties, as indicated by a subclause) to be returned from the ordered query result set.

TIPS FOR DEVELOPING QUERIES

SQL's simple basic structure results in a query language that is easy for a novice to use to write simple ad hoc queries. At the same time, it has enough flexibility and syntax options to handle complicated queries used in a production system. Both characteristics, however, lead to potential difficulties in query development. As with any other computer programming, you are likely not to write a query correctly the first time. Be sure you have access to an explanation of the error codes generated by the RDBMS. Work initially with a test set of data, usually small, for which you can compute the desired answer by hand as a way to check your coding. This is especially true if you are writing INSERT, UPDATE, or DELETE commands, and it is why organizations have test, development, and production versions of a database, so inevitable development errors do not harm production data.

First, as a novice query writer, you will find it easy to write a query that runs without error. Congratulations, but the results may not be exactly what you intended. Sometimes it will be obvious to you that there is a problem, especially if you forget to define the links between tables with a WHERE clause and get a Cartesian join of all possible combinations of records. Other times, your query will appear to be correct, but close inspection using a test set of data may reveal that your query returns 24 rows when it should return 25. Sometimes it will return duplicates you don't want or just a few of the records you want, and sometimes it won't run because you are trying to group data that can't be grouped. Watch carefully for these types of errors before you turn in your homework. Working through a well-thought-out set of test data by hand will help you to catch your errors. When you are constructing a set of test data, include some examples of common data values. Then think about possible exceptions that could occur. For example, real data might unexpectedly include null data, out-of-range data, or impossible data values.

Certain steps are necessary in writing any query. The graphical interfaces now available make it easier to construct queries and to remember table and attribute names as you work. Here are some suggestions to help you (we assume that you are working with a database that has been defined and created):

- Familiarize yourself with the data model and the entities and relationships that have been established. The data model expresses many of the business rules that may be idiosyncratic for the business or problem you are considering. It is very important to have a good grasp of the data that are available with which to work. As demonstrated in Figures 7-7a and 7-7b, you can draw the segment of the data model you intend to reference in the query and then annotate it to show qualifications and joining criteria. Alternatively you can draw figures such as Figures 7-5 and 7-6 with sample data and Venn diagrams to also help conceive of how to construct subqueries or derived tables that can be used as components in a more complex query.
- Be sure that you understand what results you want from your query. Often, a user will state a need ambiguously, so be alert and address any questions you have after working with users.
- Figure out what attributes you want in your query result. Include each attribute after the SELECT keyword.
- Locate within the data model the attributes you want and identify the entity where the required data are stored. Include these after the FROM keyword.
- Review the ERD and all the entities identified in the previous step. Determine what columns in each table will be used to establish the relationships. Consider what type of join you want between each set of entities.

- Construct a WHERE equality for each link. Count the number of entities involved and the number of links established. Usually there will be one more entity than there are WHERE clauses. When you have established the basic result set, the query may be complete. In any case, run it and inspect your results.
- When you have a basic result set to work with, you can begin to fine-tune your query by adding GROUP BY and HAVING clauses, DISTINCT, NOT IN, and so forth. Test your query as you add keywords to it to be sure you are getting the results you want.
- Until you gain query writing experience, your first draft of a query will tend to work with the data you expect to encounter. Now, try to think of exceptions to the usual data that may be encountered and test your query against a set of test data that includes unusual data, missing data, impossible values, and so forth. If you can handle those, your query is almost complete. Remember that checking by hand will be necessary; just because an SQL query runs doesn't mean it is correct.

As you start to write more complicated queries using additional syntax, debugging queries may be more difficult for you. If you are using subqueries, errors of logic can often be located by running each subquery as a freestanding query. Start with the subquery that is nested most deeply. When its results are correct, use that tested subquery with the outer query that uses its result. You can follow a similar process with derived tables. Follow this procedure until you have tested the entire query. If you are having syntax trouble with a simple query, try taking apart the query to find the problem. You may find it easier to spot a problem if you return just a few crucial attribute values and investigate one manipulation at a time.

As you gain more experience, you will be developing queries for larger databases. As the amount of data that must be processed increases, the time necessary to successfully run a query may vary noticeably, depending on how you write the query. Query optimizers are available in the more powerful database management systems such as Oracle, but there are also some simple strategies for writing queries that may prove helpful for you. The following are some common strategies to consider if you want to write queries that run more efficiently:

- Rather than use the SELECT * option, take the time to include the column names of the attributes you need in a query. If you are working with a wide table and need only a few of the attributes, using SELECT * may generate a significant amount of unnecessary network traffic as unnecessary attributes are fetched over the network. Later, when the query has been incorporated into a production system, changes in the base table may affect the query results. Specifying the attribute names will make it easier to notice and correct for such events.
- Try to build your queries so that your intended result is obtained from one query. Review your logic carefully to reduce the number of subqueries in the query as much as possible. Each subquery you include requires the DBMS to return an interim result set and integrate it with the remaining subqueries, thus increasing processing time.
- Sometimes data that reside in one table will be needed for several separate reports. Rather than obtain those data in several separate queries, create a single query that retrieves all the data that will be needed; you reduce the overhead by having the table accessed once rather than repeatedly. It may help you to recognize such a situation by thinking about the data that are typically used by a department and creating a view for the department's use.

Guidelines for Better Query Design

Now you have some strategies for developing queries that will give you the results you want. But will these strategies result in efficient queries, or will they result in the "query from hell," giving you plenty of time for the pizza to be delivered, to watch the *Star Trek*

anthology, or to organize your closet? Various database experts, such as DeLoach (1987) and Holmes (1996), provide suggestions for improving query processing in a variety of settings. Also see the Web Resources at the end of this chapter and prior chapters for links to sites where query design suggestions are continually posted. We summarize here some of their suggestions that apply to many situations:

1. ***Understand how indexes are used in query processing*** Many DBMSs will use only one index per table in a query—often the one that is the most discriminating (i.e., has the most key values). Some will never use an index with only a few values compared to the number of table rows. Others may balk at using an index for which the column has many null values across the table rows. Monitor accesses to indexes and then drop indexes that are infrequently used. This will improve the performance of database update operations. In general, queries that have equality criteria for selecting table rows (e.g., WHERE Finish = "Birch" OR "Walnut") will result in faster processing than queries involving more complex qualifications do (e.g., WHERE Finish NOT = "Walnut") because equality criteria can be evaluated via indexes.
2. ***Keep optimizer statistics up-to-date*** Some DBMSs do not automatically update the statistics needed by the query optimizer. If performance is degrading, force the running of an update-statistics-like command.
3. ***Use compatible data types for fields and literals in queries*** Using compatible data types will likely mean that the DBMS can avoid having to convert data during query processing.
4. ***Write simple queries*** Usually the simplest form of a query will be the easiest for a DBMS to process. For example, because relational DBMSs are based on set theory, write queries that manipulate sets of rows and literals.
5. ***Break complex queries into multiple simple parts*** Because a DBMS may use only one index per query, it is often good to break a complex query into multiple, simpler parts (which each use an index) and then combine together the results of the smaller queries. For example, because a relational DBMS works with sets, it is very easy for the DBMS to UNION two sets of rows that are the result of two simple, independent queries.
6. ***Don't nest one query inside another query*** Usually, nested queries, especially correlated subqueries, are less efficient than a query that avoids subqueries to produce the same result. This is another case where using UNION, INTERSECT, or MINUS and multiple queries may produce results more efficiently.
7. ***Don't combine a table with itself*** Avoid, if possible, using self-joins. It is usually better (i.e., more efficient for processing the query) to make a temporary copy of a table and then to relate the original table with the temporary one. Temporary tables, because they quickly get obsolete, should be deleted soon after they have served their purpose.
8. ***Create temporary tables for groups of queries*** When possible, reuse data that are used in a sequence of queries. For example, if a series of queries all refer to the same subset of data from the database, it may be more efficient to first store this subset in one or more temporary tables and then refer to those temporary tables in the series of queries. This will avoid repeatedly combining the same data together or repeatedly scanning the database to find the same database segment for each query. The trade-off is that the temporary tables will not change if the original tables are updated when the queries are running. Using temporary tables is a viable substitute for derived tables, and they are created only once for a series of references.
9. ***Combine update operations*** When possible, combine multiple update commands into one. This will reduce query processing overhead and allow the DBMS to seek ways to process the updates in parallel.
10. ***Retrieve only the data you need*** This will reduce the data accessed and transferred. This may seem obvious, but there are some shortcuts for query writing that violate this guideline. For example, in SQL the command SELECT * from EMP will retrieve all the fields from all the rows of the EMP table. But, if the user needs to see only some of the columns of the table, transferring the extra columns increases the query processing time.

11. ***Don't have the DBMS sort without an index*** If data are to be displayed in sorted order and an index does not exist on the sort key field, then sort the data outside the DBMS after the unsorted results are retrieved. Usually a sort utility will be faster than a sort without the aid of an index by the DBMS.
12. ***Learn!*** Track query processing times, review query plans with the EXPLAIN command, and improve your understanding of the way the DBMS determines how to process queries. Attend specialized training from your DBMS vendor on writing efficient queries, which will better inform you about the query optimizer.
13. ***Consider the total query processing time for ad hoc queries*** The total time includes the time it takes the programmer (or end user) to write the query as well as the time to process the query. Many times, for ad hoc queries, it is better to have the DBMS do extra work to allow the user to more quickly write a query. And isn't that what technology is supposed to accomplish—to allow people to be more productive? So, don't spend too much time, especially for ad hoc queries, trying to write the most efficient query. Write a query that is logically correct (i.e., produces the desired results) and let the DBMS do the work. (Of course, do an EXPLAIN first to be sure you haven't written "the query from hell" so that all other users will see a serious delay in query processing time.) This suggests a corollary: When possible, run your query when there is a light load on the database, because the total query processing time includes delays induced by other load on the DBMS and database.

All options are not available with every DBMS, and each DBMS has unique options due to its underlying design. You should refer to reference manuals for your DBMS to know which specific tuning options are available to you.

ENSURING TRANSACTION INTEGRITY

RDBMSs are no different from other types of database managers in that one of their primary responsibilities is to ensure that data maintenance is properly and completely handled. Even with extensive testing, as suggested in the prior section, bad things can happen to good data managers: A data maintenance program may not work correctly because someone submitted the job twice, some unanticipated anomaly in the data occurred, or there was a computer hardware, software, or power malfunction during the transaction. Data maintenance is defined in units of work called *transactions*, which involve one or more data manipulation commands. A transaction is a complete set of closely related update commands that must all be done, or none of them done, for the database to remain valid. Consider Figure 7-10, for example. When an order is entered into the Pine Valley database, all of the items ordered should be entered at the same time. Thus, either all OrderLine_T rows from this form are to be entered, along with all the information in Order_T, or none of them should be entered. Here, the business transaction is the complete order, not the individual items that are ordered. What we need are commands to define the boundaries of a transaction, to commit the work of a transaction as a permanent change to the database, and to abort a transaction on purpose and correctly, if necessary. In addition, we need data recovery services to clean up after abnormal termination of database processing in the middle of a transaction. Perhaps the order form is accurate, but in the middle of entering the order, the computer system malfunctions or loses power. In this case, we do not want some of the changes made and not others. It's all or nothing at all if we want a valid database.

When a single SQL command constitutes a transaction, some RDBMSs will automatically commit or roll back after the command is run. With a user-defined transaction, however, where multiple SQL commands need to be run and either entirely committed or entirely rolled back, commands are needed to manage the transaction explicitly. Many systems will have BEGIN TRANSACTION and END TRANSACTION commands, which are used to mark the boundaries of a logical unit of work. BEGIN TRANSACTION creates a log file and starts recording all changes (insertions, deletions, and updates) to the database in this file. END TRANSACTION or COMMIT WORK takes the

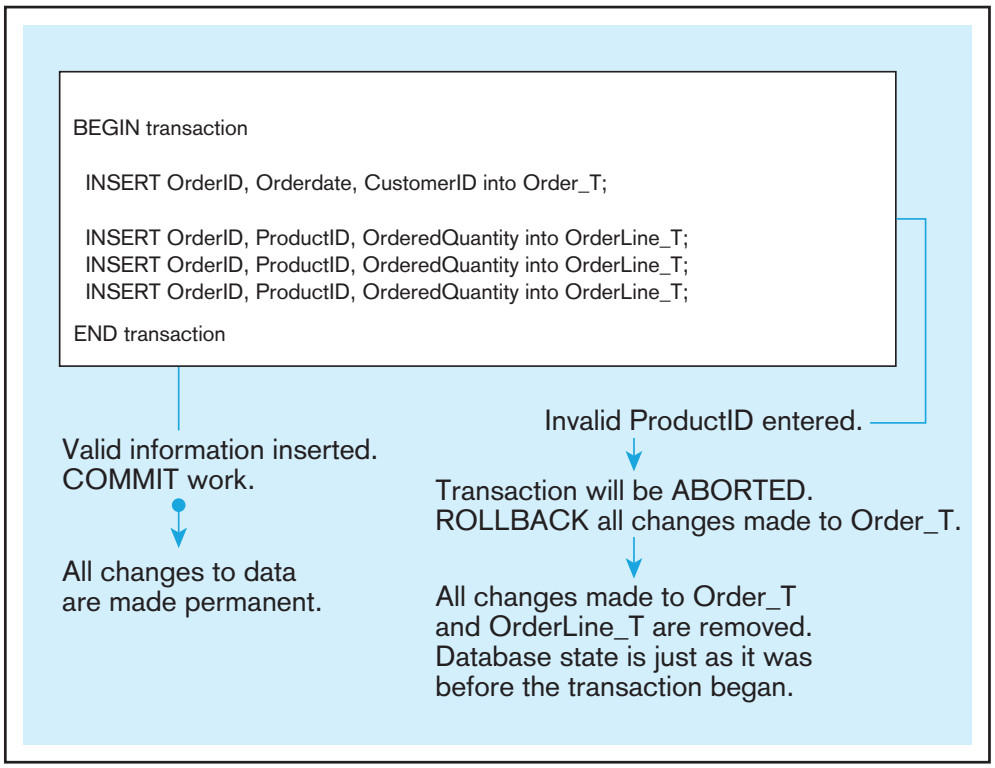


FIGURE 7-10 An SQL transaction sequence (in pseudocode)

contents of the log file and applies them to the database, thus making the changes permanent, and then empties the log file. ROLLBACK WORK asks SQL to empty the log file. Some RDBMSs also have an AUTOCOMMIT (ON/OFF) command that specifies whether changes are made permanent after each data modification command (ON) or only when work is explicitly made permanent (OFF) by the COMMIT WORK command.

User-defined transactions can improve system performance because transactions will be processed as sets rather than as individual transactions, thus reducing system overhead. When AUTOCOMMIT is set to OFF, changes will not be made automatically until the end of a transaction is indicated. When AUTOCOMMIT is set to ON, changes will be made automatically at the end of each SQL statement; this would not allow for user-defined transactions to be committed or rolled back only as a whole.

SET AUTOCOMMIT is an interactive command; therefore, a given user session can be dynamically controlled for appropriate integrity measures. Each SQL INSERT, UPDATE, and DELETE command typically works on only one table at a time. Some data maintenance requires updating of multiple tables for the work to be complete. Therefore, these transaction-integrity commands are important in clearly defining whole units of database changes that must be completed in full for the database to retain integrity.

Further, some SQL systems have concurrency controls that handle the updating of a shared database by concurrent users. These can journalize database changes so that a database can be recovered after abnormal terminations in the middle of a transaction. They can also undo erroneous transactions. For example, in a banking application, the update of a bank account balance by two concurrent users should be cumulative. Such controls are transparent to the user in SQL; no user programming is needed to ensure proper control of concurrent access to data. To ensure the integrity of a particular database, be sensitive to transaction integrity and recovery issues and make sure that application programmers are appropriately informed of when these commands are to be used.

DATA DICTIONARY FACILITIES

RDBMSs store database definition information in secure system-created tables; we can consider these system tables as a data dictionary. Becoming familiar with the systems tables for any RDBMS being used will provide valuable information, whether

you are a user or a database administrator. Because the information is stored in tables, it can be accessed by using SQL SELECT statements that can generate reports about system usage, user privileges, constraints, and so on. Also, the RDBMS will provide special SQL (proprietary) commands, such as SHOW, HELP, or DESCRIBE, to display predefined contents of the data dictionary, including the DDL that created database objects. Further, a user who understands the systems-table structure can extend existing tables or build other tables to enhance built-in features (e.g., to include data on who is responsible for data integrity). A user is, however, often restricted from modifying the structure or contents of the system tables directly, because the DBMS maintains them and depends on them for its interpretation and parsing of queries.

Each RDBMS keeps various internal tables for these definitions. In Oracle 11g, there are 522 data dictionary views for DBAs to use. Many of these views, or subsets of the DBA view (i.e., information relevant to an individual user), are also available to users who do not possess DBA privileges. Those view names begin with USER (anyone authorized to use the database) or ALL (any user) rather than DBA. Views that begin with V$ provide updated performance statistics about the database. Here is a list of some of the tables (accessible to DBAs) that keep information about tables, clusters, columns, and security. There are also tables related to storage, objects, indexes, locks, auditing, exports, and distributed environments.

Table	Description
DBA_TABLES	Describes all tables in the database
DBA_TAB_COMMENTS	Comments on all tables in the database
DBA_CLUSTERS	Describes all clusters in the database
DBA_TAB_COLUMNS	Describes columns of all tables, views, and clusters
DBA_COL_PRIVS	Includes all grants on columns in the database
DBA_COL_COMMENTS	Comments on all columns in tables and views
DBA_CONSTRAINTS	Constraint definitions on all tables in the database
DBA_CLU_COLUMNS	Maps table columns to cluster columns
DBA_CONS_COLUMNS	Information about all columns in constraint definitions
DBA_USERS	Information about all users of the database
DBA_SYS_PRIVS	Describes system privileges granted to users and to roles
DBA_ROLES	Describes all roles that exist in the database
DBA_PROFILES	Includes resource limits assigned to each profile
DBA_ROLE_PRIVS	Describes roles granted to users and to other roles
DBA_TAB_PRIVS	Describes all grants on objects in the database

To give an idea of the type of information found in the system tables, consider DBA_USERS. DBA_USERS contains information about the valid users of the database; its 12 attributes include user name, user ID, encrypted password, default tablespace, temporary tablespace, date created, and profile assigned. DBA_TAB_COLUMNS has 31 attributes, including owner of each table, table name, column name, data type, data length, precision, and scale, among others. An SQL query against DBA_TABLES to find out who owns PRODUCT_T follows. (Note that we have to specify PRODUCT_T, not Product_T, because Oracle stores data names in all capital letters.)

Query: Who is the owner of the PRODUCT_T table?

```
SELECT OWNER, TABLE_NAME
  FROM DBA_TABLES
    WHERE TABLE_NAME = 'PRODUCT_T';
```

Result:

OWNER	TABLE_NAME
MPRESCOTT	PRODUCT_T

Every RDBMS contains a set of tables in which metadata of the sort described for Oracle 11g is contained. Microsoft SQL Server 2008 divides the system tables (or views) into different categories, based on the information needed:

- ***Catalog views,*** which return information that is used by the SQL Server database engine. All user-available catalog metadata is exposed through catalog views.
- ***Compatibility views,*** which are implementations of the system tables from earlier releases of SQL Server. These views expose the same metadata available in SQL Server 2000.
- ***Dynamic management views and functions,*** which return server state information that can be used to monitor the health of a server instance, diagnose problems, and tune performance. There are two types of dynamic management views and functions:
  - ***Server-scoped dynamic management views and functions,*** which require VIEW SERVER STATE permission on the server.
  - ***Database-scoped dynamic management views and functions,*** which require VIEW DATABASE STATE permission on the database.
- ***Information schema views,*** which provide an internal system table–independent view of the SQL Server metadata. The information schema views included in SQL Server comply with the ISO standard definition for the INFORMATION_SCHEMA.
- ***Replication views,*** which contain information that is used by data replication in Microsoft SQL Server.

SQL Server metadata tables begin with sys, just as Oracle tables begin with DBA, USER, or ALL.

Here are a few of the Microsoft SQL Server 2008 catalog views:

View	Description
sys.columns	Table and column specifications
sys.computed_columns	Specifications about computed columns
sys.foreign_key_columns	Details about columns in foreign key constraints
sys.indexes	Table index information
sys.objects	Database objects listing
sys.tables	Tables and their column names
sys.synonyms	Names of objects and their synonyms

These metadata views can be queried just like a view of base table data. For example, the following query displays specific information about objects in a SQL Server database that have been modified in the past 10 days:

```
SELECT name as object_name, SCHEMA_NAME (schema_id) AS
   schema_name, type_desc, create_date, modify_date
FROM sys.objects
WHERE modify_date > GETDATE() – 10
ORDER BY modify_date;
```

You will want to investigate the system views and metadata commands available with the RDBMS you are using. They can be life savers when you need critical information to solve a homework assignment or to work exam exercises. (Is this enough motivation?)

SQL:200n ENHANCEMENTS AND EXTENSIONS TO SQL

Chapter 6 and this chapter have demonstrated the power and simplicity of SQL. However, readers with a strong interest in business analysis may have wondered about the limited set of statistical functions available. Programmers familiar with other languages may have wondered how variables will be defined, flow control established, or **user-defined data types (UDTs)** created. And, as programming becomes more object oriented, how is SQL going to adjust? SQL:1999 extended SQL by providing more programming capabilities. SQL:200n has standardized additional statistical functions. With time, the SQL standard will be modified to encompass object-oriented concepts. Other notable additions in SQL:200n include three new data types and a new part, SQL/XML. The first two areas, additional statistical functions within the WINDOW clause, and the new data types, are discussed here. SQL/XML is discussed briefly in Chapter 8.

User-defined data type (UDT)
A data type that a user can define by making it a subclass of a standard type or creating a type that behaves as an object. UDTs may also have defined functions and methods.

Analytical and OLAP Functions

SQL:200n added a set of analytical functions, referred to as OLAP (online analytical processing) functions, as SQL language extensions. Most of the functions have already been implemented in Oracle, DB2, Microsoft SQL Server, and Teradata. Including these functions in the SQL standard addresses the needs for analytical capabilities within the database engine. Linear regressions, correlations, and moving averages can now be calculated without moving the data outside the database. As SQL:200n is implemented, vendor implementations will adhere strictly to the standard and become more similar. We discuss OLAP further in Chapter 9, as part of the discussion of data warehousing.

Table 7-1 lists a few of the newly standardized functions. Both statistical and numeric functions are included. Functions such as ROW_NUMBER and RANK will allow the developer to work much more flexibly with an ordered result. For database marketing or customer relationship management applications, the ability to consider

TABLE 7-1 Some Built-in Functions Added in SQL:200n

Function	Description
CEILING	Computes the least integer greater than or equal to its argument—for example, CEIL(100) or CEILING(100).
FLOOR	Computes the greatest integer less than or equal to its argument—for example, FLOOR(25).
SQRT	Computes the square root of its argument—for example, SQRT(36).
RANK	Computes the ordinal rank of a row within its window. Implies that if duplicates exist, there will be gaps in the ranks assigned. The rank of the row is defined as 1 plus the number of rows preceding the row that are not peers of the row being ranked.
DENSE_RANK	Computes the ordinal rank of a row within its window. Implies that if duplicates exist, there will be no gaps in the ranks assigned. The rank of the row is the number of distinct rows preceding the row and itself.
ROLLUP	Works with GROUP BY to compute aggregate values for each level of the hierarchy specified by the group by columns, (The hierarchy is assumed to be left to right in the list of GROUP BY columns.)
CUBE	Works with GROUP BY to create a subtotal of all possible columns for the aggregate specified.
SAMPLE	Reduces the number of rows by returning one or more random samples (with or without replacement). (This function is not ANSI SQL-2003 compliant but is available with many RDBMSs.)
OVER or WINDOW	Creates partitions of data, based on values of one or more columns over which other analytical functions (e.g., RANK) can be computed.

only the top *n* rows or to subdivide the result into groupings by percentile is a welcome addition. Users can expect to achieve more efficient processing, too, as the functions are brought into the database engine and optimized. Once they are standardized, application vendors can depend on them, including their use in their applications and avoiding the need to create their own functions outside of the database.

SQL:1999 was amended to include an additional clause, the WINDOW clause. The WINDOW clause improves SQL's numeric analysis capabilities. It allows a query to specify that an action is to be performed over a set of rows (the window). This clause consists of a list of window definitions, each of which defines a name and specification for the window. Specifications include partitioning, ordering, and aggregation grouping.

Here is a sample query from the paper that proposed the amendment (Zemke et al., 1999, p. 4):

```
SELECT SH.Territory, SH.Month, SH.Sales,
AVG (SH.Sales) OVER W1 AS MovingAverage
    FROM SalesHistory AS SH
      WINDOW W1 AS (PARTITION BY (SH.Territory)
        ORDER BY (SH.Month ASC)
        ROWS 2 PRECEDING);
```

The window name is W1, and it is defined in the WINDOW clause that follows the FROM clause. The PARTITION clause partitions the rows in SalesHistory by Territory. Within each territory partition, the rows will be ordered in ascending order, by month. Finally, an aggregation group is defined as the current row and the two preceding rows of the partition, following the order imposed by the ORDER BY clause. Thus, a moving average of the sales for each territory will be returned as MovingAverage. Although proposed, MOVING_AVERAGE was not included in SQL:1999 or SQL:200n; it has been implemented by many RDBMS vendors, especially those supporting data warehousing and business intelligence. Though using SQL is not the preferred way to perform numeric analyses on data sets, inclusion of the WINDOW clause has made many OLAP analyses easier. Several new WINDOW functions were approved in SQL:200n. Of these new window functions, RANK and DENSE_RANK are included in Table 7-1. Previously included aggregate functions, such as AVG, SUM, MAX, and MIN, can also be used in the WINDOW clause.

New Data Types

SQL:200n includes three new data types and removed two traditional data types. The data types that were removed are BIT and BIT VARYING. Eisenberg et al. (2004) indicate that BIT and BIT VARYING were removed because they had not been widely supported by RDBMS products and were not expected to be supported.

The three new data types are BIGINT, MULTISET, and XML. BIGINT is an exact numeric type of scale 0, meaning it is an integer. The precision of BIGINT is greater than that of either INT or SMALLINT, but its exact definition is implementation specific. However, BIGINT, INT, and SMALLINT must have the same radix, or base system. All operations that can be performed using INT and SMALLINT can be performed using BIGINT, too.

MULTISET is a new collection data type. The previous collection data type is ARRAY, a noncore SQL data type. MULTISET differs from ARRAY because it can contain duplicates. This also distinguishes a table defined as MULTISET data from a relation, which is a set and cannot contain duplicates. MULTISET is unordered, and all elements are of the same element type. The elements can be any other supported data type. INTEGER MULTISET, for example, would define a multiset where all the elements are INTEGER data type. The values in a multiset may be created through INSERT or through a SELECT statement. An example of the INSERT approach would be MULTISET (2,3,5,7) and of the SELECT approach MULTISET (SELECT ProductDescription FROM Product_T WHERE ProductStandardPrice > 200;. MULTISET) reflects the real-world circumstance that some relations may contain duplicates that are acceptable when a subset is extracted from a table.

Other Enhancements

In addition to the enhancements to windowed tables described previously, the CREATE TABLE command has been enhanced by the expansion of CREATE TABLE LIKE options. CREATE TABLE LIKE allows one to create a new table that is similar to an existing table, but in SQL:1999 information such as default values, expressions used to generate a calculated column, and so forth, could not be copied to the new table. Now a general syntax of CREATE TABLE LIKE . . . INCLUDING has been approved. INCLUDING COLUMN DEFAULTS, for example, will pick up any default values defined in the original CREATE TABLE command and transfer it to the new table by using CREATE TABLE LIKE . . . INCLUDING. It should be noted that this command creates a table that seems similar to a materialized view. However, tables created using CREATE TABLE LIKE are independent of the table that was copied. Once the table is populated, it will not be automatically updated if the original table is updated.

An additional approach to updating a table can now be taken by using the new SQL:200n MERGE command. In a transactional database, it is an everyday need to be able to add new orders, new customers, new inventory, and so forth, to existing order, customer, and inventory tables. If changes that require updating information about customers and adding new customers are stored in a transaction table, to be added to the base customer table at the end of the business day, adding a new customer used to require an INSERT command, and changing information about an existing customer used to require an UPDATE command. The MERGE command allows both actions to be accomplished using only one query. Consider the following example from Pine Valley Furniture Company:


```
MERGE INTO Customer_T as Cust
    USING (SELECT CustomerID, CustomerName, CustomerAddress,
    CustomerCity, CustomerState, CustomerPostalCode
      FROM CustTrans_T)
      AS CT
    ON (Cust.CustomerID = CT.CustomerID)
WHEN MATCHED THEN UPDATE
  SET Cust.CustomerName = CT.CustomerName,
    Cust.CustomerAddress = CT.CustomerAddress,
    Cust.CustomerCity = CT.CustomerCity,
    Cust.CustomerState = CT.CustomerState,
    Cust.CustomerPostalCode = CT.CustomerPostalCode
WHEN NOT MATCHED THEN INSERT
  (CustomerID, CustomerName, CustomerAddress, CustomerCity,
  CustomerState, CustomerPostalCode)
    VALUES (CT.CustomerID, CT.CustomerName, CT.CustomerAddress,
    CT.CustomerCity, CT.CustomerState, CT.CustomerPostalCode);
```

Programming Extensions

SQL-92 and earlier standards developed the capabilities of SQL as a data retrieval and manipulation language, and not as an application language. As a result, SQL has been used in conjunction with computationally complete languages such as C, .NET, and Java to create business application programs, procedures, or functions. SQL:1999, however, extended SQL by adding programmatic capabilities in core SQL, SQL/PSM, and SQL/OLB. These capabilities have been carried forward and included in SQL:200n.

The extensions that make SQL computationally complete include flow control capabilities, such as IF-THEN, FOR, WHILE statements, and loops, which are contained in a package of extensions to the essential SQL specifications. This package, called **Persistent Stored Modules (SQL/PSM)**, is so named because the capabilities to create and drop program modules are stored in it. *Persistent* means that a module of code will be stored until dropped, thus making it available for execution across user sessions, just as the base tables are retained until they are explicitly dropped. Each module is stored in a schema as a schema object. A schema does not have to have any program modules, or it may have multiple modules.

Persistent Stored Modules (SQL/PSM)
Extensions defined in SQL:1999 that include the capability to create and drop modules of code stored in the database schema across user sessions.

Each module must have a name, an authorization ID, an association with a particular schema, an indication of the character set to be used, and any temporary table declarations that will be needed when the module executes. Every module must contain one or more SQL procedures—named programs that each execute one SQL statement when called. Each procedure must also include an SQLSTATE declaration that acts as a status parameter and indicates whether an SQL statement has been successfully executed.

SQL/PSM can be used to create applications or to incorporate procedures and functions using SQL data types directly. Using SQL/PSM introduces procedurality to SQL, because statements are processed sequentially. Remember that SQL by itself is a nonprocedural language and that no statement execution sequence is implied. SQL/PSM includes several SQL control statements:

Statement	Description
CASE	Executes different sets of SQL sequences, according to a comparison of values or the value of a WHEN clause, using either search conditions or value expressions. The logic is similar to that of an SQL CASE expression, but it ends with END CASE rather than END and has no equivalent to the ELSE NULL clause.
IF	If a predicate is TRUE, executes an SQL statement. The statement ends with an ENDIF and contains ELSE and ELSEIF statements to manage flow control for different conditions.
LOOP	Causes a statement to be executed repeatedly until a condition exists that results in an exit.
LEAVE	Sets a condition that results in exiting a loop.
FOR	Executes once for each row of a result set.
WHILE	Executes as long as a particular condition exists. Incorporates logic that functions as a LEAVE statement.
REPEAT	Similar to the WHILE statement, but tests the condition after execution of the SQL statement.
ITERATE	Restarts a loop.

SQL/PSM brings the promise of addressing several widely noted deficiencies of essential SQL. It is still too soon to know if programmers are going to embrace SQL/PSM or continue to use host languages, invoking SQL through embedded SQL or via call-level interface (CLI). The standard makes it possible to do the following:

- Create procedures and functions within SQL, thus making it possible to accept input and output parameters and to return a value directly
- Detect and handle errors within SQL rather than having to handle errors through another language
- Use the DECLARE statement to create variables that stay in scope throughout the procedure, method, or function in which they are contained
- Pass groups of SQL statements rather than individual statements, thus improving performance
- Handle the impedance-mismatch problem, where SQL processes sets of data while procedural languages process single rows of data within modules

SQL/PSM has not yet been widely implemented, and therefore we have not included extensive syntax examples in this chapter. Oracle's PL/SQL and Microsoft SQL Server's T-SQL bear some resemblance to the new standard, with its modules of code and BEGIN . . . END, LOOP, and WHILE statements. Although SQL/PSM is not yet widely popular, this situation could change quickly.

TRIGGERS AND ROUTINES

Prior to the issuance of SQL:1999, no support for user-defined functions or procedures was included in the SQL standards. Commercial products, recognizing the need for such capabilities, have provided them for some time, and we expect to see their syntax

change over time to be in line with the SQL:1999 requirements, just as we expect to see inclusion of SQL/PSM standards.

Triggers and routines are very powerful database objects because they are stored in the database and controlled by the DBMS. Thus, the code required to create them is stored in only one location and is administered centrally. As with table and column constraints, this promotes stronger data integrity and consistency of use within the database; it can be useful in data auditing and security to create logs of information about data updates. Not only can triggers be used to prevent unauthorized changes to the database, they can also be used to evaluate changes and take actions based on the nature of the changes. Because triggers are stored only once, code maintenance is also simplified (Mullins, 1995). Also, because they can contain complex SQL code, they are more powerful than table and column constraints; however, constraints are usually more efficient and should be used instead of the equivalent triggers, if possible. A significant advantage of a trigger over a constraint to accomplish the same control is that the processing logic of a trigger can produce a customized user message about the occurrence of a special event, whereas a constraint will produce a standardized, DBMS error message, which often is not very clear about the specific event that occurred.

Trigger
A named set of SQL statements that are considered (triggered) when a data modification (i.e., INSERT, UPDATE, DELETE) occurs or if certain data definitions are encountered. If a condition stated within a trigger is met, then a prescribed action is taken.

Both triggers and routines consist of blocks of procedural code. Routines are stored blocks of code that must be called to operate (see Figure 7-11). They do not run automatically. In contrast, trigger code is stored in the database and runs automatically whenever the triggering event, such as an UPDATE, occurs. Triggers are a special type of stored procedure and may run in response to either DML or DDL commands. Trigger syntax and functionality vary from RDBMS to RDBMS. A trigger written to work with an Oracle database will need to be rewritten if the database is ported to Microsoft SQL Server and vice versa. For example, Oracle triggers can be written to fire once per INSERT, UPDATE, or DELETE command or to fire once per row affected by the command. Microsoft SQL Server triggers can fire only once per DML command, not once per row.

Triggers

Because triggers are stored and executed in the database, they execute against all applications that access the database. Triggers can also cascade, causing other triggers to fire. Thus, a single request from a client can result in a series of integrity or logic checks being performed on the server without causing extensive network traffic between client

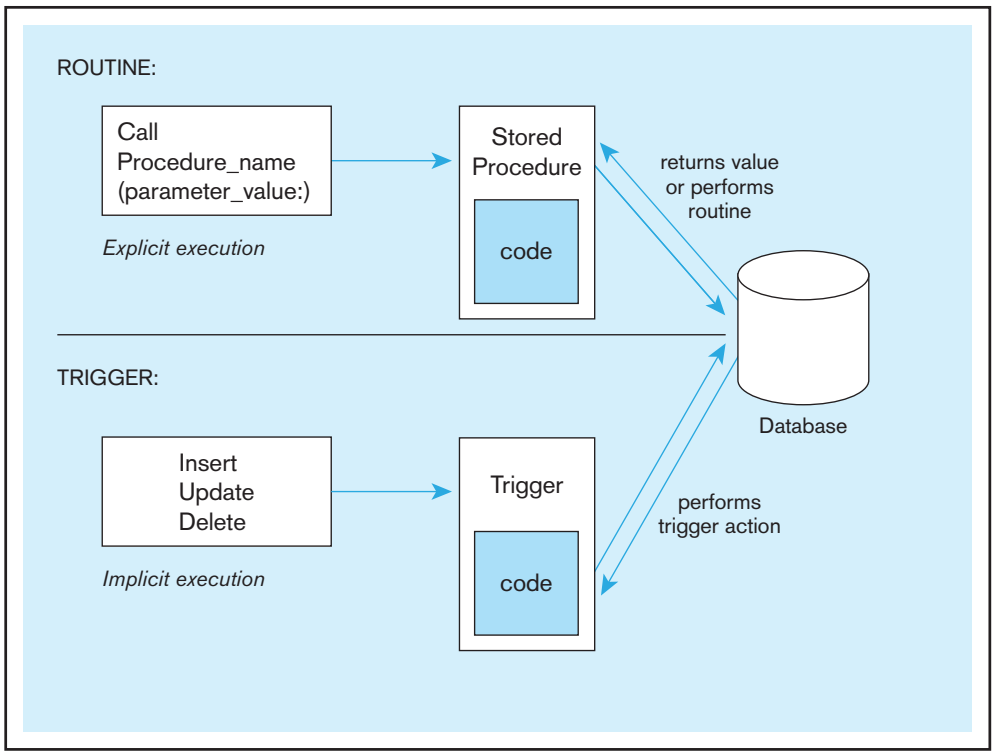


FIGURE 7-11 Triggers contrasted with stored procedures

Source: Based on Mullins (1995).

FIGURE 7-12 Simplified trigger syntax in SQL:200n

```
CREATETRIGGER trigger_name
        {BEFORE| AFTER | INSTEAD OF} {INSERT | DELETE | UPDATE} ON
        table_name
        [FOR EACH {ROW | STATEMENT}] [WHEN ( search condition)]
        <triggered SQL statement here>;
```

and server. Triggers can be used to ensure referential integrity, enforce business rules, create audit trails, replicate tables, or activate a procedure (Rennhackkamp, 1996).

Constraints can be thought of as a special case of triggers. They also are applied (triggered) automatically as a result of data modification commands, but their precise syntax is determined by the DBMS and does not have the flexibility of a trigger.

Triggers are used when you need to perform, under specified conditions, a certain action as the result of some database event (e.g., the execution of a DML statement such as INSERT, UPDATE, or DELETE or the DDL statement ALTER TABLE). Thus, a trigger has three parts—the *event*, the *condition*, and the *action*—and these parts are reflected in the coding structure for triggers. (See Figure 7-12 for a simplified trigger syntax.) Consider the following example from Pine Valley Furniture Company: Perhaps the manager in charge of maintaining inventory needs to know (the action of being informed) when an inventory item's standard price is updated in the Product_T table (the event). After creating a new table, PriceUpdates_T, a trigger can be written that enters each product when it is updated, the date that the change was made, and the new standard price that was entered. The trigger is named StandardPriceUpdate, and the code for this trigger follows:

```
CREATE TRIGGER StandardPriceUpdate
AFTER UPDATE OF ProductStandardPrice ON Product_T
FOR EACH ROW
INSERT INTO PriceUpdates_T VALUES (ProductDescription, SYSDATE,
ProductStandardPrice);
```

In this trigger, the *event* is an update of ProductStandardPrice, the *condition* is FOR EACH ROW (i.e., not just certain rows), and the *action after the event* is to insert the specified values in the PriceUpdates_T table, which stores a log of when (SYSDATE) the change occurred and important information about changes made to the ProductStandardPrice of any row in the table. More complicated conditions are possible, such as taking the action for rows where the new ProductStandardPrice meets some limit or the product is associated with only a certain product line. It is important to remember that the procedure in the trigger is performed every time the event occurs; no user has to ask for the trigger to fire, nor can any user prevent it from firing. Because the trigger is associated with the Product_T table, the trigger will fire no matter the source (application) causing the event; thus, an interactive UPDATE command or an UPDATE command in an application program or stored procedure against the ProductStandardPrice in the Product_T table will cause the trigger to execute. In contrast, a routine (or stored procedure) executes only when a user or program asks for it to run.

Triggers may occur either *before*, *after*, or *instead of* the statement that aroused the trigger is executed. An "instead of" trigger is not the same as a before trigger but executes instead of the intended transaction, which does not occur if the "instead of" trigger fires. DML triggers may occur on INSERT, UPDATE, or DELETE commands. And they may fire each time a *row* is affected, or they may fire only once per *statement*, regardless of the number of rows affected. In the case just shown, the trigger should insert the new standard price information into PriceUpdate_T after Product_T has been updated.

DDL triggers are useful in database administration and may be used to regulate database operations and perform auditing functions. They fire in response to DDL events such as CREATE, ALTER, DROP, GRANT, DENY, and REVOKE. The sample trigger below, taken from SQL Server 2008 Books Online [**http://msdn2.microsoft.com/en-us/library/ms175941**], demonstrates how a trigger can be used to prevent the unintentional modification or drop of a table in the database:

```
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
    PRINT 'You must disable Trigger "safety" to drop or alter tables!'
    ROLLBACK;
```

A developer who wishes to include triggers should be careful. Because triggers fire automatically, unless a trigger includes a message to the user, the user will be unaware that the trigger has fired. Also, triggers can cascade and cause other triggers to fire. For example, a BEFORE UPDATE trigger could require that a row be inserted in another table. If that table has a BEFORE INSERT trigger, it will also fire, possibly with unintended results. It is even possible to create an endless loop of triggers! So, while triggers have many possibilities, including enforcement of complex business rules, creation of sophisticated auditing logs, and enforcement of elaborate security authorizations, they should be included with care.

Triggers can be written that provide little notification when they are triggered. A user who has access to the database but not the authority to change access permissions might insert the following trigger, also taken from SQL Server 2008 Books Online [**http://msdn2.microsoft.com/en-us/library/ms191134**]:

```
CREATE TRIGGER DDL_trigJohnDoe
ON DATABASE
FOR ALTER_TABLE
AS
GRANT CONTROL SERVER TO JohnDoe;
```

When an administrator with appropriate permissions issues any ALTER _TABLE command, the trigger DDL_trigJohnDoe will fire without notifying the administrator, and it will grant CONTROL SERVER permissions to John Doe.

Routines

In contrast to triggers, which are automatically run when a specified event occurs, routines must be explicitly called, just as the MIN built-in function is called. SQL-invoked routines can be either procedures or functions. The terms *procedure* and *function* are used in the same manner as they are in other programming languages. A **function** returns one value and has only input parameters. You have already seen the many built-in functions included in SQL, including the newest functions listed in Table 7-1. A **procedure** may have input parameters, output parameters, and parameters that are both input and output parameters. You may declare and name a unit of procedural code using proprietary code of the RDBMS product being used or invoke (via a CALL to an external procedure) a host-language library routine. SQL products had developed their own versions of routines prior to the issuance of SQL:1999, so be sure to become familiar with the syntax and capabilities of any product you use. Some of these proprietary languages, such as Microsoft SQL Server's Transact-SQL and Oracle's PL/SQL, are in wide use and will continue to be available. To give you an idea of how much stored procedure syntax has varied across products, Table 7-2 examines the CREATE PROCEDURE syntax used by three RDBMS vendors; this is the syntax for a procedure stored with the database. This table comes from **www.tdan.com/i023fe03.htm** by Peter Gulutzan (accessed June 6, 2007, but no longer accessible).

Function
A stored subroutine that returns one value and has only input parameters.

Procedure
A collection of procedural and SQL statements that are assigned a unique name within the schema and stored in the database.

The following are some of the advantages of SQL-invoked routines:

- ***Flexibility*** Routines may be used in more situations than constraints or triggers, which are limited to data-modification circumstances. Just as triggers have more code options than constraints, routines have more code options than triggers.
- ***Efficiency*** Routines can be carefully crafted and optimized to run more quickly than slower, generic SQL statements.

TABLE 7-2 Comparison of Vendor Syntax Differences in Stored Procedures

The vendors' syntaxes differ in stored procedures more than in ordinary SQL. For an illustration, here is a chart that shows what CREATE PROCEDURE looks like in three dialects. We use one line for each significant part, so you can compare dialects by reading across the line.

SQL:1999/IBM	MICROSOFT/SYBASE	ORACLE
CREATE PROCEDURE	CREATE PROCEDURE	CREATE PROCEDURE
Sp_proc1	Sp_proc1	Sp_proc1
(param1 INT)	@param1 INT	(param1 IN OUT INT)
MODIFIES SQL DATA BEGIN DECLARE num1 INT;	AS DECLARE @num1 INT	AS num1 INT; BEGIN
IF param1 <> 0	IF @param1 <> 0	IF param1 <> 0
THEN SET param1 = 1;	SELECT @param1 = 1;	THEN param1 :=1;
END IF		END IF;
UPDATE Table1 SET column1 = param1;	UPDATE Table1 SET column1 = @param1	UPDATE Table1 SET column1 = param1;
END		END

Source: Data from *SQL Performance Tuning* (Gulutzan and Pelzer, Addison-Wesley, 2002). Viewed at **www.tdan.com/i023fe03.htm**, June 6, 2007 (no longer available from this site).

- ***Sharability*** Routines may be cached on the server and made available to all users so that they do not have to be rewritten.
- ***Applicability*** Routines are stored as part of the database and may apply to the entire database rather than be limited to one application. This advantage is a corollary to sharability.

The SQL:200n syntax for procedure and function creation is shown in Figure 7-13. As you can see, the syntax is complicated, and we will not go into the details about each clause here. However, a simple procedure follows, to give you an idea of how the code works.

A procedure is a collection of procedural and SQL statements that are assigned a unique name within the schema and stored in the database. When it is needed to run the procedure, it is called by name. When it is called, all of the statements in the procedure will be executed. This characteristic of procedures helps to reduce network traffic, because all of the statements are transmitted at one time, rather than sent individually. A procedure can access database contents and may have local variables. When the procedure accesses database contents, the procedure will generate an error message if the user/program calling the procedure does not have the necessary rights to access the part of the database used by the procedure.

FIGURE 7-13 Syntax for creating a routine, SQL:200n

```
{CREATE PROCEDURE | CREATE FUNCTION} routine_name
([parameter [{,parameter} . . .]])
[RETURNS data_type result_cast]    /* for functions only */
[LANGUAGE {ADA|C|COBOL|FORTRAN|MUMPS|PASCAL|PLI|SQL}]
[PARAMETER STYLE {SQL|GENERAL}]
[SPECIFIC specific_name]
[DETERMINISTIC|NOT DETERMINISTIC]
[NO SQL|CONTAINS SQL|READS SQL DATA|MODIFIES SQL DATA]
[RETURNS NULL ON NULL INPUT|CALLED ON NULL INPUT]
[DYNAMIC RESULT SETS unsigned_integer]      /* for procedures only */
[STATIC DISPATCH]                           /* for functions only */
[NEW SAVEPOINT LEVEL | OLD SAVEPOINT LEVEL]
routine_body
```

To build a simple procedure that will set a sale price, the existing Product_T table in Pine Valley Furniture company is altered by adding a new column, SalePrice, that will hold the sale price for the products:

```
ALTER TABLE Product_T
ADD (SalePrice DECIMAL (6,2));
```

Result:

```
Table altered.
```

This simple procedure will execute two SQL statements, and there are no input or output parameters; if present, parameters are listed and given SQL data types in a parenthetical clause after the name of the procedure, similar to the columns in a CREATE TABLE command. The procedure scans all rows of the Product_T table. Products with a ProductStandardPrice of $400 or higher are discounted 10 percent, and products with a ProductStandardPrice of less than $400 are discounted 15 percent. As with other database objects, there are SQL commands to create, alter, replace, drop, and show the code for procedures. The following is an Oracle code module that will create and store the procedure named ProductLineSale:

```
CREATE OR REPLACE PROCEDURE ProductLineSale
  AS BEGIN
    UPDATE Product_T
      SET SalePrice = .90 * ProductStandardPrice
      WHERE ProductStandardPrice > = 400;
    UPDATE Product_T
      SET SalePrice = .85 * ProductStandardPrice
      WHERE ProductStandardPrice < 400;
END;
```

Oracle returns the comment "Procedure created" if the syntax has been accepted.

To run the procedure in Oracle, use this command (which can be run interactively, as part of an application program, or as part of another stored procedure):

```
SQL > EXEC ProductLineSale
```

Oracle gives this response:

```
PL/SQL procedure successfully completed.
```

Now Product_T contains the following:

PRODUCTLINE	PRODUCTID	PRODUCTDESCRIPTION	PRODUCTFINISH	PRODUCTSTANDARDPRICE	SALEPRICE
10001	1	End Table	Cherry	175	148.75
20001	2	Coffee Table	Natural Ash	200	170
20001	3	Computer Desk	Natural Ash	375	318.75
30001	4	Entertainment Center	Natural Maple	650	585
10001	5	Writer's Desk	Cherry	325	276.25
20001	6	8-Drawer Dresser	White Ash	750	675
20001	7	Dining Table	Natural Ash	800	720
30001	8	Computer Desk	Walnut	250	212.5

We have emphasized numerous times that SQL is a set-oriented language, meaning that, in part, the result of an SQL command is a set of rows. You probably noticed in Figure 7-13 that procedures can be written to work with many different host languages, most of which are record-oriented languages, meaning they are designed to manipulate one record, or row, at a time. This difference is often called an *impedance mismatch* between SQL and the host language that uses SQL commands. When SQL calls an SQL procedure, as in the example above, this is not an issue, but when the procedure is called, for example, by a C program, it can be an issue. In the next section we consider embedding SQL in host languages and some of the additional capabilities needed to allow SQL to work seamlessly with languages not designed to communicate with programs written in other, set-oriented languages.

EMBEDDED SQL AND DYNAMIC SQL

We have been using the interactive, or direct, form of SQL. With interactive SQL, one SQL command is entered and executed at a time. Each command constitutes a logical unit of work, or a transaction. The commands necessary to maintain a valid database, such as ROLLBACK and COMMIT, are transparent to the user in most interactive SQL situations. SQL was originally created to handle database access alone and did not have flow control or the other structures necessary to create an application. SQL/PSM, introduced in SQL:1999, provides for the types of programmatic extensions needed to develop a database application.

Embedded SQL
Hard-coded SQL statements included in a program written in another language, such as C or Java.

Dynamic SQL
Specific SQL code generated on the fly while an application is processing.

Prior to SQL/PSM, two other forms of SQL were widely used in creating applications on both clients and servers; they are referred to as **embedded SQL** and **dynamic SQL**. SQL commands can be embedded in third-generation langagues (3GLs), such as Ada, and COBOL, as well as in C, PHP, .NET, and Java if the commands are placed at appropriate locations in a 3GL host program. Oracle also offers PL/SQL, or SQL Procedural Language, a proprietary language that extends SQL by adding some procedural language features such as variables, types, control structures (including IF-THEN-ELSE loops), functions, and procedures. PL/SQL blocks of code can also be embedded within 3GL programs.

Dynamic SQL derives the precise SQL statement at run time. Programmers write to an application programming interface (API) to achieve the interface between languages. Embedded SQL and dynamic SQL will continue to be used. Programmers are used to them, and in many cases they are still an easier approach than attempting to use SQL as an application language in addition to using it for database creation, administration, and querying.

There are several reasons to consider embedding SQL in a 3GL:

1. It is possible to create a more flexible, accessible interface for the user. Using interactive SQL requires a good understanding of both SQL and the database structure—understanding that a typical application user may not have. Although many RDBMSs come with form, report, and application generators (or such capabilities available as add-ons), developers frequently envision capabilities that are not easily accomplished with these tools but that can be easily accomplished using a 3GL. Large, complex programs that require access to a relational database may best be programmed in a 3GL with embedded SQL calls to an SQL database.
2. It may be possible to improve performance by using embedded SQL. Using interactive SQL requires that each query be converted to executable machine code each time the query is processed. Or, the query optimizer, which runs automatically in a direct SQL situation, may not successfully optimize the query, causing it to run slowly. With embedded SQL, the developer has more control over database access and may be able to create significant performance improvements. Knowing when to rely on the SQL translator and optimizer and when to control it through the program depends on the nature of the problem, and making this trade-off is best accomplished through experience and testing.

3. Database security may be improved by using embedded SQL. Restricted access can be achieved by a DBA through the GRANT and REVOKE permissions in SQL and through the use of views. These same restrictions can also be invoked in an embedded SQL application, thus providing another layer of protection. Complex data integrity checks also may be more easily accomplished, including cross-field consistency checks.

A program that uses embedded SQL will consist of the host program written in a 3GL such as C or COBOL, and there will also be sections of SQL code sprinkled throughout. Each section of SQL code will begin with EXEC SQL, keywords used to indicate an embedded SQL command that will be converted to the host source code when run through the precompiler. You will need a separate precompiler for each host language that you plan to use. Be sure to determine that the 3GL compiler is compatible with your RDBMS's precompiler for each language.

When the precompiler encounters an EXEC SQL statement, it will translate that SQL command into the host program language. Some, but not all, precompilers will check for correct SQL syntax and generate any required error messages at this point. Others will not generate an error message until the SQL statement actually attempts to execute. Some products' precompilers (DB2, SQL/DS, Ingres) create a separate file of SQL statements that is then processed by a separate utility called a binder, which determines that the referenced objects exist, that the user possesses sufficient privileges to run the statement, and the processing approach that will be used. Other products (Oracle, Informix) interpret the statements at run time rather than compiling them. In either case, the resulting program will contain calls to DBMS routines, and the link/editor programs will link these routines into the program.

Here is a simple example, using C as the host language, that will give you an idea of what embedded SQL looks like in a program. This example uses a prepared SQL statement named GETCUST, which will be compiled and stored as executable code in the database. CustID is the primary key of the customer table. GETCUST, the prepared SQL statement, returns customer information (cname, caddress, city, state, postcode) for an order number. A placeholder is used for the order information, which is an input parameter. Customer information is output from the SQL query and stored into host variables using the *into* clause. This example assumes that only one row is returned from the query, what is often called a singleton SELECT. (We'll discuss below how to handle the situation in which it is possible to return more than one row.)

```
    exec sql prepare getcust from
    "select cname, c_address, city, state, postcode
    from customer_t, order_t
    where customer_t.custid = order_t.custid and orderid = ?";
    .
    .
    ./* code to get proper value in theOrder */
exec sql execute getcust into :cname, :caddress, :city, :state,
:postcode using theOrder;
    .
    .
    .
```

If a prepared statement returns multiple rows, it is necessary to write a program loop using cursors to return a row at a time to be stored. A cursor is a data structure, internal to the programming environment, that points to a result table row (similarly to how a display screen cursor points to where data would be inserted in a form if you began entering data). Cursors help to eliminate the impedance mismatch between SQL's set-at-a-time processing and procedural languages' record-at-a-time processing. Record-at-a-time languages have to be able to move cursor values forward and backward in the set (FETCH NEXT or FETCH PRIOR), to find the first or last row in a result

set (FETCH FIRST and FETCH LAST), to move the cursor to a specific row or one relative to the current position (FETCH ABSOLUTE or FETCH RELATIVE), and to know the number of rows to process and when the end of the result set is reached, which often triggers the end of a programming loop (FOR . . . END FOR). There are different types of cursors, and the number of types and how they are each handled varies by RDBMS. Thus, this topic is beyond the scope of this book, although you are now aware of this important aspect of embedded SQL.

Dynamic SQL is used to generate appropriate SQL code on the fly while an application is processing. Most programmers write to an API, such as ODBC, which can then be passed through to any ODBC-compliant database. Dynamic SQL is central to most Internet applications. The developer is able to create a more flexible application because the exact SQL query is determined at run time, including the number of parameters to be passed, which tables will be accessed, and so forth. Dynamic SQL is very useful when an SQL statement shell will be used repeatedly, with different parameter values being inserted each time it executes.

Embedded and dynamic SQL code is vulnerable to malicious modification. Any procedure that has or especially constructs SQL statements should be reviewed for such vulnerabilities. A common form of such an attack involves insertion of the malicious code into user input variables that are concatenated with SQL commands and then executed. Alternatively, malicious code can be included in text stored in the database. As long as the malicious code is syntactically correct, the SQL database engine will process it. Preventing and detecting such attacks can be complicated, and this is beyond the scope of this text. The reader is encouraged to do an Internet search on the topic of SQL injection for recommendations. At a minimum, user input should be carefully validated, strong typing of columns should be used to limit exposure, and input data can be filtered or modified so that special SQL characters (e.g., ;) or words (e.g., DELETE) are put in quotes so they cannot be executed.

Currently, the Open Database Connectivity (ODBC) standard is the most commonly used API. SQL:1999 includes the SQL Call Level Interface (SQL/CLI). Both are written in C, and both are based on the same earlier standard. Java Database Connectivity (JDBC) is an industry standard used for connecting from Java. It is not yet an ISO standard. No new functionality has been added in SQL:200n.

As SQL:200n becomes implemented more completely, the use of embedded and dynamic SQL will become more standardized because the standard creates a computationally complete SQL language for the first time. Because most vendors have created these capabilities independently, though, the next few years will be a period in which SQL:1999-compliant products will exist side by side with older, but entrenched, versions. The user will need to be aware of these possibilities and deal with them.

Summary

This chapter continues from Chapter 6, which introduced the SQL language. Equi-joins, natural joins, outer joins, and union joins have been considered. Equi-joins are based on equal values in the common columns of the tables that are being joined and will return all requested results including the values of the common columns from each table included in the join. Natural joins return all requested results, but values of the common columns are included only once. Outer joins return all the values in one of the tables included in the join, regardless of whether a match exists in the other table or not. Union joins return a table that includes all data from each table that was joined.

Nested subqueries, where multiple SELECT statements are nested within a single query, are useful for more complex query situations. A special form of the subquery, a correlated subquery, requires that a value be known from the outer query before the inner query can be processed. Other subqueries process the inner query, return a result to the next outer query, and then process that outer query.

Other advanced SQL topics include the use of embedded SQL and the use of triggers and routines. SQL can be included within the context of many third-generation languages including COBOL, C, Fortran, and Ada and more modern languages such as C, PHP, .NET, and Java. The use of embedded SQL allows for the development of more flexible interfaces, improved performance, and improved database security. User-defined functions that run automatically when records are inserted, updated, or deleted are called triggers. Procedures are user-defined

code modules that can be called to execute. OLTP and OLAP are used for operational transaction processing and data analysis respectively.

New analytical functions included in SQL:200n are shown. Extensions already included in SQL:1999 made SQL computationally complete and included flow control capabilities in a set of SQL specifications known as Persistent Stored Modules (SQL/PSM). SQL/PSM can be used to create applications or to incorporate procedures and functions using SQL data types directly. SQL-invoked routines, including triggers, functions, and procedures, were also included in SQL:1999. Users must realize that these capabilities have been included as vendor-specific extensions and will continue to exist for some time.

Dynamic SQL is an integral part of Web-enabling databases and will be demonstrated in more detail in Chapter 8. This chapter has presented some of the more complex capabilities of SQL and has created awareness of the extended and complex capabilities of SQL that must be mastered to build database application programs.

Chapter Review

Key Terms

Correlated subquery *303*
Dynamic SQL *326*
Embedded SQL *326*
Equi-join *291*
Function *323*
Join *290*
Natural join *292*
Outer join *293*
Persistent Stored Modules (SQL/PSM) *319*
Procedure *323*
Trigger *321*
User-defined data type (UDT) *317*

Review Questions

1. Define each of the following terms:
   a. dynamic SQL
   b. correlated subquery
   c. embedded SQL
   d. procedure
   e. join
   f. equi-join
   g. self join
   h. outer join
   i. function
   j. Persistent Stored Modules (SQL/PSM)

2. Match the following terms to the appropriate definition:

____ equi-join	a. undoes changes to a table
____ natural join	b. user-defined data type
____ outer join	c. SQL:1999 extension
____ trigger	d. returns all records of designated table
____ procedure	e. keeps redundant columns
____ embedded SQL	f. makes changes to a table permanent
____ UDT	g. process that includes SQL statements within a host language
____ COMMIT	h. process of making an application capable of generating specific SQL code on the fly
____ SQL/PSM	i. does not keep redundant columns
____ Dynamic SQL	j. set of SQL statements that execute under stated conditions
____ ROLLBACK	k. stored, named collection of procedural and SQL statements

3. When is an outer join used instead of a natural join?
4. Explain the processing order of a correlated subquery.
5. Explain the following statement regarding SQL: Any query that can be written using the subquery approach can also be written using the joining approach but not vice versa.
6. What is the purpose of the COMMIT command in SQL? How does commit relate to the notion of a business transaction (e.g., entering a customer order or issuing a customer invoice)?
7. Care must be exercised when writing triggers for a database. What are some of the problems that could be encountered?
8. Explain the structure of a module of code that defines a trigger.
9. Under what conditions can a UNION clause be used?
10. Discuss the differences between triggers and stored procedures.
11. Explain the purpose of SQL/PSM.
12. List four advantages of SQL-invoked routines.
13. When would you consider using embedded SQL? When would you use dynamic SQL?
14. When do you think that the CASE keyword in SQL would be useful?
15. Explain the use of derived tables.
16. Describe an example in which you would want to use a derived table.
17. What other Oracle object can be used in place of a derived table? Which approach do you think is better?
18. If two queries involved in a UNION operation contained columns that were data type incompatible, how would you recommend fixing this?
19. Can an outer join be easily implemented when joining more than two tables? Why or why not?
20. This chapter discusses the data dictionary views for Oracle 11g. Research another RDBMS, such as Microsoft SQL Server, and report on its data dictionary facility and how it compares with Oracle.

Problems and Exercises

Problems and Exercises 1 through 5 are based on the class schedule 3NF relations along with some sample data in Figure 7-14. For Problems and Exercises 1 through 5, draw a Venn or ER diagrams and mark it to show the data you expect your query use to produce the results.

1. Write SQL retrieval commands for each of the following queries:
   a. Display the course ID and course name for all courses with an ISM prefix.
   b. Display all courses for which Professor Berndt has been qualified.
   c. Display the class roster, including student name, for all students enrolled in section 2714 of ISM 4212.
2. Write an SQL query to answer the following question: Which instructors are qualified to teach ISM 3113?
3. Write an SQL query to answer the following question: Is any instructor qualified to teach ISM 3113 and not qualified to teach ISM 4930?
4. Write SQL queries to answer the following questions:
   a. How many students were enrolled in section 2714 during semester I-2008?
   b. How many students were enrolled in ISM 3113 during semester I-2008?
5. Write an SQL query to answer the following question: Which students were not enrolled in any courses during semester I-2008?

   Problems and Exercises 6 through 14 are based on Figure 7-15. This problem set continues from Chapter 6, Problems and Exercises 10 through 15, which were based on Figure 6-12.

6. Determine the relationships among the four entities in Figure 7-15. List primary keys for each entity and any foreign keys necessary to establish the relationships and maintain referential integrity. Pay particular attention to the data contained in TUTOR REPORTS when you set up its primary key.

STUDENT (StudentID, StudentName)

StudentID	StudentName
38214	Letersky
54907	Altvater
66324	Aiken
70542	Marra
...	

QUALIFIED (FacultyID, CourseID, DateQualified)

FacultyID	CourseID	DateQualified
2143	ISM 3112	9/1988
2143	ISM 3113	9/1988
3467	ISM 4212	9/1995
3467	ISM 4930	9/1996
4756	ISM 3113	9/1991
4756	ISM 3112	9/1991
...		

FACULTY (FacultyID, FacultyName)

FacultyID	FacultyName
2143	Birkin
3467	Berndt
4756	Collins
...	

SECTION (SectionNo, Semester, CourseID)

SectionNo	Semester	CourseID
2712	I-2008	ISM 3113
2713	I-2008	ISM 3113
2714	I-2008	ISM 4212
2715	I-2008	ISM 4930
...		

COURSE (CourseID, CourseName)

CourseID	CourseName
ISM 3113	Syst Analysis
ISM 3112	Syst Design
ISM 4212	Database
ISM 4930	Networking
...	

REGISTRATION (StudentID, SectionNo, Semester)

StudentID	SectionNo	Semester
38214	2714	I-2008
54907	2714	I-2008
54907	2715	I-2008
66324	2713	I-2008
...		

FIGURE 7-14 **Class scheduling relations (for Problems and Exercises 1–5)**

FIGURE 7-15 **Adult literacy program (for Problems and Exercises 6–14)**

TUTOR (TutorID, CertDate, Status)

TutorID	CertDate	Status
100	1/05/2008	Active
101	1/05/2008	Temp Stop
102	1/05/2008	Dropped
103	5/22/2008	Active
104	5/22/2008	Active
105	5/22/2008	Temp Stop
106	5/22/2008	Active

STUDENT (StudentID, Read)

StudentID	Read
3000	2.3
3001	5.6
3002	1.3
3003	3.3
3004	2.7
3005	4.8
3006	7.8
3007	1.5

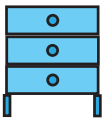
MATCH HISTORY (MatchID, TutorID, StudentID, StartDate, EndDate)

MatchID	TutorID	StudentID	StartDate	EndDate
1	100	3000	1/10/2008	
2	101	3001	1/15/2008	5/15/2008
3	102	3002	2/10/2008	3/01/2008
4	106	3003	5/28/2008	
5	103	3004	6/01/2008	6/15/2008
6	104	3005	6/01/2008	6/28/2008
7	104	3006	6/01/2008	

TUTOR REPORT (MatchID, Month, Hours, Lessons)

MatchID	Month	Hours	Lessons
1	6/08	8	4
4	6/08	8	6
5	6/08	4	4
4	7/08	10	5
1	7/08	4	2

7. Write the SQL command to add MATH SCORE to the STUDENT table.
8. Write the SQL command to add SUBJECT to TUTOR. The only values allowed for SUBJECT will be Reading, Math, and ESL.
9. What do you need to do if a tutor signs up and wants to tutor in both reading and math? Draw the new ERD and write any SQL statements that would be needed to handle this development.
10. Write the SQL command to find any tutors who have not submitted a report for July.
11. Where do you think student and tutor information such as name, address, phone, and e-mail should be kept? Write the necessary SQL commands to capture this information.
12. List all active students in June by name. (Make up names and other data if you are actually building a prototype database.) Include the number of hours students received tutoring and how many lessons they completed.
13. Which tutors, by name, are available to tutor? Write the SQL command.
14. Which tutor needs to be reminded to turn in reports? Write the SQL command.



PINE VALLEY FURNITURE

Problems and Exercises 15 through 44 are based on the entire ("big" version) Pine Valley Furniture Company database. Note: Depending on what DBMS you are using, some field names may have changed to avoid conflicting with reserved words for the DBMS. When you first use the DBMS, check the table definitions to see what the field names are for the DBMS you are using. See the Preface and inside covers of this book for instructions on where to find this database, including on **www.teradatastudentnetwork.com**.

15. Write an SQL command that will find any customers who have not placed orders.
16. List the names and number of employees supervised (label this value HeadCount) for each supervisor who supervises more than two employees.
17. List the name of each employee, his or her birth date, the name of his or her manager, and the manager's birth date for those employees who were born before their manager was born; label the manager's data Manager and ManagerBirth.
18. Write an SQL command to display the order number, customer number, order date, and items ordered for order number 1.
19. Write an SQL command to display each item ordered for order number 1, its standard price, and the total price for each item ordered.
20. Write an SQL command to total the cost of order number 1.
21. Calculate the total raw material cost (label TotCost) for each product compared to its standard product price. Display product ID, product description, standard price, and the total cost in the result.
22. For every order that has been received, display the order ID, the total dollar amount owed on that order (you'll have to calculate this total from attributes in one or more tables; label this result TotalDue), and the amount received in

payments on that order (assume that there is only one payment made on each order). To make this query a little simpler, you don't have to include those orders for which no payment has yet been received. List the results in decreasing order of the difference between total due and amount paid.

23. Write an SQL query to list each salesperson who has sold computer desks and the number of units sold by each salesperson.

24. List, in alphabetical order, the names of all employees (managers) who are now managing people with skill ID BS12; list each manager's name only once, even if that manager manages several people with this skill.

25. Display the salesperson name, product finish, and total quantity sold (label as TotSales) for each finish by each salesperson.

26. Write a query to list the number of products produced in each work center (label as TotalProducts). If a work center does not produce any products, display the result with a total of 0.

27. The production manager at PVFC is concerned about support for purchased parts in products owned by customers. A simple analysis he wants done is to determine for each customer how many vendors are in the same state as that customer. Develop a list of *all* the PVFC customers by name with the number of vendors in the same state as that customer. (Label this computed result NumVendors.)

28. Display the order IDs for customers who have not made any payment, yet, on that order. Use the set command UNION, INTERSECT, or MINUS in your query.

29. Display the names of the states in which customers reside but for which there is no salesperson residing in that state. There are several ways to write this query. Try to write it without any WHERE clause. Write this query two ways, using the set command UNION, INTERSECT, or MINUS and not using any of these commands. Which was the most natural approach for you, and why?

30. Write an SQL query to produce a list of all the products (i.e., product description) and the number of times each product has been ordered.

31. Display the customer ID, name, and order ID for all customer orders. For those customers who do not have any orders, include them in the display once.

32. Display the EmployeeID and EmployeeName for those employees who do not possess the skill Router. Display the results in order by EmployeeName.

33. Display the name of customer 16 and the names of all the customers that are in the same zip code as customer 16. (Be sure this query will work for any customer.)

34. Rewrite your answer to Problem and Exercise 33 for each customer, not just customer 16.

35. Display the customer ID, name, and order ID for all customer orders. For those customers who do not have any orders, include them in the display once by showing order ID 0.

36. Show the customer ID and name for all the customers who have ordered both products with IDs 3 and 4 on the same order.

37. Display the customer names of all customer who have ordered (on the same or different orders) both products with IDs 3 and 4.

38. Review the first query in the "Correlated Subqueries" section. Can you identify a special set of standard prices for which this query will not yield the desired result? How might you rewrite the query to handle this situation?

39. Write an SQL query to list the order number and order quantity for all customer orders for which the order quantity is greater than the average order quantity of that product. (Hint: This involves using a correlated subquery.)

40. Write an SQL query to list the salesperson who has sold the most computer desks.

41. Display in product ID order the product ID and total amount ordered of that product by the customer who has bought the most of that product; use a derived table in a FROM clause to answer this query.

42. Display employee information for all the employees in each state who were hired before the most recently hired person in that state.

43. The head of marketing is interested in some opportunities for cross-selling of products. She thinks that the way to identify cross-selling opportunities is to know for each product how many other products are sold to the same customer on the same order (e.g., a product that is bought by a customer in the same order with lots of other products is a better candidate for cross-selling than a product bought by itself).

a. To help the marketing manager, first list the IDs for all the products that have sold in total more than 20 units across all orders. (These are popular products, which are the only products she wants to consider as triggers for potential cross-selling.)

b. Make a new query that lists all the IDs for the orders that include products that satisfy the first query, along with the number of products on those orders. Only orders with three or more products on them are of interest to the marketing manager. Write this query as general as possible to cover any answer to the first query, which might change over time. To clarify, if product X is one of the products that is in the answer set from part a, then in part b we want to see the desired order information for orders that include product X.

c. The marketing manager needs to know what other products were sold on the orders that are in the result for part b. (Again, write this query for the general, not specific, result to the query in part b.) These are products that are sold, for example, with product X from part a, and these are the ones that if people buy that product, we'd want to try to cross-sell them product X because history says they are likely to buy it along with what else they are buying. Write a query to identify these other products by ID and description. It is okay to include "product X" in your result (i.e., you don't need to exclude the products in the result of part a.).

44. For each product, display in ascending order, by product ID, the product ID and description, along with the customer ID and name for the customer who has bought the most of that product; also show the total quantity ordered by that customer (who has bought the most of that product). Use a correlated subquery.

Field Exercises

1. Conduct a search of the Web to locate as many links as possible that discuss SQL standards.
2. Compare two versions of SQL to which you have access, such as Microsoft Access and Oracle SQL*Plus. Identify at least five similarities and three dissimilarities in the SQL code from these two SQL systems. Do the dissimilarities cause results to differ?

References

DeLoach, A. 1987. "The Path to Writing Efficient Queries in SQL/DS." *Database Programming & Design* 1,1 (January): 26–32.

Eisenberg, A., J. Melton, K. Kulkarni, J. E. Michels, and F. Zemke. 2004. "SQL:2003 Has Been Published." *SIGMOD Record* 33,1 (March):119–126.

Gulutzan, P., and T. Pelzer. 1999. *SQL-99 Complete, Really!* Lawrence, KS: R&D Books.

Holmes, J. 1996. "More Paths to Better Performance." *Database Programming & Design* 9, 2 (February):47–48.

Mullins, C. S. 1995. "The Procedural DBA." *Database Programming & Design* 8,12 (December): 40–45.

Rennhackkamp, M. 1996. "Trigger Happy." *DBMS* 9,5 (May): 89–91, 95.

Zemke, F., K. Kulkarni, A. Witkowski, and B. Lyle. 1999. "Introduction to OLAP Functions." ISO/IEC JTC1/SC32 WG3: YGJ.068 ANSI NCITS H2–99–154r2.

Further Reading

American National Standards Institute. 2000. *ANSI Standards Action* 31,11 (June 2): 20.

Celko, J. 2006. *Analytics and OLAP in SQL.* San Francisco: Morgan Kaufmann.

Codd, E. F. 1970. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM* 13,6 (June): 77–87.

Date, C. J., and H. Darwen. 1997. *A Guide to the SQL Standard.* Reading, MA: Addison-Wesley.

Itzik, B., L. Kollar, and D. Sarka. 2006. *Inside Microsoft SQL Server 2005 T-SQL Querying*. Redmond, WA: Microsoft Press.

Itzik B., D. Sarka, and R. Wolter. 2006. *Inside Microsoft SQL Server 2005: T-SQL Programming*. Redmond, WA: Microsoft Press.

Kulkarni, K. 2004. "Overview of SQL:2003." Accessed at **www.wiscorp.com/SQLStandards.html#keyreadings**.

Melton, J. 1997. "A Case for SQL Conformance Testing." *Database Programming & Design* 10,7 (July): 66–69.

van der Lans, R. F. 1993. *Introduction to SQL*, 2nd ed. Workingham, UK: Addison-Wesley.

Winter, R. 2000. "SQL-99's New OLAP Functions." *Intelligent Enterprise* 3,2 (January 20): 62, 64–65.

Winter, R. 2000. "The Extra Mile." *Intelligent Enterprise* 3,10 (June 26): 62–64.

See also "Further Reading" in Chapter 6.

Web Resources

www.ansi.org Web site of the American National Standards Institute. Contains information on the ANSI federation and the latest national and international standards.

www.coderecipes.net Web site that explains and shows examples for a wide range of SQL commands.

www.fluffycat.com/SQL/ Web site that defines a sample database and shows examples of SQL queries against this database.

www.iso.ch The International Organization for Standardization's (ISO's) Web site, which provides information about the ISO. Copies of current standards may be purchased here.

www.sqlcourse.com and www.sqlcourse2.com Web sites that provide tutorials for a subset of ANSI SQL with a practice database.

standards.ieee.org The home page of the IEEE standards organization.

www.tizag.com/sqlTutorial/ Web site that provides a set of tutorials on SQL concepts and commands.

http://troelsarvin.blogspot.com/ Blog that provides a detailed comparison of different SQL implementations, including DB2, Microsoft SQL, MySQL, Oracle, and PostGreSQL

www.teradatastudentnetwork.com Web site where your instructor may have created some course environments for you to use Teradata SQL Assistant, Web Edition, with one or more of the Pine Valley Furniture and Mountain View Community Hospital data sets for this text.

CASE

Mountain View Community Hospital

Case Description

Use the databases you implemented in Chapter 6 for Mountain View Community Hospital to complete the case questions and case exercises.

Case Questions

1. Does your SQL-based DBMS support dynamic SQL, functions, triggers, stored procedures, and UDTs?
2. HIPAA's privacy and security rules mandate audit controls "that record and examine activity in information systems that contain or use electronic protected health information" [§164.312(b)]. How can DDL triggers be used in support of this mandate?

Case Exercises

1. Using the small sample database you created for Dr. Z in Case Exercise 1 in Chapter 6, write queries that illustrate the more complex queries covered in this chapter:
   - **a.** Select information from two or more tables (e.g., all the details of all the visits of a patient, etc.).
   - **b.** Use subquery syntax (e.g., a listing of all the patients who reported pain that exceeded the average pain for all visits).
   - **c.** Return a result table that could be used to produce a report, sorted by patient name or date, for a particular week or after a particular date, or a listing of patient visits for patients assigned to a specific social worker.
2. Review the exercises below and select several to attempt. You will probably need to add to your prototype and populate your tables with sample data in order to test your queries:
   - **a.** For a given physician, which treatments has that physician performed on each patient referred by that physician to the hospital?
   - **b.** For the query in part a, also include physicians who have not referred patients to the hospital.
   - **c.** For each patient, what is the average number of treatments performed on him or her by each physician who has treated that patient?
   - **d.** List all patients who have received no treatments.
   - **e.** For each nurse in charge, what is the total number of hours worked by all employees who work in the care center which that nurse supervises?
   - **f.** Which technicians have more than one skill listed? Which technicians have no skills listed?
   - **g.** Determine whether any outpatients were accidentally assigned to resident beds.
   - **h.** Determine which item is consumed most.
   - **i.** Determine which physicians prescribe the most expensive item.
   - **j.** Return a result table that could be used to produce a hospital report, such as nursing staff assigned to each care center.
   - **k.** Use the UNION statement to provide a combined listing of care center names and their locations as well as laboratories and their location. The list should be sorted by location, in ascending order. (You should use aliases to rename the fields in this query.)

Project Assignments

P1. Write and execute the queries for the five reports you identified in Chapter 5.

P2. Identify opportunities for using triggers in your database and create at least one DDL trigger. For example, the claims manager at the hospital may need to know that a patient's health insurance has been updated.