Chapter 4

Query Formulation with SQL

Learning Objectives

This chapter provides the foundation for query formulation using the industry standard Structured Query Language (SQL). Query formulation is the process of converting a request for data into a statement of a database language such as SQL. After this chapter the student should have acquired the following knowledge and skills:

- Write SQL SELECT statements for queries involving the restrict, project, and join operators.
- Use the critical questions to transform a problem statement into a database representation.
- Write SELECT statements for difficult joins involving three or more tables, self joins, and multiple joins between tables.
- Understand the meaning of the GROUP BY clause using the conceptual evaluation process.
- Write English descriptions to document SQL statements.
- Write INSERT, UPDATE, and DELETE statements to change the rows of a table.

Overview

Chapter 3 provided a foundation for using relational databases. Most importantly, you learned about connections among tables and fundamental operators to extract useful data. This chapter shows you how to apply this knowledge in using the data manipulation statements of SQL.

Much of your skill with SQL or other computer languages is derived from imitating examples. This chapter provides many examples to facilitate your learning process. Initially you are presented with relatively simple examples so that you become comfortable with the basics of the SQL SELECT statement. To prepare you for more difficult examples, two problem-solving guidelines (conceptual evaluation process and critical questions) are presented. The conceptual evaluation process explains the meaning of the SELECT statement through the sequence of operations and intermediate tables that produce the result.

The critical questions help you transform a problem statement into a relational database representation in a language such as SQL. These guidelines are used to help formulate and explain the advanced problems presented in the last part of this chapter.

4.1 Background

Before using SQL, it is informative to understand its history and scope. The history reveals the origin of the name and the efforts to standardize the language. The scope puts the various parts of SQL into perspective. You have already seen the CREATE TABLE statement in Chapter 3. The SELECT, UPDATE, DELETE, and INSERT statements are the subject of this chapter and Chapter 9. To broaden your understanding, you should be aware of other parts of SQL and different usage contexts.

4.1.1 Brief History of SQL

The Structured Query Language (SQL) has a colorful history. Table 4.1 depicts the highlights of SQL's development. SQL began life as the SQUARE language in IBM's System R project. The System R project was a response to the interest in relational databases sparked by Dr. Ted Codd, an IBM fellow who wrote a famous paper in 1970 about relational databases. The SQUARE language was somewhat mathematical in nature. After conducting human factors experiments, the IBM research team revised the language and renamed it SEQUEL (a follow-up to SQUARE). After another revision, the language was dubbed SEQUEL 2. Its current name, SQL, resulted from legal issues surrounding the name SEQUEL. Because of this naming history, a number of database professionals, particularly those working during the 1970s, pronounce the name as "sequel" rather than SQL.

SQL is now an international standard although it was not always so.[1] With the force of IBM behind SQL, many imitators used some variant of SQL. Such was the old order of the computer industry when IBM was dominant. It may seem surprising, but IBM was not the first company to commercialize SQL. Until a standards effort developed in the 1980s, SQL was in a state of confusion. Many vendors implemented different subsets of SQL with unique extensions. The standards efforts by the American National Standards Institute (ANSI), the International Organization for Standards (ISO), and the International Electrotechnical Commission (IEC) have restored some order. Although SQL was not initially the best database language developed, the standards efforts have improved the language as well as standardized its specification.

TABLE 4.1
SQL Timeline

Year	Event
1972	System R project at IBM Research Labs
1974	SQUARE language developed
1975	Language revision and name change to SEQUEL
1976	Language revision and name change to SEQUEL 2
1977	Name change to SQL
1978	First commercial implementation by Oracle Corporation
1981	IBM product SQL/DS featuring SQL
1986	SQL-86 (SQL1) standard approved
1989	SQL-89 standard approved (revision to SQL-86)
1992	SQL-92 (SQL2) standard approved
1999	SQL:1999 (SQL3) standard approved
2003	SQL:2003 approved

[1] Dr. Michael Stonebraker, an early database pioneer, has even referred to SQL as "intergalactic data speak."

The size and scope of the SQL standard has increased significantly since the first standard was adopted. The original standard (SQL-86) contained about 150 pages, while the SQL-92 standard contained more than 600 pages. In contrast, the most recent standards (SQL:1999 and SQL:2003) contained more than 2,000 pages. The early standards (SQL-86 and SQL-89) had two levels (entry and full). SQL-92 added a third level (entry, intermediate, and full). The SQL:1999 and SQL:2003 standards contain a single level called Core SQL along with parts and packages for noncore features. SQL:2003 contains three core parts, six optional parts, and seven optional packages.

The weakness of the SQL standards is the lack of conformance testing. Until 1996, the U.S. Department of Commerce's National Institute of Standards and Technology conducted conformance tests to provide assurance that government software can be ported among conforming DBMSs. Since 1996, however, DBMS vendor claims have substituted for independent conformance testing. Even for Core SQL, the major vendors lack support for some features and provide proprietary support for other features. With the optional parts and packages, conformance has much greater variance. Writing portable SQL code requires careful study for Core SQL but is not possible for advanced parts of SQL.

The presentation in this chapter is limited to a subset of Core SQL:2003. Most features presented in this chapter were part of SQL-92 as well as Core SQL:2003. Other chapters present other parts of Core SQL as well as important features from selected SQL:2003 packages.

4.1.2 Scope of SQL

SQL was designed as a language for database definition, manipulation, and control. Table 4.2 shows a quick summary of important SQL statements. Only database administrators use most of the database definition and database control statements. You have already seen the CREATE TABLE statement in Chapter 3. This chapter and Chapter 9 cover the database manipulation statements. Power users and analysts use the database manipulation statements. Chapter 10 covers the CREATE VIEW statement. The CREATE VIEW statement can be used by either database administrators or analysts. Chapter 11 covers the CREATE TRIGGER statement used by both database administrators and analysts. Chapter 14 covers the GRANT, REVOKE, and CREATE ASSERTION statements used primarily by database administrators. The transaction control statements (COMMIT and ROLLBACK) presented in Chapter 15 are used by analysts.

SQL usage contexts
SQL statements can be used stand-alone with a specialized editor, or embedded inside a computer program.

SQL can be used in two contexts: stand-alone and embedded. In the stand-alone context, the user submits SQL statements with the use of a specialized editor. The editor alerts the user to syntax errors and sends the statements to the DBMS. The presentation in this chapter assumes stand-alone usage. In the embedded context, an executing program submits SQL statements, and the DBMS sends results back to the program. The program includes

TABLE 4.2 Selected SQL Statements

Statement Type	Statements	Purpose
Database definition	CREATE SCHEMA, TABLE, VIEW	Define a new database, table, and view
	ALTER TABLE	Modify table definition
Database manipulation	SELECT	Retrieve contents of tables
	UPDATE, DELETE, INSERT	Modify, remove, and add rows
Database control	COMMIT, ROLLBACK	Complete, undo transaction
	GRANT, REVOKE	Add and remove access rights
	CREATE ASSERTION	Define integrity constraint
	CREATE TRIGGER	Define database rule

SQL statements along with statements of the host programming language such as Java or Visual Basic. Additional statements allow SQL statements (such as SELECT) to be used inside a computer program. Chapter 11 covers embedded SQL.

4.2 Getting Started with the SELECT Statement

The SELECT statement supports data retrieval from one or more tables. This section describes a simplified format of the SELECT statement. More complex formats are presented in Chapter 9. The SELECT statement described here has the following format:

```
SELECT <list of columns and expressions usually involving columns>
  FROM <list of tables and join operations>
  WHERE <list of row conditions connected by AND, OR, NOT>
  GROUP BY <list of grouping columns>
  HAVING <list of group conditions connected by AND, OR, NOT>
  ORDER BY <list of sorting specifications>
```

expression
a combination of constants, column names, functions, and operators that produces a value. In conditions and result columns, expressions can be used in any place that column names can appear.

In the preceding format, the uppercase words are keywords. You replace the angle brackets < > with information to make a meaningful statement. For example, after the keyword SELECT, type the list of columns that should appear in the result, but do not type the angle brackets. The result list can include columns such as *StdFirstName* or expressions involving constants, column names, and functions. Example expressions are *Price * Qty* and *1.1 * FacSalary*. To make meaningful names for computed columns, you can rename a column in the result table using the AS keyword. For example, *SELECT Price * Qty AS Amount* renames the expression *Price * Qty* to *Amount* in the result table.

To depict this SELECT format and show the meaning of statements, this chapter shows numerous examples. Examples are provided for both Microsoft Access, a popular desktop DBMS, and Oracle, a prominent enterprise DBMS. Most examples execute on both systems. Unless noted, the examples run on the 1997 through 2003 versions of Access and the 8i through and 10g versions of Oracle. Examples that only execute on one product are marked. In addition to the examples, Appendix 4.B summarizes syntax differences among major DBMSs.

The examples use the university database tables introduced in Chapter 3. Tables 4.3 through 4.7 list the contents of the tables. CREATE TABLE statements are listed in Appendix 3.A. For your reference, the relationship diagram showing the primary and foreign

TABLE 4.3 **Sample *Student* Table**

StdSSN	StdFirstName	StdLastName	StdCity	StdState	StdZip	StdMajor	StdClass	StdGPA
123-45-6789	HOMER	WELLS	SEATTLE	WA	98121-1111	IS	FR	3.00
124-56-7890	BOB	NORBERT	BOTHELL	WA	98011-2121	FIN	JR	2.70
234-56-7890	CANDY	KENDALL	TACOMA	WA	99042-3321	ACCT	JR	3.50
345-67-8901	WALLY	KENDALL	SEATTLE	WA	98123-1141	IS	SR	2.80
456-78-9012	JOE	ESTRADA	SEATTLE	WA	98121-2333	FIN	SR	3.20
567-89-0123	MARIAH	DODGE	SEATTLE	WA	98114-0021	IS	JR	3.60
678-90-1234	TESS	DODGE	REDMOND	WA	98116-2344	ACCT	SO	3.30
789-01-2345	ROBERTO	MORALES	SEATTLE	WA	98121-2212	FIN	JR	2.50
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	98114-1332	IS	SR	4.00
890-12-3456	LUKE	BRAZZI	SEATTLE	WA	98116-0021	IS	SR	2.20
901-23-4567	WILLIAM	PILGRIM	BOTHELL	WA	98113-1885	IS	SO	3.80

TABLE 4.4A **Sample *Faculty* Table (first part)**

FacSSN	FacFirstName	FacLastName	FacCity	FacState	FacDept	FacRank	FacSalary
098-76-5432	LEONARD	VINCE	SEATTLE	WA	MS	ASST	$35,000
543-21-0987	VICTORIA	EMMANUEL	BOTHELL	WA	MS	PROF	$120,000
654-32-1098	LEONARD	FIBON	SEATTLE	WA	MS	ASSC	$70,000
765-43-2109	NICKI	MACON	BELLEVUE	WA	FIN	PROF	$65,000
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	MS	ASST	$40,000
987-65-4321	JULIA	MILLS	SEATTLE	WA	FIN	ASSC	$75,000

TABLE 4.4B
Sample *Faculty* Table (second part)

FacSSN	FacSupervisor	FacHireDate	FacZipCode
098-76-5432	654-32-1098	10-Apr-1995	98111-9921
543-21-0987		15-Apr-1996	98011-2242
654-32-1098	543-21-0987	01-May-1994	98121-0094
765-43-2109		11-Apr-1997	98015-9945
876-54-3210	654-32-1098	01-Mar-1999	98114-1332
987-65-4321	765-43-2109	15-Mar-2000	98114-9954

TABLE 4.5 **Sample *Offering* Table**

OfferNo	CourseNo	OffTerm	OffYear	OffLocation	OffTime	FacSSN	OffDays
1111	IS320	SUMMER	2006	BLM302	10:30 AM		MW
1234	IS320	FALL	2005	BLM302	10:30 AM	098-76-5432	MW
2222	IS460	SUMMER	2005	BLM412	1:30 PM		TTH
3333	IS320	SPRING	2006	BLM214	8:30 AM	098-76-5432	MW
4321	IS320	FALL	2005	BLM214	3:30 PM	098-76-5432	TTH
4444	IS320	WINTER	2006	BLM302	3:30 PM	543-21-0987	TTH
5555	FIN300	WINTER	2006	BLM207	8:30 AM	765-43-2109	MW
5678	IS480	WINTER	2006	BLM302	10:30 AM	987-65-4321	MW
5679	IS480	SPRING	2006	BLM412	3:30 PM	876-54-3210	TTH
6666	FIN450	WINTER	2006	BLM212	10:30 AM	987-65-4321	TTH
7777	FIN480	SPRING	2006	BLM305	1:30 PM	765-43-2109	MW
8888	IS320	SUMMER	2006	BLM405	1:30 PM	654-32-1098	MW
9876	IS460	SPRING	2006	BLM307	1:30 PM	654-32-1098	TTH

TABLE 4.6
Sample *Course* Table

CourseNo	CrsDesc	CrsUnits
FIN300	FUNDAMENTALS OF FINANCE	4
FIN450	PRINCIPLES OF INVESTMENTS	4
FIN480	CORPORATE FINANCE	4
IS320	FUNDAMENTALS OF BUSINESS PROGRAMMING	4
IS460	SYSTEMS ANALYSIS	4
IS470	BUSINESS DATA COMMUNICATIONS	4
IS480	FUNDAMENTALS OF DATABASE MANAGEMENT	4

TABLE 4.7
Sample *Enrollment* Table

OfferNo	StdSSN	EnrGrade
1234	123-45-6789	3.3
1234	234-56-7890	3.5
1234	345-67-8901	3.2
1234	456-78-9012	3.1
1234	567-89-0123	3.8
1234	678-90-1234	3.4
4321	123-45-6789	3.5
4321	124-56-7890	3.2
4321	789-01-2345	3.5
4321	876-54-3210	3.1
4321	890-12-3456	3.4
4321	901-23-4567	3.1
5555	123-45-6789	3.2
5555	124-56-7890	2.7
5678	123-45-6789	3.2
5678	234-56-7890	2.8
5678	345-67-8901	3.3
5678	456-78-9012	3.4
5678	567-89-0123	2.6
5679	123-45-6789	2
5679	124-56-7890	3.7
5679	678-90-1234	3.3
5679	789-01-2345	3.8
5679	890-12-3456	2.9
5679	901-23-4567	3.1
6666	234-56-7890	3.1
6666	567-89-0123	3.6
7777	876-54-3210	3.4
7777	890-12-3456	3.7
7777	901-23-4567	3.4
9876	124-56-7890	3.5
9876	234-56-7890	3.2
9876	345-67-8901	3.2
9876	456-78-9012	3.4
9876	567-89-0123	2.6
9876	678-90-1234	3.3
9876	901-23-4567	4

keys is repeated in Figure 4.1. Recall that the *Faculty1* table with relationship to the *Faculty* table represents a self-referencing relationship with *FacSupervisor* as the foreign key.

4.2.1 Single Table Problems

Let us begin with the simple SELECT statement in Example 4.1. In all the examples, keywords appear in uppercase while information specific to the query appears in mixed case. In Example 4.1, only the *Student* table is listed in the FROM clause because the conditions in the WHERE clause and columns after the SELECT keyword involve only the *Student* table. In Oracle, a semicolon or / (on a separate line) terminates a statement.

FIGURE 4.1
Relationship Window for the University Database

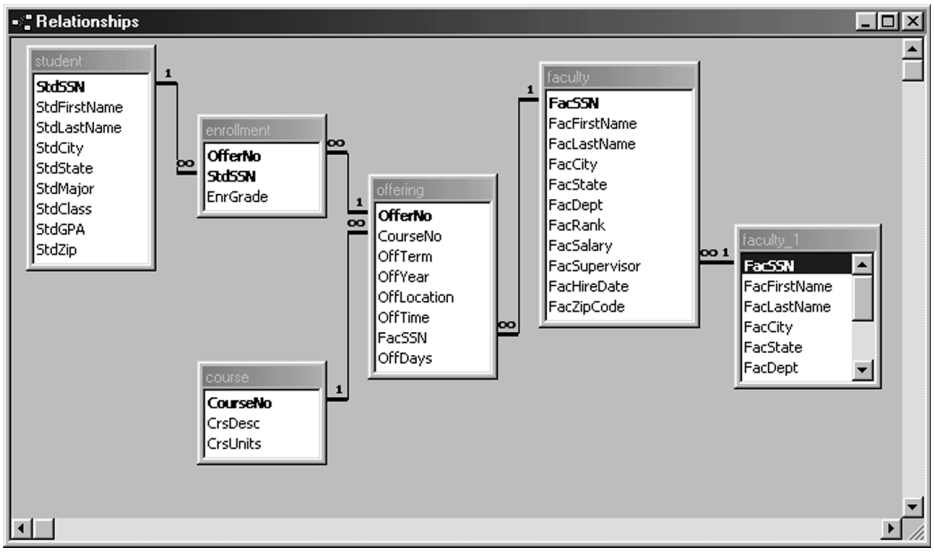


TABLE 4.8
Standard Comparison Operators

Comparison Operator	Meaning
=	equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
< > or !=	not equal (check your DBMS)

EXAMPLE 4.1 **Testing Rows Using the WHERE Clause**

Retrieve the name, city, and grade point average (GPA) of students with a high GPA (greater than or equal to 3.7). The result follows the SELECT statement.

```
SELECT  StdFirstName, StdLastName, StdCity, StdGPA
    FROM Student
    WHERE StdGPA >= 3.7
```

StdFirstName	StdLastName	StdCity	StdGPA
CRISTOPHER	COLAN	SEATTLE	4.00
WILLIAM	PILGRIM	BOTHELL	3.80

Table 4.8 depicts the standard comparison operators. Note that the symbol for some operators depends on the DBMS.

Example 4.2 is even simpler than Example 4.1. The result is identical to the original *Faculty* table in Table 4.4. Example 4.2 uses a shortcut to list all columns. The asterisk * in the column list indicates that all columns of the tables in the FROM clause appear in the result. The asterisk serves as a wildcard character matching all column names.

EXAMPLE 4.2 **Show all Columns**

List all columns and rows of the *Faculty* table. The resulting table is shown in two parts.

```
SELECT * FROM Faculty
```

FacSSN	FacFirstName	FacLastName	FacCity	FacState	FacDept	FacRank	FacSalary
098-76-5432	LEONARD	VINCE	SEATTLE	WA	MS	ASST	$35,000
543-21-0987	VICTORIA	EMMANUEL	BOTHELL	WA	MS	PROF	$120,000
654-32-1098	LEONARD	FIBON	SEATTLE	WA	MS	ASSC	$70,000
765-43-2109	NICKI	MACON	BELLEVUE	WA	FIN	PROF	$65,000
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	MS	ASST	$40,000
987-65-4321	JULIA	MILLS	SEATTLE	WA	FIN	ASSC	$75,000

FacSSN	FacSupervisor	FacHireDate	FacZipCode
098-76-5432	654-32-1098	10-Apr-1995	98111-9921
543-21-0987		15-Apr-1996	98011-2242
654-32-1098	543-21-0987	01-May-1994	98121-0094
765-43-2109		11-Apr-1997	98015-9945
876-54-3210	654-32-1098	01-Mar-1999	98114-1332
987-65-4321	765-43-2109	15-Mar-2000	98114-9954

Example 4.3 depicts expressions in the SELECT and WHERE clauses. The expression in the SELECT clause increases the salary by 10 percent. The AS keyword is used to rename the computed column. Without renaming, most DBMSs will generate a meaningless name such as Expr001. The expression in the WHERE clause extracts the year from the hiring date. Because functions for the date data type are not standard, Access and Oracle formulations are provided. To become proficient with SQL on a particular DBMS, you will need to study the available functions especially with date columns.

EXAMPLE 4.3 (Access) **Expressions in SELECT and WHERE Clauses**

List the name, city, and increased salary of faculty hired after 1996. The **year** function extracts the year part of a column with a date data type.

```
SELECT  FacFirstName, FacLastName, FacCity,
        FacSalary*1.1 AS IncreasedSalary, FacHireDate
  FROM  Faculty
  WHERE year(FacHireDate) > 1996
```

FacFirstName	FacLastName	FacCity	IncreasedSalary	FacHireDate
NICKI	MACON	BELLEVUE	71500	11-Apr-1997
CRISTOPHER	COLAN	SEATTLE	44000	01-Mar-1999
JULIA	MILLS	SEATTLE	82500	15-Mar-2000

EXAMPLE 4.3 (Oracle)

Expressions in SELECT and WHERE Clauses

The **to_char** function extracts the four-digit year from the *FacHireDate* column and the **to_number** function converts the character representation of the year into a number.

```
SELECT FacFirstName, FacLastName, FacCity,
          FacSalary*1.1 AS IncreasedSalary, FacHireDate
  FROM Faculty
  WHERE to_number(to_char(FacHireDate, 'YYYY')  ) > 1996
```

Inexact matching supports conditions that match some pattern rather than matching an identical string. One of the most common types of inexact matching is to find values having a common prefix such as "IS4" (400 level IS Courses). Example 4.4 uses the LIKE operator along with a pattern-matching character * to perform prefix matching.[2] The string constant '*IS4**'' means match strings beginning with "IS4" and ending with anything. The wildcard character * matches any string. The Oracle formulation of Example 4.4 uses the percent symbol %, the SQL:2003 standard for the wildcard character. Note that string constants must be enclosed in quotation marks.[3]

EXAMPLE 4.4 (Access)

Inexact Matching with the LIKE Operator

List the senior-level IS courses.

```
SELECT *
  FROM Course
  WHERE CourseNo LIKE 'IS4*'
```

CourseNo	CrsDesc	CrsUnits
IS460	SYSTEMS ANALYSIS	4
IS470	BUSINESS DATA COMMUNICATIONS	4
IS480	FUNDAMENTALS OF DATABASE MANAGEMENT	4

EXAMPLE 4.4 (Oracle)

Inexact Matching with the LIKE Operator

List the senior-level IS courses.

```
SELECT *
  FROM Course
  WHERE CourseNo LIKE 'IS4%'
```

[2] Beginning with Access 2002, the SQL:2003 pattern-matching characters can be used by specifying ANSI 92 query mode in the Options window. Since earlier Access versions do not support this option and this option is not default in Access 2002, the textbook uses the * and ? pattern-matching characters for Access SQL statements.

[3] Most DBMSs require single quotes, the SQL:2003 standard. Microsoft Access allows either single or double quotes for string constants.

Another common type of inexact matching is to match strings containing a substring. To perform this kind of matching, a wildcard character should be used before and after the substring. For example, to find courses containing the word DATABASE anywhere in the course description, write the condition: *CrsDesc LIKE '*DATABASE*'* in Access or *CrsDesc LIKE '%DATABASE%'* in Oracle.

The wildcard character is not the only pattern-matching character. SQL:2003 specifies the underscore character _ to match any single character. Some DBMSs such as Access use the question mark ? to match any single character. In addition, most DBMSs have pattern-matching characters for matching a range of characters (for example, the digits 0 to 9) and any character from a list of characters. The symbols used for these other pattern-matching characters are not standard. To become proficient at writing inexact matching conditions, you should study the pattern-matching characters available with your DBMS.

In addition to performing pattern matching with strings, you can use exact matching with the equality = comparison operator. For example, the condition, *CourseNo = 'IS480'* matches a single row in the *Course* table. For both exact and inexact matching, case sensitivity is an important issue. Some DBMSs such as Microsoft Access are not case sensitive. In Access SQL, the previous condition matches "is480", "Is480", and "iS480" in addition to "IS480". Other DBMSs such as Oracle are case sensitive. In Oracle SQL, the previous condition matches only "IS480", not "is480", "Is480", or "iS480". To alleviate confusion, you can use the Oracle **upper** or **lower** functions to convert strings to upper- or lowercase, respectively.

BETWEEN-AND operator
a shortcut operator to test a numeric or date column against a range of values. The BETWEEN-AND operator returns true if the column is greater than or equal to the first value and less than or equal to the second value.

Example 4.5 depicts range matching on a column with the date data type. In Access SQL, pound symbols enclose date constants, while in Oracle SQL, single quotation marks enclose date constants. Date columns can be compared just like numbers with the usual comparison operators (=, <, etc.). The BETWEEN-AND operator defines a closed interval (includes end points). In Access Example 4.5, the BETWEEN-AND condition is a shortcut for *FacHireDate >= #1/1/1999# AND FacHireDate <= #12/31/2000#.*

EXAMPLE 4.5 (Access)

Conditions on Date Columns

List the name and hiring date of faculty hired in 1999 or 2000.

```
SELECT   FacFirstName, FacLastName, FacHireDate
  FROM   Faculty
  WHERE FacHireDate BETWEEN #1/1/1999# AND #12/31/2000#
```

FacFirstName	FacLastName	FacHireDate
CRISTOPHER	COLAN	01-Mar-1994
JULIA	MILLS	15-Mar-2000

EXAMPLE 4.5 (Oracle)

Conditions on Date Columns

In Oracle SQL, the standard format for dates is DD-Mon-YYYY where DD is the day number, Mon is the month abbreviation, and YYYY is the four-digit year.

```
SELECT FacFirstName, FacLastName, FacHireDate
  FROM Faculty
  WHERE FacHireDate BETWEEN '1-Jan-1999'  AND '31-Dec-2000'
```

Besides testing columns for specified values, you sometimes need to test for the lack of a value. Null values are used when there is no normal value for a column. A null can mean that the value is unknown or the value is not applicable to the row. For the *Offering* table, a null value for *FacSSN* means that the instructor is not yet assigned. Testing for null values is done with the IS NULL comparison operator as shown in Example 4.6. You can also test for a normal value using IS NOT NULL.

EXAMPLE 4.6 **Testing for Nulls**

List the offering number and course number of summer 2006 offerings without an assigned instructor.

```
SELECT OfferNo, CourseNo
  FROM Offering
  WHERE FacSSN IS NULL AND OffTerm = 'SUMMER'
      AND OffYear = 2006
```

OfferNo	CourseNo
1111	IS320

mixing AND and OR
always use parentheses to make the grouping of conditions explicit.

Example 4.7 depicts a complex logical expression involving both logical operators AND and OR. When mixing AND and OR in a logical expression, it is a good idea to use parentheses. Otherwise, the reader of the SELECT statement may not understand how the AND and OR conditions are grouped. Without parentheses, you must depend on the default way that AND and OR conditions are grouped.

EXAMPLE 4.7 **Complex Logical Expression**

List the offer number, course number, and faculty Social Security number for course offerings scheduled in fall 2005 or winter 2006.

```
SELECT OfferNo, CourseNo, FacSSN
  FROM Offering
  WHERE (OffTerm = 'FALL' AND OffYear = 2005)
      OR (OffTerm = 'WINTER' AND OffYear = 2006)
```

OfferNo	CourseNo	FacSSN
1234	IS320	098-76-5432
4321	IS320	098-76-5432
4444	IS320	543-21-0987
5555	FIN300	765-43-2109
5678	IS480	987-65-4321
6666	FIN450	987-65-4321

4.2.2 Joining Tables

Example 4.8 demonstrates a join of the *Course* and *Offering* tables. The join condition *Course.CourseNo = Offering.CourseNo* is specified in the WHERE clause.

EXAMPLE 4.8 (Access)

Join Tables but Show Columns from One Table Only

List the offering number, course number, days, and time of offerings containing the words *database* or *programming* in the course description and taught in spring 2006. The Oracle version of this example uses the % instead of the * as the wildcard character.

```
SELECT OfferNo, Offering.CourseNo, OffDays, OffTime
  FROM Offering, Course
  WHERE OffTerm = 'SPRING' AND OffYear = 2006
       AND (CrsDesc LIKE '*DATABASE*'
        OR CrsDesc LIKE '*PROGRAMMING*')
       AND Course.CourseNo = Offering.CourseNo
```

OfferNo	CourseNo	OffDays	OffTime
3333	IS320	MW	8:30 AM
5679	IS480	TTH	3:30 PM

There are two additional points of interest about Example 4.8. First, the *CourseNo* column names must be *qualified* (prefixed) with a table name (*Course* or *Offering*). Otherwise, the SELECT statement is ambiguous because *CourseNo* can refer to a column in either the *Course* or *Offering* tables. Second, both tables must be listed in the FROM clause even though the result columns come from only the *Offering* table. The *Course* table is needed in the FROM clause because conditions in the WHERE clause reference *CrsDesc,* a column of the *Course* table.

Example 4.9 demonstrates another join, but this time the result columns come from both tables. There are conditions on each table in addition to the join conditions. The Oracle formulation uses the % instead of the * as the wildcard character.

EXAMPLE 4.9 (Access)

Join Tables and Show Columns from Both Tables

List the offer number, course number, and name of the instructor of IS course offerings scheduled in fall 2005 taught by assistant professors.

```
SELECT OfferNo, CourseNo, FacFirstName, FacLastName
  FROM Offering, Faculty
  WHERE OffTerm = 'FALL' AND OffYear = 2005
       AND FacRank = 'ASST' AND CourseNo LIKE 'IS*'
       AND Faculty.FacSSN = Offering.FacSSN
```

OfferNo	CourseNo	FacFirstName	FacLastName
1234	IS320	LEONARD	VINCE
4321	IS320	LEONARD	VINCE

EXAMPLE 4.9 (Oracle)

Join Tables and Show Columns from Both Tables

List the offer number, course number, and name of the instructor of IS course offerings scheduled in fall 2005 taught by assistant professors.

```
SELECT OfferNo, CourseNo, FacFirstName, FacLastName
  FROM Offering, Faculty
  WHERE OffTerm = 'FALL' AND OffYear = 2005
        AND FacRank = 'ASST' AND CourseNo LIKE 'IS%'
        AND Faculty.FacSSN = Offering.FacSSN
```

In the SQL:2003 standard, the join operation can be expressed directly in the FROM clause rather than being expressed in both the FROM and WHERE clauses as shown in Examples 4.8 and 4.9. Note that Oracle beginning with version 9i supports join operations in the FROM clause, but previous versions do not support join operations in the FROM clause. To make a join operation in the FROM clause, use the keywords INNER JOIN as shown in Example 4.10. The join conditions are indicated by the ON keyword inside the FROM clause. Notice that the join condition no longer appears in the WHERE clause.

EXAMPLE 4.10 (Access)

Join Tables Using a Join Operation in the FROM Clause

List the offer number, course number, and name of the instructor of IS course offerings scheduled in fall 2005 that are taught by assistant professors (result is identical to Example 4.9). In Oracle, you should use the % instead of *.

```
SELECT OfferNo, CourseNo, FacFirstName, FacLastName
  FROM Offering INNER JOIN Faculty
     ON Faculty.FacSSN = Offering.FacSSN
  WHERE OffTerm = 'FALL'  AND OffYear = 2005
     AND FacRank = 'ASST' AND CourseNo LIKE 'IS*'
```

GROUP BY reminder
the columns in the SELECT clause must either be in the GROUP BY clause or be part of a summary calculation with an aggregate function.

4.2.3 Summarizing Tables with GROUP BY and HAVING

So far, the results of all examples in this section relate to individual rows. Even Example 4.9 relates to a combination of columns from individual *Offering* and *Faculty* rows. As mentioned in Chapter 3, it is sometimes important to show summaries of rows. The GROUP BY and HAVING clauses are used to show results about groups of rows rather than individual rows.

Example 4.11 depicts the GROUP BY clause to summarize groups of rows. Each result row contains a value of the grouping column (*StdMajor*) along with the aggregate calculation summarizing rows with the same value for the grouping column. The GROUP BY clause must contain every column in the SELECT clause except for aggregate expressions. For example, adding the *StdClass* column in the SELECT clause would make Example 4.11 invalid unless *StdClass* was also added to the GROUP BY clause.

EXAMPLE 4.11 **Grouping on a Single Column**

Summarize the averageGPA of students by major.

```
SELECT StdMajor, AVG(StdGPA) AS AvgGPA
 FROM Student
 GROUP BY StdMajor
```

StdMajor	AvgGPA
ACCT	3.39999997615814
FIN	2.80000003178914
IS	3.23333330949148

COUNT function usage
COUNT(*) and COUNT(column) produce identical results except when "column" contains null values. See Chapter 9 for more details about the effect of null values on aggregate functions.

Table 4.9 shows the standard aggregate functions. If you have a statistical calculation that cannot be performed with these functions, check your DBMS. Most DBMSs feature many functions beyond these standard ones.

The COUNT, AVG, and SUM functions support the DISTINCT keyword to restrict the computation to unique column values. Example 4.12 demonstrates the DISTINCT keyword for the COUNT function. This example retrieves the number of offerings in a year as well as the number of distinct courses taught. Some DBMSs such as Microsoft Access

EXAMPLE 4.12 (Oracle) **Counting Rows and Unique Column Values**

Summarize the number of offerings and unique courses by year.

```
SELECT OffYear, COUNT(*) AS NumOfferings,
        COUNT(DISTINCT CourseNo) AS NumCourses
 FROM Offering
 GROUP BY OffYear
```

OffYear	NumOfferings	NumCourses
2005	3	2
2006	10	6

TABLE 4.9
Standard Aggregate Functions

Aggregate Function	Meaning and Comments
COUNT(*)	Computes the number of rows.
COUNT(column)	Counts the non-null values in column; DISTINCT can be used to count the unique column values.
AVG	Computes the average of a numeric column or expression excluding null values; DISTINCT can be used to compute the average of unique column values.
SUM	Computes the sum of a numeric column or expression excluding null values; DISTINCT can be used to compute the average of unique column values.
MIN	Computes the smallest value. For string columns, the collating sequence is used to compare strings.
MAX	Computes the largest value. For string columns, the collating sequence is used to compare strings.

WHERE vs. HAVING use the WHERE clause for conditions that can be tested on individual rows. Use the HAVING clause for conditions that can be tested only on groups. Conditions in the HAVING clause should involve aggregate functions, whereas conditions in the WHERE clause cannot involve aggregate functions.

do not support the DISTINCT keyword inside of aggregate functions. Chapter 9 presents an alternative formulation in Access SQL to compensate for the inability to use the DISTINCT keyword inside the COUNT function.

Examples 4.13 and 4.14 contrast the WHERE and HAVING clauses. In Example 4.13, the WHERE clause selects upper-division students (juniors or seniors) before grouping on major. Because the WHERE clause eliminates students before grouping occurs, only upper-division students are grouped. In Example 4.14, a HAVING condition retains groups with an average GPA greater than 3.1. The HAVING clause applies to groups of rows, whereas the WHERE clause applies to individual rows. To use a HAVING clause, there must be a GROUP BY clause.

EXAMPLE 4.13 **Grouping with Row Conditions**

Summarize the average GPA of upper-division (junior or senior) students by major.

```
SELECT StdMajor, AVG(StdGPA) AS AvgGpa
  FROM Student
  WHERE StdClass = 'JR' OR StdClass = 'SR'
  GROUP BY StdMajor
```

StdMajor	AvgGPA
ACCT	3.5
FIN	2.800000031789
IS	3.149999976158

EXAMPLE 4.14 **Grouping with Row and Group Conditions**

Summarize the average GPA of upper-division (junior or senior) students by major. Only list the majors with average GPA greater than 3.1.

```
SELECT StdMajor, AVG(StdGPA) AS AvgGpa
  FROM Student
  WHERE StdClass IN ('JR', 'SR')
  GROUP BY StdMajor
  HAVING AVG(StdGPA) > 3.1
```

StdMajor	AvgGPA
ACCT	3.5
IS	3.149999976158

HAVING reminder the HAVING clause must be preceded by the GROUP BY clause.

One other point about Examples 4.13 and 4.14 is the use of the OR operator as compared to the IN operator (set element of operator). The WHERE condition in Examples 4.13 and 4.14 retains the same rows. The IN condition is true if *StdClass* matches any value in the parenthesized list. Chapter 9 provides additional explanation about the IN operator for nested queries.

To summarize all rows, aggregate functions can be used in SELECT without a GROUP BY clause as demonstrated in Example 4.15. The result is always a single row containing just the aggregate calculations.

EXAMPLE 4.15 **Grouping all Rows**

List the number of upper-division students and their average GPA.

```
SELECT COUNT(*) AS StdCnt, AVG(StdGPA) AS AvgGPA
  FROM Student
  WHERE StdClass = 'JR' OR StdClass = 'SR'
```

StdCnt	AvgGPA
8	3.0625

Sometimes it is useful to group on more than one column as demonstrated by Example 4.16. The result shows one row for each combination of *StdMajor* and *StdClass*. Some rows have the same value for both aggregate calculations because there is only one associated row in the *Student* table. For example, there is only one row for the combination ('ACCT', 'JR').

EXAMPLE 4.16 **Grouping on Two Columns**

Summarize the minimum and maximum GPA of students by major and class.

```
SELECT StdMajor, StdClass, MIN(StdGPA) AS MinGPA, MAX(StdGPA) AS
          MaxGPA
  FROM Student
  GROUP BY StdMajor, StdClass
```

StdMajor	StdClass	MinGPA	MaxGPA
ACCT	JR	3.5	3.5
ACCT	SO	3.3	3.3
FIN	JR	2.5	2.7
FIN	SR	3.2	3.2
IS	FR	3	3
IS	JR	3.6	3.6
IS	SO	3.8	3.8
IS	SR	2.2	4

A powerful combination is to use grouping with joins. There is no reason to restrict grouping to just one table. Often, more useful information is obtained by summarizing rows that result from a join. Example 4.17 demonstrates grouping applied to a join between *Course* and *Offering*. It is important to note that the join is performed before the grouping occurs. For example, after the join, there are six rows for BUSINESS PROGRAMMING. Because queries combining joins and grouping can be difficult to understand, Section 4.3 provides a more detailed explanation.

EXAMPLE 4.17 (Access)

Combining Grouping and Joins

Summarize the number of IS course offerings by course description.

```
SELECT CrsDesc, COUNT(*) AS OfferCount
  FROM Course, Offering
  WHERE Course.CourseNo = Offering.CourseNo
        AND Course.CourseNo LIKE 'IS*'
 GROUP   BY CrsDesc
```

CrsDesc	OfferCount
FUNDAMENTALS OF BUSINESS PROGRAMMING	6
FUNDAMENTALS OF DATABASE MANAGEMENT	2
SYSTEMS ANALYSIS	2

EXAMPLE 4.17 (Oracle)

Combining Grouping and Joins

Summarize the number of IS course offerings by course description.

```
SELECT CrsDesc, COUNT(*) AS OfferCount
  FROM Course, Offering
  WHERE Course.CourseNo = Offering.CourseNo
      AND  Course.CourseNo LIKE 'IS%'
  GROUP BY CrsDesc
```

4.2.4 Improving the Appearance of Results

We finish this section with two parts of the SELECT statement that can improve the appearance of results. Examples 4.18 and 4.19 demonstrate sorting using the ORDER BY clause. The sort sequence depends on the date type of the sorted field (numeric for

EXAMPLE 4.18

Sorting on a Single Column

List the GPA, name, city, and state of juniors. Order the result by GPA in ascending order.

```
SELECT StdGPA, StdFirstName, StdLastName, StdCity, StdState
  FROM Student
  WHERE StdClass = 'JR'
  ORDER BY StdGPA
```

StdGPA	StdFirstName	StdLastName	StdCity	StdState
2.50	ROBERTO	MORALES	SEATTLE	WA
2.70	BOB	NORBERT	BOTHELL	WA
3.50	CANDY	KENDALL	TACOMA	WA
3.60	MARIAH	DODGE	SEATTLE	WA

numeric data types, ASCII collating sequence for string fields, and calendar sequence for data fields). By default, sorting occurs in ascending order. The keyword DESC can be used after a column name to sort in descending order as demonstrated in Example 4.19.

EXAMPLE 4.19 **Sorting on Two Columns with Descending Order**

List the rank, salary, name, and department of faculty. Order the result by ascending (alphabetic) rank and descending salary.

```
SELECT FacRank, FacSalary, FacFirstName, FacLastName, FacDept
  FROM Faculty
  ORDER BY FacRank, FacSalary DESC
```

FacRank	FacSalary	FacFirstName	FacLastName	FacDept
ASSC	75000.00	JULIA	MILLS	FIN
ASSC	70000.00	LEONARD	FIBON	MS
ASST	40000.00	CRISTOPHER	COLAN	MS
ASST	35000.00	LEONARD	VINCE	MS
PROF	120000.00	VICTORIA	EMMANUEL	MS
PROF	65000.00	NICKI	MACON	FIN

ORDER BY vs. DISTINCT
use the ORDER BY clause to sort a result table on one or more columns. Use the DISTINCT keyword to remove duplicates in the result.

Some students confuse ORDER BY and GROUP BY. In most systems, GROUP BY has the side effect of sorting by the grouping columns. You should not depend on this side effect. If you just want to sort, use ORDER BY rather than GROUP BY. If you want to sort and group, use both ORDER BY and GROUP BY.

Another way to improve the appearance of the result is to remove duplicate rows. By default, SQL does not remove duplicate rows. Duplicate rows are not possible when the primary keys of the result tables are included. There are a number of situations in which the primary key does not appear in the result. Example 4.21 demonstrates the DISTINCT keyword to remove duplicates that appear in the result of Example 4.20.

EXAMPLE 4.20 **Result with Duplicates**

List the city and state of faculty members.

```
SELECT FacCity, FacState
  FROM Faculty
```

FacCity	FacState
SEATTLE	WA
BOTHELL	WA
SEATTLE	WA
BELLEVUE	WA
SEATTLE	WA
SEATTLE	WA

EXAMPLE 4.21 **Eliminating Duplicates with DISTINCT**

List the unique city and state combinations in the *Faculty* table.

```
SELECT DISTINCT FacCity, FacState
  FROM Faculty
```

FacCity	FacState
BELLEVUE	WA
BOTHELL	WA
SEATTLE	WA

4.3 Conceptual Evaluation Process for SELECT Statements

conceptual evaluation process
the sequence of operations and intermediate tables used to derive the result of a SELECT statement. The conceptual evaluation process may help you gain an initial understanding of the SELECT statement as well as help you to understand more difficult problems.

To develop a clearer understanding of the SELECT statement, it is useful to understand the conceptual evaluation process or sequence of steps to produce the desired result. The conceptual evaluation process describes operations (mostly relational algebra operations) that produce intermediate tables leading to the result table. You may find it useful to refer to the conceptual evaluation process when first learning to write SELECT statements. After you gain initial competence with SELECT, you should not need to refer to the conceptual evaluation process except to gain insight about difficult problems.

To demonstrate the conceptual evaluation process, consider Example 4.22, which involves many parts of the SELECT statement. It involves multiple tables (*Enrollment* and *Offering* in the FROM clause), row conditions (following WHERE), aggregate functions (COUNT and AVG) over groups of rows (GROUP BY), a group condition (following HAVING), and sorting of the final result (ORDER BY).

EXAMPLE 4.22 (Access) **Depict Many Parts of the SELECT Statement**

List the course number, offer number, and average grade of students enrolled in fall 2005 IS course offerings in which more than one student is enrolled. Sort the result by course number in ascending order and average grade in descending order. The Oracle version of Example 4.22 is identical except for the % instead of the * as the wildcard character.

```
SELECT CourseNo, Offering.OfferNo, AVG(EnrGrade) AS AvgGrade
  FROM Enrollment, Offering
  WHERE CourseNo LIKE 'IS*' AND OffYear = 2005
        AND OffTerm = 'FALL'
        AND Enrollment.OfferNo = Offering.OfferNo
  GROUP BY CourseNo, Offering.OfferNo
  HAVING  COUNT(*) > 1
  ORDER BY CourseNo, 3 DESC1
```

In the ORDER BY clause, note the number 3 as the second column to sort. The number 3 means sort by the third column (*AvgGrade*) in SELECT. Some DBMSs do not allow aggregate expressions or alias names (*AvgGrade*) in the ORDER BY clause.

TABLE 4.10
Sample *Offering* Table

OfferNo	CourseNo	OffYear	OffTerm
1111	IS480	2005	FALL
2222	IS480	2005	FALL
3333	IS320	2005	FALL
5555	IS480	2006	WINTER
6666	IS320	2006	SPRING

TABLE 4.11
Sample *Enrollment* Table

StdSSN	OfferNo	EnrGrade
111-11-1111	1111	3.1
111-11-1111	2222	3.5
111-11-1111	3333	3.3
111-11-1111	5555	3.8
222-22-2222	1111	3.2
222-22-2222	2222	3.3
333-33-3333	1111	3.6

TABLE 4.12
Example 4.22 Result

CourseNo	OfferNo	AvgGrade
IS480	2222	3.4
IS480	1111	3.3

Tables 4.10 to 4.12 show the input tables and the result. Only small input and result tables have been used so that you can understand more clearly the process to derive the result. It does not take large tables to depict the conceptual evaluation process well.

The conceptual evaluation process is a sequence of operations as indicated in Figure 4.2. This process is conceptual rather than actual because most SQL compilers can produce the same output using many shortcuts. Because the shortcuts are system specific, rather mathematical, and performance oriented, we will not review them. The conceptual evaluation process provides a foundation for understanding the meaning of SQL statements that is independent of system and performance issues. The remainder of this section applies the conceptual evaluation process to Example 4.22.

1. The first step in the conceptual process combines the tables in the FROM clause with the cross product and join operators. In Example 4.22, a cross product operation is necessary because two tables are listed. A join operation is not necessary because the INNER JOIN keyword does not appear in the FROM statement. Recall that the cross product operator shows all possible rows by combining two tables. The resulting table contains the product of the number of rows and the sum of the columns. In this case, the cross product contains 35 rows (5 × 7) and 7 columns (3 + 4). Table 4.13 shows a partial result. As an exercise, you are encouraged to derive the entire result. As a notational shortcut here, the table name (abbreviated as *E* and *O*) is prefixed before the column name for *OfferNo*.
2. The second step uses a restriction operation to retrieve rows that satisfy the conditions in the WHERE clause from the result of step 1. We have four conditions: a join condition on *OfferNo,* a condition on *CourseNo,* a condition on *OffYear,* and a condition on *OffTerm*. Note that the condition on *CourseNo* includes the wildcard character (*). Any course

FIGURE 4.2
Flowchart of the Conceptual Evaluation Process

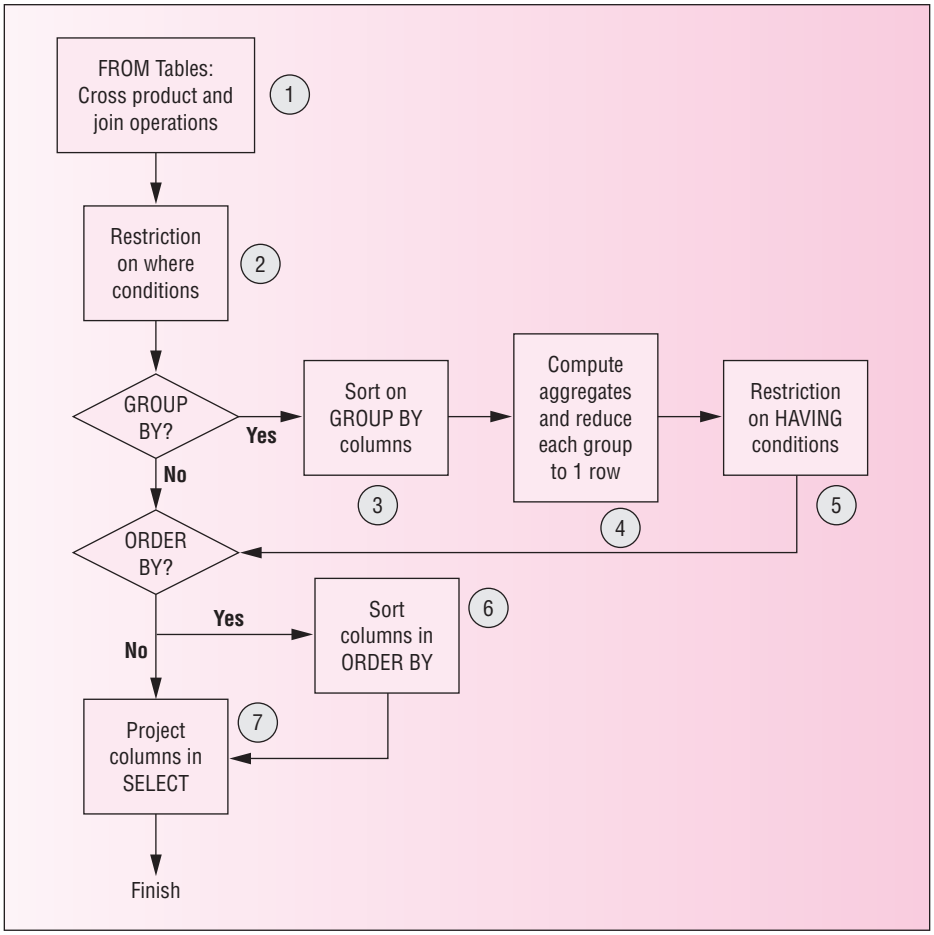


TABLE 4.13
Partial Result of Step 1 for First Two *Offering* Rows (1111 and 2222)

O.OfferNo	CourseNo	OffYear	OffTerm	StdSSN	E.OfferNo	EnrGrade
1111	IS480	2005	FALL	111-11-1111	1111	3.1
1111	IS480	2005	FALL	111-11-1111	2222	3.5
1111	IS480	2005	FALL	111-11-1111	3333	3.3
1111	IS480	2005	FALL	111-11-1111	5555	3.8
1111	IS480	2005	FALL	222-22-2222	1111	3.2
1111	IS480	2005	FALL	222-22-2222	2222	3.3
1111	IS480	2005	FALL	333-33-3333	1111	3.6
2222	IS480	2005	FALL	111-11-1111	1111	3.1
2222	IS480	2005	FALL	111-11-1111	2222	3.5
2222	IS480	2005	FALL	111-11-1111	3333	3.3
2222	IS480	2005	FALL	111-11-1111	5555	3.8
2222	IS480	2005	FALL	222-22-2222	1111	3.2
2222	IS480	2005	FALL	222-22-2222	2222	3.3
2222	IS480	2005	FALL	333-33-3333	1111	3.6

numbers beginning with IS match this condition. Table 4.14 shows that the result of the cross product (35 rows) is reduced to six rows.

3. The third step sorts the result of step 2 by the columns specified in the GROUP BY clause. The GROUP BY clause indicates that the output should relate to groups of rows

TABLE 4.14
Result of Step 2

O.OfferNo	CourseNo	OffYear	OffTerm	StdSSN	E.OfferNo	EnrGrade
1111	IS480	2005	FALL	111-11-1111	1111	3.1
2222	IS480	2005	FALL	111-11-1111	2222	3.5
1111	IS480	2005	FALL	222-22-2222	1111	3.2
2222	IS480	2005	FALL	222-22-2222	2222	3.3
1111	IS480	2005	FALL	333-33-3333	1111	3.6
3333	IS320	2005	FALL	111-11-1111	3333	3.3

TABLE 4.15
Result of Step 3

CourseNo	O.OfferNo	OffYear	OffTerm	StdSSN	E.OfferNo	EnrGrade
IS320	3333	2005	FALL	111-11-1111	3333	3.3
IS480	1111	2005	FALL	111-11-1111	1111	3.1
IS480	1111	2005	FALL	222-22-2222	1111	3.2
IS480	1111	2005	FALL	333-33-3333	1111	3.6
IS480	2222	2005	FALL	111-11-1111	2222	3.5
IS480	2222	2005	FALL	222-22-2222	2222	3.3

rather than individual rows. If the output relates to individual rows rather than groups of rows, the GROUP BY clause is omitted. When using the GROUP BY clause, you must include *every* column from the SELECT clause except for expressions that involve an aggregrate function.[4] Table 4.15 shows the result of step 2 sorted by *CourseNo* and *O.OfferNo*. Note that the columns have been rearranged to make the result easier to read.

4. The fourth step is only necessary if there is a GROUP BY clause. The fourth step computes aggregate function(s) for each group of rows and reduces each group to a single row. All rows in a group have the same values for the GROUP BY columns. In Table 4.16, there are three groups {<IS320,3333>, <IS480, 1111>, <IS480,2222>}. Computed columns are added for aggregate functions in the SELECT and HAVING clauses. Table 4.16 shows two new columns for the AVG function in the SELECT clause and the COUNT function in the HAVING clause. Note that remaining columns are eliminated at this point because they are not needed in the remaining steps.
5. The fifth step eliminates rows that do not satisfy the HAVING condition. Table 4.17 shows that the first row in Table 4.16 is removed because it fails the HAVING condition. Note that the HAVING clause specifies a restriction operation for groups of rows. The HAVING clause cannot be present without a preceding GROUP BY clause. The conditions in the HAVING clause always relate to groups of rows, not to individual rows. Typically, conditions in the HAVING clause involve aggregate functions.
6. The sixth step sorts the results according to the ORDER BY clause. Note that the ORDER BY clause is optional. Table 4.18 shows the result table after sorting.
7. The seventh step performs a final projection. Columns appearing in the result of step 6 are eliminated if they do not appear in the SELECT clause. Table 4.19 (identical to Table 4.12) shows the result after the projection of step 6. The *Count(*)* column is eliminated because it does not appear in SELECT. The seventh step (projection) occurs after the sixth step (sorting) because the ORDER BY clause can contain columns that do not appear in the SELECT list.

[4] In other words, when using the GROUP BY clause, every column in the SELECT clause should either be in the GROUP BY clause or be part of an expression with an aggregate function.

TABLE 4.16
Result of Step 4

CourseNo	O.OfferNo	AvgGrade	Count(*)
IS320	3333	3.3	1
IS480	1111	3.3	3
IS480	2222	3.4	2

TABLE 4.17
Result of Step 5

CourseNo	O.OfferNo	AvgGrade	Count(*)
IS480	1111	3.3	3
IS480	2222	3.4	2

TABLE 4.18
Result of Step 6

CourseNo	O.OfferNo	AvgGrade	Count(*)
IS480	2222	3.4	3
IS480	1111	3.3	2

TABLE 4.19
Result of Step 7

CourseNo	O.OfferNo	AvgGrade
IS480	2222	3.4
IS480	1111	3.3

This section finishes by discussing three major lessons about the conceptual evaluation process. These lessons are more important to remember than the specific details about the conceptual process.

- GROUP BY conceptually occurs after WHERE. If you have an error in a SELECT statement involving WHERE or GROUP BY, the problem is most likely in the WHERE clause. You can check the intermediate results after the WHERE clause by submitting a SELECT statement without the GROUP BY clause.
- Grouping occurs only one time in the evaluation process. If your problem involves more than one independent aggregate calculation, you may need more than one SELECT statement.
- Using sample tables can help you analyze difficult problems. It is often not necessary to go through the entire evaluation process. Rather, use sample tables to understand only the difficult part. Section 4.5 and Chapter 9 depict the use of sample tables to help analyze difficult problems.

4.4 Critical Questions for Query Formulation

critical questions for query formulation
provide a checklist to convert a problem statement into a database representation consisting of tables, columns, table connection operations, and row grouping requirements.

The conceptual evaluation process depicted in Figure 4.2 should help you understand the meaning of most SELECT statements, but it will probably not help you to formulate queries. Query formulation involves a conversion from a problem statement into a statement of a database language such as SQL as shown in Figure 4.3. In between the problem statement and the database language statement, you convert the problem statement into a database representation. Typically, the difficult part is to convert the problem statement into a database representation. This conversion involves a detailed knowledge of the tables and relationships and careful attention to possible ambiguities in the problem statement. The critical questions presented in this section provide a structured process to convert a problem statement into a database representation.

FIGURE 4.3
Query Formulation Process

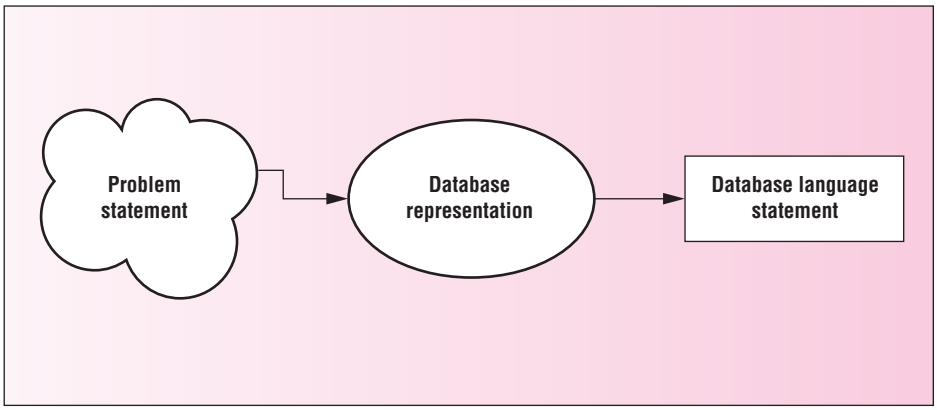


TABLE 4.20
Summary of Critical Questions for Query Formulation

Question	Analysis Tips
What tables are needed?	Match columns to output data requirements and conditions to test. If tables are not directly related, identify intermediate tables to provide a join path between tables.
How are the tables combined?	Most tables are combined using a primary key from a parent table to a foreign key of a child table. More difficult problems may involve other join conditions as well as other combining operators (outer join, difference, or division).
Does the output relate to individual rows or groups of rows?	Identify aggregate functions used in output data requirements and conditions to test. SQL statement requires a GROUP BY clause if aggregate functions are needed. A HAVING clause is needed if conditions use aggregate functions.

In converting from the problem statement into a database representation, you should answer three critical questions. Table 4.20 summarizes the analysis for the critical questions.

What tables are needed? For the first question, you should match data requirements to columns and tables. You should identify columns that are needed for output and conditions as well as intermediate tables needed to connect other tables. For example, if you want to join the *Student* and *Offering* tables, the *Enrollment* table should be included because it provides a connection to these tables. The *Student* and *Offering* tables cannot be combined directly. All tables needed in the query should be listed in the FROM clause.

How are the tables combined? For the second question, most tables are combined by a join operation. In Chapter 9, you will use the outer join, difference, and division operators to combine tables. For now, just concentrate on combining tables with joins. You need to identify the matching columns for each join. In most joins, the primary key of a parent table is matched with a foreign key of a related child table. Occasionally, the primary key of the parent table contains multiple columns. In this case, you need to match on both columns. In some situations, the matching columns do not involve a primary key/foreign key combination. You can perform a join as long as the matching columns have compatible data types. For example, when joining customer tables from different databases, there may not be a common primary key. Joining on other fields such as name, address, and so on may be necessary.

Does the output relate to individual rows or groups of rows? For the third question, look for computations involving aggregate functions in the problem statement. For example, the problem "list the name and average grade of students" contains an aggregate computation. Problems referencing an aggregate function indicate that the output relates to groups of rows. Hence the SQL statement requires a GROUP BY clause. If the problem contains

conditions with aggregate functions, a HAVING clause should accompany the GROUP BY clause. For example, the problem "list the offer number of course offerings with more than 30 students" needs a HAVING clause with a condition involving the count function.

After answering these questions, you are ready to convert the database representation into a database language statement. To help in this process, you should develop a collection of statements for each kind of relational algebra operator using a database that you understand well. For example, you should have statements for problems that involve join operations, joins with grouping, and joins with grouping conditions. As you increase your understanding of SQL, this conversion will become easy for most problems. For difficult problems such as those discussed in Section 4.5 and Chapter 9, relying on similar problems may be necessary because difficult problems are not common.

4.5 Refining Query Formulation Skills with Examples

Let's apply your query formulation skills and knowledge of the SELECT statement to more difficult problems. All problems in this section involve the parts of SELECT discussed in Sections 4.2 and 4.3. The problems involve more difficult aspects such as joining more than two tables, grouping after joins of several tables, joining a table to itself, and traditional set operators.

4.5.1 Joining Multiple Tables with the Cross Product Style

cross product style lists tables in the FROM clause and join conditions in the WHERE clause. The cross product style is easy to read but does not support outer join operations.

We begin with a number of join problems that are formulated using cross product operations in the FROM clause. This way to formulate joins is known as the cross product style because of the implied cross product operations. The next subsection uses join operations in the FROM clause to contrast the ways that joins can be expressed.

In Example 4.23, some student rows appear more than once in the result. For example, Roberto Morales appears twice. Because of the 1-M relationship between the *Student* and *Enrollment* tables, a *Student* row can match multiple *Enrollment* rows.

EXAMPLE 4.23 **Joining Two Tables**

List the student name, offering number, and grade of students who have a grade ≥ 3.5 in a course offering.

```
SELECT StdFirstName, StdLastName, OfferNo, EnrGrade
  FROM Student, Enrollment
  WHERE EnrGrade >= 3.5
      AND Student.StdSSN = Enrollment.StdSSN
```

StdFirstName	StdLastName	OfferNo	EnrGrade
CANDY	KENDALL	1234	3.5
MARIAH	DODGE	1234	3.8
HOMER	WELLS	4321	3.5
ROBERTO	MORALES	4321	3.5
BOB	NORBERT	5679	3.7
ROBERTO	MORALES	5679	3.8
MARIAH	DODGE	6666	3.6
LUKE	BRAZZI	7777	3.7
BOB	NORBERT	9876	3.5
WILLIAM	PILGRIM	9876	4

Examples 4.24 and 4.25 depict duplicate elimination after a join. In Example 4.24, some students appear more than once as in Example 4.23. Because only columns from the *Student* table are used in the output, duplicate rows appear. When you join a parent table to a child table and show only columns from the parent table in the result, duplicate rows can appear in the result. To eliminate duplicate rows, you can use the DISTINCT keyword as shown in Example 4.25.

EXAMPLE 4.24 **Join with Duplicates**

List the names of students who have a grade ≥ 3.5 in a course offering.

```
SELECT StdFirstName, StdLastName
  FROM Student, Enrollment
  WHERE EnrGrade >= 3.5
      AND Student.StdSSN = Enrollment.StdSSN
```

StdFirstName	StdLastName
CANDY	KENDALL
MARIAH	DODGE
HOMER	WELLS
ROBERTO	MORALES
BOB	NORBERT
ROBERTO	MORALES
MARIAH	DODGE
LUKE	BRAZZI
BOB	NORBERT
WILLIAM	PILGRIM

EXAMPLE 4.25 **Join with Duplicates Removed**

List the student names (without duplicates) that have a grade ≥ 3.5 in a course offering.

```
SELECT DISTINCT StdFirstName, StdLastName
  FROM Student, Enrollment
  WHERE EnrGrade >= 3.5
      AND Student.StdSSN = Enrollment.StdSSN
```

StdFirstName	StdLastName
BOB	NORBERT
CANDY	KENDALL
HOMER	WELLS
LUKE	BRAZZI
MARIAH	DODGE
ROBERTO	MORALES
WILLIAM	PILGRIM

Examples 4.26 through 4.29 depict problems involving more than two tables. In these problems, it is important to identify the tables in the FROM clause. Make sure that you examine conditions to test as well as columns in the result. In Example 4.28, the *Enrollment* table is needed even though it does not supply columns in the result or conditions to test.

EXAMPLE 4.26 **Joining Three Tables with Columns from Only Two Tables**

List the student name and the offering number in which the grade is greater than 3.7 and the offering is given in fall 2005.

```
SELECT StdFirstName, StdLastName, Enrollment.OfferNo
  FROM Student, Enrollment, Offering
  WHERE Student.StdSSN = Enrollment.StdSSN
       AND Offering.OfferNo = Enrollment.OfferNo
       AND OffYear = 2005 AND OffTerm = 'FALL'
       AND EnrGrade >= 3.7
```

StdFirstName	StdLastName	OfferNo
MARIAH	DODGE	1234

EXAMPLE 4.27 **Joining Three Tables with Columns from Only Two Tables**

List Leonard Vince's teaching schedule in fall 2005. For each course, list the offering number, course number, number of units, days, location, and time.

```
SELECT OfferNo, Offering.CourseNo, CrsUnits, OffDays, OffLocation, OffTime
  FROM Faculty, Course, Offering
  WHERE Faculty.FacSSN = Offering.FacSSN
       AND Offering.CourseNo = Course.CourseNo
       AND OffYear = 2005 AND OffTerm = 'FALL'
       AND FacFirstName = 'LEONARD'
       AND FacLastName = 'VINCE'
```

OfferNo	CourseNo	CrsUnits	OffDays	OffLocation	OffTime
1234	IS320	4	MW	BLM302	10:30 AM
4321	IS320	4	TTH	BLM214	3:30 PM

EXAMPLE 4.28 **Joining Four Tables**

List Bob Norbert's course schedule in spring 2006. For each course, list the offering number, course number, days, location, time, and faculty name.

```
SELECT Offering.OfferNo, Offering.CourseNo, OffDays, OffLocation,
            OffTime, FacFirstName, FacLastName
  FROM Faculty, Offering, Enrollment, Student
  WHERE Offering.OfferNo = Enrollment.OfferNo
       AND Student.StdSSN = Enrollment.StdSSN
       AND Faculty.FacSSN = Offering.FacSSN
       AND OffYear = 2006 AND OffTerm = 'SPRING'
       AND StdFirstName = 'BOB'
       AND StdLastName = 'NORBERT'
```

OfferNo	CourseNo	OffDays	OffLocation	OffTime	FacFirstName	FacLastName
5679	IS480	TTH	BLM412	3:30 PM	CRISTOPHER	COLAN
9876	IS460	TTH	BLM307	1:30 PM	LEONARD	FIBON

EXAMPLE 4.29 **Joining Five Tables**

List Bob Norbert's course schedule in spring 2006. For each course, list the offering number, course number, days, location, time, course units, and faculty name.

```
SELECT Offering.OfferNo, Offering.CourseNo, OffDays, OffLocation, OffTime,
          CrsUnits, FacFirstName, FacLastName
  FROM Faculty, Offering, Enrollment, Student, Course
  WHERE Faculty.FacSSN = Offering.FacSSN
      AND Offering.OfferNo = Enrollment.OfferNo
      AND Student.StdSSN = Enrollment.StdSSN
      AND Offering.CourseNo = Course.CourseNo
      AND OffYear = 2006 AND OffTerm = 'SPRING'
      AND StdFirstName = 'BOB'
      AND StdLastName = 'NORBERT'
```

OfferNo	CourseNo	OffDays	OffLocation	OffTime	CrsUnits	FacFirstName	FacLastName
5679	IS480	TTH	BLM412	3:30 PM	4	CRISTOPHER	COLAN
9876	IS460	TTH	BLM307	1:30 PM	4	LEONARD	FIBON

The *Enrollment* table is needed to connect the *Student* table with the *Offering* table. Example 4.29 extends Example 4.28 with details from the *Course* table. All five tables are needed to supply outputs, to test conditions, or to connect other tables.

Example 4.30 demonstrates another way to combine the *Student* and *Faculty* tables. In Example 4.28, you saw it was necessary to combine the *Student, Enrollment, Offering,* and *Faculty* tables to find faculty teaching a specified student. To find students who are on the faculty (perhaps teaching assistants), the tables can be joined directly. Combining the *Student* and *Faculty* tables in this way is similar to an intersection operation. However, intersection cannot actually be performed here because the *Student* and *Faculty* tables are not union compatible.

EXAMPLE 4.30 **Joining Two Tables without Matching on a Primary and Foreign Key**

List students who are on the faculty. Include all student columns in the result.

```
SELECT Student.*
  FROM Student, Faculty
  WHERE StdSSN = FacSSN
```

StdSSN	StdFirstName	StdLastName	StdCity	StdState	StdMajor	StdClass	StdGPA	StdZip
876-54-3210	CRISTOPHER	COLAN	SEATTLE	WA	IS	SR	4.00	98114-1332

join operator style lists join operations in the FROM clause using the INNER JOIN and ON keywords. The join operator style can be somewhat difficult to read for many join operations but it supports outer join operations as shown in Chapter 9.

A minor point about Example 4.30 is the use of the * after the SELECT keyword. Prefixing the * with a table name and period indicates all columns of the specified table are in the result. Using an * without a table name prefix indicates that all columns from all FROM tables are in the result.

4.5.2 Joining Multiple Tables with the Join Operator Style

As demonstrated in Section 4.2, join operations can be expressed directly in the FROM clause using the INNER JOIN and ON keywords. This join operator style can be used to combine any number of tables. To ensure that you are comfortable using this style, this

subsection presents examples of multiple table joins beginning with a two-table join in Example 4.31. Note that these examples do not execute in Oracle versions before 9i.

EXAMPLE 4.31 (Access and Oracle 9i versions and beyond)

Join Two Tables Using the Join Operator Style

Retrieve the name, city, and grade of students who have a high grade (greater than or equal to 3.5) in a course offering.

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
  FROM Student INNER JOIN Enrollment
      ON Student.StdSSN = Enrollment.StdSSN
  WHERE EnrGrade >= 3.5
```

StdFirstName	StdLastName	StdCity	EnrGrade
CANDY	KENDALL	TACOMA	3.5
MARIAH	DODGE	SEATTLE	3.8
HOMER	WELLS	SEATTLE	3.5
ROBERTO	MORALES	SEATTLE	3.5
BOB	NORBERT	BOTHELL	3.7
ROBERTO	MORALES	SEATTLE	3.8
MARIAH	DODGE	SEATTLE	3.6
LUKE	BRAZZI	SEATTLE	3.7
BOB	NORBERT	BOTHELL	3.5
WILLIAM	PILGRIM	BOTHELL	4

The join operator style can be extended for any number of tables. Think of the join operator style as writing a complicated formula with lots of parentheses. To add another part to the formula, you need to add the arguments, operator, and another level of parentheses. For example, with the formula (X + Y) * Z, you can add another operation as ((X + Y) * Z) / W. This same principle can be applied with the join operator style. Examples 4.32 and 4.33 extend Example 4.31 with additional conditions that need other tables. In both examples, another INNER JOIN is added to the end of the previous INNER JOIN operations. The INNER JOIN could also have been added at the beginning or middle if desired. The ordering of INNER JOIN operations is not important.

EXAMPLE 4.32 (Access and Oracle 9i versions and beyond)

Join Three Tables Using the Join Operator Style

Retrieve the name, city, and grade of students who have a high grade (greater than or equal 3.5) in a course offered in fall 2005.

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
  FROM ( Student INNER JOIN Enrollment
         ON Student.StdSSN = Enrollment.StdSSN )
    INNER JOIN Offering
      ON Offering.OfferNo = Enrollment.OfferNo
 WHERE EnrGrade >= 3.5 AND OffTerm = 'FALL'
     AND OffYear = 2005
```

StdFirstName	StdLastName	StdCity	EnrGrade
CANDY	KENDALL	TACOMA	3.5
MARIAH	DODGE	SEATTLE	3.8
HOMER	WELLS	SEATTLE	3.5
ROBERTO	MORALES	SEATTLE	3.5

EXAMPLE 4.33 (Access and Oracle 9i versions and beyond)

Join Four Tables Using the Join Operator Style

Retrieve the name, city, and grade of students who have a high grade (greater than or equal to 3.5) in a course offered in fall 2005 taught by Leonard Vince.

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
  FROM ( (Student INNER JOIN Enrollment
            ON Student.StdSSN = Enrollment.StdSSN )
    INNER JOIN Offering
       ON Offering.OfferNo = Enrollment.OfferNo )
    INNER JOIN Faculty ON Faculty.FacSSN = Offering.FacSSN
  WHERE EnrGrade >= 3.5 AND OffTerm = 'FALL'
      AND OffYear = 2005 AND FacFirstName = 'LEONARD'
      AND FacLastName = 'VINCE'
```

StdFirstName	StdLastName	StdCity	EnrGrade
CANDY	KENDALL	TACOMA	3.5
MARIAH	DODGE	SEATTLE	3.8
HOMER	WELLS	SEATTLE	3.5
ROBERTO	MORALES	SEATTLE	3.5

The cross product and join operator styles can be mixed as demonstrated in Example 4.34. In most cases, it is preferable to use one style or the other, however.

EXAMPLE 4.34 (Access and Oracle 9i versions and beyond)

Combine the Cross Product and Join Operator Styles

Retrieve the name, city, and grade of students who have a high grade (greater than or equal to 3.5) in a course offered in fall 2005 taught by Leonard Vince (same result as Example 4.33).

```
SELECT StdFirstName, StdLastName, StdCity, EnrGrade
  FROM ( (Student INNER JOIN Enrollment
       ON Student.StdSSN = Enrollment.StdSSN )
    INNER JOIN Offering
       ON Offering.OfferNo = Enrollment.OfferNo ),
      Faculty
  WHERE EnrGrade >= 3.5 AND OffTerm = 'FALL'
      AND OffYear = 2005 AND FacFirstName = 'LEONARD'
      AND FacLastName = 'VINCE'
      AND Faculty.FacSSN = Offering.FacSSN
```

The choice between the cross product and the join operator styles is largely a matter of preference. In the cross product style, it is easy to see the tables in the SQL statement. For multiple joins, the join operator style can be difficult to read because of nested parentheses. The primary advantage of the join operator style is that you can formulate queries involving outer joins as described in Chapter 9.

You should be comfortable reading both join styles even if you only write SQL statements using one style. You may need to maintain statements written with both styles. In addition, some visual query languages generate code in one of the styles. For example, Query Design, the visual query language of Microsoft Access, generates code in the join operator style.

4.5.3 Self-Joins and Multiple Joins between Two Tables

self-join
a join between a table and itself (two copies of the same table). Self-joins are useful for finding relationships among rows of the same table.

Example 4.35 demonstrates a self-join, a join involving a table with itself. A self-join is necessary to find relationships among rows of the same table. The foreign key, *FacSupervisor,* shows relationships among *Faculty* rows. To find the supervisor name of a faculty member, match on the *FacSupervisor* column with the *FacSSN* column. The trick is to imagine that you are working with two copies of the *Faculty* table. One copy plays the role of the subordinate, while the other copy plays the role of the superior. In SQL, a self-join requires alias names (*Subr* and *Supr*) in the FROM clause to distinguish between the two roles or copies.

EXAMPLE 4.35 **Self-join**

List faculty members who have a higher salary than their supervisor. List the Social Security number, name, and salary of the faculty and supervisor.

```
SELECT Subr.FacSSN, Subr.FacLastName, Subr.FacSalary, Supr.FacSSN,
       Supr.FacLastName, Supr.FacSalary
  FROM Faculty Subr, Faculty Supr
  WHERE Subr.FacSupervisor = Supr.FacSSN
    AND Subr.FacSalary > Supr.FacSalary
```

Subr.FacSSN	Subr.FacLastName	Subr.FacSalary	Supr.FacSSN	Supr.FacLastName	Supr.FacSalary
987-65-4321	MILLS	75000.00	765-43-2109	MACON	65000.00

Problems involving self-joins can be difficult to understand. If you are having trouble understanding Example 4.35, use the conceptual evaluation process to help. Start with a small *Faculty* table. Copy this table and use the names *Subr* and *Supr* to distinguish between the two copies. Join the two tables over *Subr.FacSupervisor* and *Supr.FacSSN*. If you need, derive the join using a cross product operation. You should be able to see that each result row in the join shows a subordinate and supervisor pair.

Problems involving self-referencing (unary) relationships are part of tree-structured queries. In tree-structured queries, a table can be visualized as a structure such as a tree or hierarchy. For example, the *Faculty* table has a structure showing an organization hierarchy. At the top, the college dean resides. At the bottom, faculty members without

subordinates reside. Similar structures apply to the chart of accounts in accounting systems, part structures in manufacturing systems, and route networks in transportation systems.

A more difficult problem than a self-join is to find all subordinates (direct or indirect) in an organization hierarchy. This problem can be solved in SQL if the number of subordinate levels is known. One join for each subordinate level is needed. Without knowing the number of subordinate levels, this problem cannot be done in SQL-92 although it can be solved in SQL:2003 using the WITH RECURSIVE clause and in proprietary SQL extensions. In SQL-92, tree-structured queries can be solved by using SQL inside a programming language.

Example 4.36 shows another difficult join problem. This problem involves two joins between the same two tables (*Offering* and *Faculty*). Alias table names (*O1* and *O2*) are needed to distinguish between the two copies of the *Offering* table used in the statement.

EXAMPLE 4.36 **More Than One Join between Tables Using Alias Table Names**

List the names of faculty members and the course number for which the faculty member teaches the same course number as his or her supervisor in 2006.

```
SELECT FacFirstName, FacLastName, O1.CourseNo
  FROM Faculty, Offering O1, Offering O2
  WHERE Faculty.FacSSN = O1.FacSSN
      AND Faculty.FacSupervisor = O2.FacSSN
      AND O1.OffYear = 2006 AND O2.OffYear = 2006
      AND O1.CourseNo = O2.CourseNo
```

FacFirstName	FacLastName	CourseNo
LEONARD	VINCE	IS320
LEONARD	FIBON	IS320

If this problem is too difficult, use the conceptual evaluation process (Figure 4.2) with sample tables to gain insight. Perform a join between the sample *Faculty* and *Offering* tables, then join this result to another copy of *Offering* (*O2*) matching *FacSupervisor* with *O2.FacSSN*. In the resulting table, select the rows that have matching course numbers and year equal to 2006.

4.5.4 Combining Joins and Grouping

Example 4.37 demonstrates why it is sometimes necessary to group on multiple columns. After studying Example 4.37, you might be confused about the necessity to group on both *OfferNo* and *CourseNo*. One simple explanation is that any columns appearing in SELECT must be either a grouping column or an aggregate expression. However, this explanation does not quite tell the entire story. Grouping on *OfferNo* alone produces the same values for the computed column (*NumStudents*) because *OfferNo* is the primary key. Including nonunique columns such as *CourseNo* adds information to each result row but does not change the aggregate calculations. If you do not understand this point, use sample tables to demonstrate it. When evaluating your sample tables, remember that joins occur before grouping as indicated in the conceptual evaluation process.

EXAMPLE 4.37 **Join with Grouping on Multiple Columns**

List the course number, the offering number, and the number of students enrolled. Only include courses offered in spring 2006.

```
SELECT CourseNo, Enrollment.OfferNo, Count(*) AS NumStudents
  FROM Offering, Enrollment
  WHERE Offering.OfferNo = Enrollment.OfferNo
      AND OffYear = 2006 AND OffTerm = 'SPRING'
  GROUP BY Enrollment.OfferNo, CourseNo
```

CourseNo	OfferNo	NumStudents
FIN480	7777	3
IS460	9876	7
IS480	5679	6

Example 4.38 demonstrates another problem involving joins and grouping. An important part of this problem is the need for the *Student* table and the HAVING condition. They are needed because the problem statement refers to an aggregate function involving the *Student* table.

EXAMPLE 4.38 **Joins, Grouping, and Group Conditions**

List the course number, the offering number, and the average GPA of students enrolled. Only include courses offered in fall 2005 in which the average GPA of enrolled students is greater than 3.0.

```
SELECT CourseNo, Enrollment.OfferNo, Avg(StdGPA) AS AvgGPA
  FROM Student, Offering, Enrollment
  WHERE Offering.OfferNo = Enrollment.OfferNo
      AND Enrollment.StdSSN = Student.StdSSN
      AND OffYear = 2005 AND OffTerm = 'FALL'
  GROUP BY CourseNo, Enrollment.OfferNo
  HAVING Avg(StdGPA) > 3.0
```

CourseNo	OfferNo	AvgGPA
IS320	1234	3.23333330949148
IS320	4321	3.03333334128062

4.5.5 Traditional Set Operators in SQL

In SQL, you can directly use the traditional set operators with the UNION, INTERSECT, and EXCEPT keywords. Some DBMSs including Microsoft Access do not support the INTERSECT and EXCEPT keywords. As with relational algebra, the problem is always to make sure that the tables are union compatible. In SQL, you can use a SELECT statement to make tables compatible by listing only compatible columns. Examples 4.39 through 4.41 demonstrate set operations on column subsets of the *Faculty* and *Student* tables. The columns have been renamed to avoid confusion.

EXAMPLE 4.39 **UNION Query**

Show all faculty and students. Only show the common columns in the result.

```
SELECT FacSSN AS SSN, FacFirstName AS FirstName, FacLastName AS
          LastName, FacCity AS City, FacState AS State
  FROM Faculty
     UNION
SELECT StdSSN AS SSN, StdFirstName AS FirstName, StdLastName AS
          LastName, StdCity AS City, StdState AS State
  FROM Student
```

SSN	FirstName	LastName	City	State
098765432	LEONARD	VINCE	SEATTLE	WA
123456789	HOMER	WELLS	SEATTLE	WA
124567890	BOB	NORBERT	BOTHELL	WA
234567890	CANDY	KENDALL	TACOMA	WA
345678901	WALLY	KENDALL	SEATTLE	WA
456789012	JOE	ESTRADA	SEATTLE	WA
543210987	VICTORIA	EMMANUEL	BOTHELL	WA
567890123	MARIAH	DODGE	SEATTLE	WA
654321098	LEONARD	FIBON	SEATTLE	WA
678901234	TESS	DODGE	REDMOND	WA
765432109	NICKI	MACON	BELLEVUE	WA
789012345	ROBERTO	MORALES	SEATTLE	WA
876543210	CRISTOPHER	COLAN	SEATTLE	WA
890123456	LUKE	BRAZZI	SEATTLE	WA
901234567	WILLIAM	PILGRIM	BOTHELL	WA
987654321	JULIA	MILLS	SEATTLE	WA

EXAMPLE 4.40 (Oracle) **INTERSECT Query**

Show teaching assistants, faculty who are students. Only show the common columns in the result.

```
SELECT FacSSN AS SSN, FacFirstName AS FirstName, FacLastName AS
          LastName, FacCity AS City, FacState AS State
  FROM Faculty
     INTERSECT
SELECT StdSSN AS SSN, StdFirstName AS FirstName,
          StdLastName AS LastName, StdCity AS City,
          StdState AS State
  FROM Student
```

SSN	FirstName	LastName	City	State
876543210	CRISTOPHER	COLAN	SEATTLE	WA

EXAMPLE 4.41 (Oracle)

Difference Query

Show faculty who are *not* students (pure faculty). Only show the common columns in the result. Oracle uses the MINUS keyword instead of the EXCEPT keyword used in SQL:2003.

```
SELECT FacSSN AS SSN, FacFirstName AS FirstName, FacLastName AS
        LastName, FacCity AS City, FacState AS State
  FROM Faculty
        MINUS
SELECT StdSSN AS SSN, StdFirstName AS FirstName, StdLastName AS
        LastName, StdCity AS City, StdState AS State
  FROM Student
```

SSN	FirstName	LastName	City	State
098765432	LEONARD	VINCE	SEATTLE	WA
543210987	VICTORIA	EMMANUEL	BOTHELL	WA
654321098	LEONARD	FIBON	SEATTLE	WA
765432109	NICKI	MACON	BELLEVUE	WA
987654321	JULIA	MILLS	SEATTLE	WA

By default, duplicate rows are removed in the results of SQL statements with the UNION, INTERSECT, and EXCEPT (MINUS) keywords. If you want to retain duplicate rows, use the ALL keyword after the operator. For example, the UNION ALL keyword performs a union operation but does not remove duplicate rows.

4.6 SQL Modification Statements

The modification statements support entering new rows (INSERT), changing columns in one or more rows (UPDATE), and deleting one or more rows (DELETE). Although well designed and powerful, they are not as widely used as the SELECT statement because data entry forms are easier to use for end users.

The INSERT statement has two formats as demonstrated in Examples 4.42 and 4.43. In the first format, one row at a time can be added. You specify values for each column with the VALUES clause. You must format the constant values appropriate for each column. Refer to the documentation of your DBMS for details about specifying constants, especially string and date constants. Specifying a null value for a column is also not standard across DBMSs. In some systems, you simply omit the column name and the value. In other systems, you use a particular symbol for a null value. Of course, you must be careful that the table definition permits null values for the column of interest. Otherwise, the INSERT statement will be rejected.

EXAMPLE 4.42

Single Row Insert

Insert a row into the *Student* table supplying values for all columns.

```
INSERT INTO Student
   (StdSSN, StdFirstName, StdLastName, StdCity, StdState, StdZip, StdClass,
StdMajor, StdGPA)
  VALUES ('999999999', 'JOE', 'STUDENT', 'SEATAC', 'WA', '98042-1121', 'FR',
          'IS', 0.0)
```

The second format of the INSERT statement supports addition of a set of records as shown in Example 4.43. Using the SELECT statement inside the INSERT statement, you can specify any derived set of rows. You can use the second format when you want to create temporary tables for specialized processing.

EXAMPLE 4.43 **Multiple Row Insert**

Assume a new table *ISStudent* has been previously created. *ISStudent* has the same columns as *Student.* This INSERT statement adds rows from *Student* into *ISStudent.*

```
INSERT INTO ISStudent
  SELECT  *  FROM Student WHERE StdMajor = 'IS'
```

The UPDATE statement allows one or more rows to be changed, as shown in Examples 4.44 and 4.45. Any number of columns can be changed, although typically only one column at a time is changed. When changing the primary key, update rules on referenced rows may not allow the operation.

EXAMPLE 4.44 **Single Column Update**

Give faculty members in the MS department a 10 percent raise. Four rows are updated.

```
UPDATE Faculty
  SET FacSalary = FacSalary  *  1.1
  WHERE FacDept = 'MS'
```

EXAMPLE 4.45 **Update Multiple Columns**

Change the major and class of Homer Wells. One row is updated.

```
UPDATE Student
  SET StdMajor = 'ACCT', StdClass = 'SO'
  WHERE StdFirstName = 'HOMER'
      AND StdLastName = 'WELLS'
```

The DELETE statement allows one or more rows to be removed, as shown in Examples 4.46 and 4.47. DELETE is subject to the rules on referenced rows. For example, a *Student* row cannot be deleted if related *Enrollment* rows exist and the deletion action is restrict.

EXAMPLE 4.46 **Delete Selected Rows**

Delete all IS majors who are seniors. Three rows are deleted.

```
DELETE FROM Student
  WHERE StdMajor = 'IS' AND StdClass = 'SR'
```

EXAMPLE 4.47 **Delete All Rows in a Table**

Delete all rows in the *ISStudent* table. This example assumes that the *ISStudent* table has been previously created.

```
DELETE FROM ISStudent
```

Sometimes it is useful for the condition inside the WHERE clause of the DELETE statement to reference rows from other tables. Microsoft Access supports the join operator style to combine tables as shown in Example 4.48. You *cannot* use the cross product style inside a DELETE statement. Chapter 9 shows another way to reference other tables in a DELETE statement that most DBMSs (including Access and Oracle) support.

EXAMPLE 4.48 (Access) **DELETE Statement Using the Join Operator Style**

Delete offerings taught by Leonard Vince. Three Offering rows are deleted. In addition, this statement deletes related rows in the Enrollment table because the ON DELETE clause is set to CASCADE.

```
DELETE Offering.*
  FROM Offering INNER JOIN Faculty
      ON Offering.FacSSN = Faculty.FacSSN
  WHERE FacFirstName = 'LEONARD'
       AND FacLastName = 'VINCE'
```

Closing Thoughts

Chapter 4 has introduced the fundamental statements of the industry standard Structured Query Language (SQL). SQL has a wide scope covering database definition, manipulation, and control. As a result of careful analysis and compromise, standards groups have produced a well-designed language. SQL has become the common glue that binds the database industry even though strict conformance to the standard is sometimes lacking. You will no doubt continually encounter SQL throughout your career.

This chapter has focused on the most widely used parts of the SELECT statement from the core part of the SQL:2003 standard. Numerous examples were shown to demonstrate conditions on different data types, complex logical expressions, multiple table joins, summarization of tables with GROUP BY and HAVING, sorting of tables, and the traditional set operators. To facilitate hands-on usage of SQL, examples were shown for both Oracle and Access with special attention to deviations from the SQL:2003 standard. This chapter also briefly described the modification statements INSERT, UPDATE, and DELETE. These statements are not as complex and widely used as SELECT.

This chapter has emphasized two problem-solving guidelines to help you formulate queries. The conceptual evaluation process was presented to demonstrate derivation of result rows for SELECT statements involving joins and grouping. You may find this evaluation process helps in your initial learning of SELECT as well as provides insight on more challenging problems. To help formulate queries, three questions were provided to guide you. You should explicitly or implicitly answer these questions before writing a SELECT

statement to solve a problem. An understanding of both the critical questions and the conceptual evaluation process will provide you a solid foundation for using relational databases. Even with these formulation aids, you need to work many problems to learn query formulation and the SELECT statement.

This chapter covered an important subset of the SELECT statement. Other parts of the SELECT statement not covered in this chapter are outer joins, nested queries, and division problems. Chapter 9 covers advanced query formulation and additional parts of the SELECT statement so that you can hone your skills.

Review Concepts

- SQL consists of statements for database definition (CREATE TABLE, ALTER TABLE, etc.), database manipulation (SELECT, INSERT, UPDATE, and DELETE), and database control (GRANT, REVOKE, etc.).
- The most recent SQL standard is known as SQL:2003. Major DBMS vendors support most features in the core part of this standard although the lack of independent conformance testing hinders strict conformance with the standard.
- SELECT is a complex statement. Chapter 4 covered SELECT statements with the format:

```
SELECT <list of column and column expressions>
  FROM <list of tables and join operations>
  WHERE <list of row conditions connected by AND, OR, and NOT>
  GROUP BY <list of columns>
  HAVING <list of group conditions connected by AND, OR, and NOT>
  ORDER BY <list of sorting specifications>
```

- Use the standard comparison operators to select rows:

```
SELECT StdFirstName, StdLastName, StdCity, StdGPA
  FROM Student
  WHERE StdGPA >= 3.7
```

- Inexact matching is done with the LIKE operator and pattern-matching characters:

  Oracle and SQL:2003

```
SELECT CourseNo, CrsDesc
  FROM Course
  WHERE CourseNo LIKE 'IS4%'
```

  Access

```
SELECT CourseNo, CrsDesc
  FROM Course
  WHERE CourseNo LIKE 'IS4*'
```

- Use BETWEEN . . . AND to compare dates:

  Oracle

```
SELECT FacFirstName, FacLastName, FacHireDate
  FROM Faculty
  WHERE FacHireDate BETWEEN '1-Jan-1999' AND '31-Dec-2000'
```

Access:

```
SELECT FacFirstName, FacLastName, FacHireDate
  FROM Faculty
  WHERE FacHireDate BETWEEN #1/1/1999# AND #12/31/2000#
```

- Use expressions in the SELECT column list and WHERE clause:

  Oracle

```
SELECT FacFirstName, FacLastName, FacCity, FacSalary*1.1 AS
            InflatedSalary, FacHireDate
  FROM Faculty
  WHERE to_number(to_char(FacHireDate, 'YYYY' ) ) > 1999
```

  Access

```
SELECT FacFirstName, FacLastName, FacCity, FacSalary*1.1 AS
            InflatedSalary, FacHireDate
  FROM Faculty
  WHERE year(FacHireDate) > 1999
```

- Test for null values:

```
SELECT OfferNo, CourseNo
  FROM Offering
  WHERE FacSSN IS NULL AND OffTerm = 'SUMMER'
       AND OffYear = 2006
```

- Create complex logical expressions with AND and OR:

```
SELECT OfferNo, CourseNo, FacSSN
  FROM Offering
  WHERE (OffTerm = 'FALL' AND OffYear = 2005)
       OR (OffTerm = 'WINTER' AND OffYear = 2006)
```

- Sort results with the ORDER BY clause:

```
SELECT StdGPA, StdFirstName, StdLastName, StdCity, StdState
  FROM Student
  WHERE StdClass = 'JR'
  ORDER BY StdGPA
```

- Eliminate duplicates with the DISTINCT keyword:

```
SELECT DISTINCT FacCity, FacState
  FROM Faculty
```

- Qualify column names in join queries:

```
SELECT Course.CourseNo, CrsDesc
  FROM Offering, Course
  WHERE OffTerm = 'SPRING' AND OffYear = 2006
      AND Course.CourseNo = Offering.CourseNo
```

- Use the GROUP BY clause to summarize rows:

```
SELECT StdMajor, AVG(StdGPA) AS AvgGpa
  FROM Student
  GROUP BY StdMajor
```

- GROUP BY must precede HAVING:

```
SELECT StdMajor, AVG(StdGPA) AS AvgGpa
  FROM Student
  GROUP BY StdMajor
  HAVING AVG(StdGPA) > 3.1
```

- Use WHERE to test row conditions and HAVING to test group conditions:

```
SELECT StdMajor, AVG(StdGPA) AS AvgGpa
  FROM Student
  WHERE StdClass IN ('JR', 'SR')
  GROUP BY StdMajor
  HAVING AVG(StdGPA) > 3.1
```

- Difference between COUNT(*) and COUNT(DISTINCT column)—not supported by Access:

```
SELECT OffYear, COUNT(*) AS NumOfferings, COUNT(DISTINCT CourseNo)
          AS NumCourses
  FROM Offering
  GROUP BY OffYear
```

- Conceptual evaluation process lessons: use small sample tables, GROUP BY occurs after WHERE, only one grouping per SELECT statement.
- Query formulation questions: what tables?, how combined?, and row or group output?
- Joining more than two tables with the cross product and join operator styles (not supported by Oracle versions before 9i):

```
SELECT OfferNo, Offering.CourseNo, CrsUnits, OffDays, OffLocation,
          OffTime
  FROM Faculty, Course, Offering
  WHERE Faculty.FacSSN = Offering.FacSSN
       AND Offering.CourseNo = Course.CourseNo
       AND OffYear = 2005 AND OffTerm = 'FALL'
       AND FacFirstName = 'LEONARD'
       AND FacLastName = 'VINCE'
SELECT OfferNo, Offering.CourseNo, CrsUnits, OffDays, OffLocation, OffTime
  FROM ( Faculty INNER JOIN Offering
           ON Faculty.FacSSN = Offering.FacSSN )
       INNER JOIN Course
           ON Offering.CourseNo = Course.CourseNo
  WHERE OffYear = 2005 AND OffTerm = 'FALL'
       AND FacFirstName = 'LEONARD'
       AND FacLastName = 'VINCE'
```

- Self-joins:

```
SELECT Subr.FacSSN, Subr.FacLastName, Subr.FacSalary,
             Supr.FacSSN, Supr.FacLastName, Supr.FacSalary
  FROM Faculty Subr, Faculty Supr
  WHERE Subr.FacSupervisor = Supr.FacSSN
       AND Subr.FacSalary > Supr.FacSalary
```

- Combine joins and grouping:

```
SELECT CourseNo, Enrollment.OfferNo, Count(*) AS NumStudents
  FROM Offering, Enrollment
  WHERE Offering.OfferNo = Enrollment.OfferNo
     AND OffYear = 2006 AND OffTerm = 'SPRING'
  GROUP BY Enrollment.OfferNo, CourseNo
```

- Traditional set operators and union compatibility:

```
SELECT FacSSN AS SSN, FacLastName AS LastName FacCity AS City,
            FacState AS State
  FROM Faculty
     UNION
SELECT StdSSN AS SSN, StdLastName AS LastName, StdCity AS City,
            StdState AS State
  FROM Student
```

- Use the INSERT statement to add one or more rows:

```
INSERT INTO Student
 (StdSSN, StdFirstName, StdLastName, StdCity, StdState, StdClass,
  StdMajor, StdGPA)
 VALUES ('999999999', 'JOE', 'STUDENT', 'SEATAC', 'WA', 'FR', 'IS', 0.0)
```

- Use the UPDATE statement to change columns in one or more rows:

```
UPDATE Faculty
  SET FacSalary = FacSalary * 1.1
  WHERE FacDept = 'MS'
```

- Use the DELETE statement to remove one or more rows:

```
DELETE FROM Student
  WHERE StdMajor = 'IS' AND StdClass = 'SR'
```

- Use a join operation inside a DELETE statement (Access only):

```
DELETE Offering.*
  FROM Offering INNER JOIN Faculty
     ON Offering.FacSSN = Faculty.FacSSN
  WHERE FacFirstName = 'LEONARD' AND FacLastName = 'VINCE'
```

Questions

1. Why do some people pronounce SQL as "sequel"?
2. Why are the manipulation statements of SQL more widely used than the definition and control statements?
3. How many levels do the SQL-92, SQL:1999, and SQL:2003 standards have?
4. Why is conformance testing important for the SQL standard?
5. In general, what is the state of conformance among major DBMS vendors for the SQL:2003 standard?
6. What is stand-alone SQL?
7. What is embedded SQL?
8. What is an expression in the context of database languages?
9. From the examples and the discussion in Chapter 4, what parts of the SELECT statement are not supported by all DBMSs?

10. Recite the rule about the GROUP BY and HAVING clauses.
11. Recite the rule about columns in SELECT when a GROUP BY clause is used.
12. How does a row condition differ from a group condition?
13. Why should row conditions be placed in the WHERE clause rather than the HAVING clause?
14. Why are most DBMSs not case sensitive when matching on string conditions?
15. Explain how working with sample tables can provide insight about difficult problems.
16. When working with date columns, why is it necessary to refer to documentation of your DBMS?
17. How do exact and inexact matching differ in SQL?
18. How do you know when the output of a query relates to groups of rows as opposed to individual rows?
19. What tables belong in the FROM statement?
20. Explain the cross product style for join operations.
21. Explain the join operator style for join operations.
22. Discuss the pros and cons of the cross product versus the join operator styles. Do you need to know both the cross product and the join operator styles?
23. What is a self-join? When is a self-join useful?
24. Provide a SELECT statement example in which a table is needed even though the table does not provide conditions to test or columns to show in the result.
25. What is the requirement when using the traditional set operators in a SELECT statement?
26. When combining joins and grouping, what conceptually occurs first, joins or grouping?
27. How many times does grouping occur in a SELECT statement?
28. Why is the SELECT statement more widely used than the modification statements INSERT, UPDATE, and DELETE?
29. Provide an example of an INSERT statement that can insert multiple rows.
30. What is the relationship between the DELETE statement and the rules about deleting referenced rows?
31. What is the relationship between the UPDATE statement and the rules about updating the primary key of referenced rows?
32. How does COUNT(*) differ from COUNT(ColumnName)?
33. How does COUNT(DISTINCT ColumnName) differ from COUNT(ColumnName)?
34. When mixing AND and OR in a logical expression, why is it a good idea to use parentheses?
35. What are the most important lessons about the conceptual evaluation process?
36. What are the mental steps involved in query formulation?
37. What kind of join queries often have duplicates in the result?
38. What mental steps in the query formulation process are addressed by the conceptual evaluation process and critical questions?

Problems

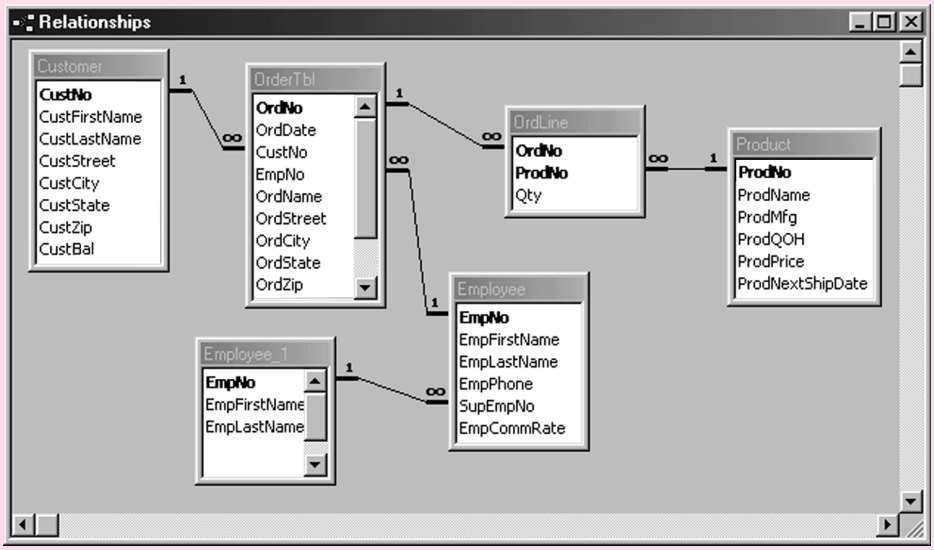
The problems use the tables of the Order Entry database, an extension of the order entry tables used in the problems of Chapter 3. Table 4.P1 lists the meaning of each table and Figure 4.P1 shows the Access Relationship window. After the relationship diagram, row listings and Oracle CREATE TABLE statements are shown for each table. In addition to the other documentation, here are some notes about the Order Entry Database:

- The primary key of the *OrdLine* table is a combination of *OrdNo* and *ProdNo*.
- The *Employee* table has a self-referencing (unary) relationship to itself through the foreign key, *SupEmpNo,* the employee number of the supervising employee. In the relationship diagram, the table *Employee_1* is a representation of the self-referencing relationship, not a real table.
- The relationship from *OrderTbl* to *OrdLine* cascades deletions and primary key updates of referenced rows. All other relationships restrict deletions and primary key updates of referenced rows if related rows exist.

TABLE 4.P1
Tables of the Order Entry Database

Table Name	Description
Customer	List of customers who have placed orders
OrderTbl	Contains the heading part of an order; Internet orders do not have an employee
Employee	List of employees who can take orders
OrdLine	Contains the detail part of an order
Product	List of products that may be ordered

FIGURE 4.P1
Relationship Window for the Order Entry Database



Customer

CustNo	CustFirstName	CustLastName	CustStreet	CustCity	CustState	CustZip	CustBal
C0954327	Sheri	Gordon	336 Hill St.	Littleton	CO	80129-5543	$230.00
C1010398	Jim	Glussman	1432 E. Ravenna	Denver	CO	80111-0033	$200.00
C2388597	Beth	Taylor	2396 Rafter Rd	Seattle	WA	98103-1121	$500.00
C3340959	Betty	Wise	4334 153rd NW	Seattle	WA	98178-3311	$200.00
C3499503	Bob	Mann	1190 Lorraine Cir.	Monroe	WA	98013-1095	$0.00
C8543321	Ron	Thompson	789 122nd St.	Renton	WA	98666-1289	$85.00
C8574932	Wally	Jones	411 Webber Ave.	Seattle	WA	98105-1093	$1,500.00
C8654390	Candy	Kendall	456 Pine St.	Seattle	WA	98105-3345	$50.00
C9128574	Jerry	Wyatt	16212 123rd Ct.	Denver	CO	80222-0022	$100.00
C9403348	Mike	Boren	642 Crest Ave.	Englewood	CO	80113-5431	$0.00
C9432910	Larry	Styles	9825 S. Crest Lane	Bellevue	WA	98104-2211	$250.00
C9543029	Sharon	Johnson	1223 Meyer Way	Fife	WA	98222-1123	$856.00
C9549302	Todd	Hayes	1400 NW 88th	Lynnwood	WA	98036-2244	$0.00
C9857432	Homer	Wells	123 Main St.	Seattle	WA	98105-4322	$500.00
C9865874	Mary	Hill	206 McCaffrey	Littleton	CO	80129-5543	$150.00
C9943201	Harry	Sanders	1280 S. Hill Rd.	Fife	WA	98222-2258	$1,000.00

OrderTbl

OrdNo	OrdDate	CustNo	EmpNo	OrdName	OrdStreet	OrdCity	OrdState	OrdZip
O1116324	01/23/2007	C0954327	E8544399	Sheri Gordon	336 Hill St.	Littleton	CO	80129-5543
O1231231	01/23/2007	C9432910	E9954302	Larry Styles	9825 S. Crest Lane	Bellevue	WA	98104-2211
O1241518	02/10/2007	C9549302		Todd Hayes	1400 NW 88th	Lynnwood	WA	98036-2244
O1455122	01/09/2007	C8574932	E9345771	Wally Jones	411 Webber Ave.	Seattle	WA	98105-1093
O1579999	01/05/2007	C9543029	E8544399	Tom Johnson	1632 Ocean Dr.	Des Moines	WA	98222-1123
O1615141	01/23/2007	C8654390	E8544399	Candy Kendall	456 Pine St.	Seattle	WA	98105-3345
O1656777	02/11/2007	C8543321		Ron Thompson	789 122nd St.	Renton	WA	98666-1289
O2233457	01/12/2007	C2388597	E9884325	Beth Taylor	2396 Rafter Rd	Seattle	WA	98103-1121
O2334661	01/14/2007	C0954327	E1329594	Mrs. Ruth Gordon	233 S. 166th	Seattle	WA	98011
O3252629	01/23/2007	C9403348	E9954302	Mike Boren	642 Crest Ave.	Englewood	CO	80113-5431
O3331222	01/13/2007	C1010398		Jim Glussman	1432 E. Ravenna	Denver	CO	80111-0033
O3377543	01/15/2007	C9128574	E8843211	Jerry Wyatt	16212 123rd Ct.	Denver	CO	80222-0022
O4714645	01/11/2007	C2388597	E1329594	Beth Taylor	2396 Rafter Rd	Seattle	WA	98103-1121
O5511365	01/22/2007	C3340959	E9884325	Betty White	4334 153rd NW	Seattle	WA	98178-3311
O6565656	01/20/2007	C9865874	E8843211	Mr. Jack Sibley	166 E. 344th	Renton	WA	98006-5543
O7847172	01/23/2007	C9943201		Harry Sanders	1280 S. Hill Rd.	Fife	WA	98222-2258
O7959898	02/19/2007	C8543321	E8544399	Ron Thompson	789 122nd St.	Renton	WA	98666-1289
O7989497	01/16/2007	C3499503	E9345771	Bob Mann	1190 Lorraine Cir.	Monroe	WA	98013-1095
O8979495	01/23/2007	C9865874		HelenSibley	206 McCaffrey	Renton	WA	98006-5543
O9919699	02/11/2007	C9857432	E9954302	Homer Wells	123 Main St.	Seattle	WA	98105-4322

Employee

EmpNo	EmpFirstName	EmpLastName	EmpPhone	EmpEMail	SupEmpNo	EmpCommRate
E1329594	Landi	Santos	(303) 789-1234	LSantos@bigco.com	E8843211	0.02
E8544399	Joe	Jenkins	(303) 221-9875	JJenkins@bigco.com	E8843211	0.02
E8843211	Amy	Tang	(303) 556-4321	ATang@bigco.com	E9884325	0.04
E9345771	Colin	White	(303) 221-4453	CWhite@bigco.com	E9884325	0.04
E9884325	Thomas	Johnson	(303) 556-9987	TJohnson@bigco.com		0.05
E9954302	Mary	Hill	(303) 556-9871	MHill@bigco.com	E8843211	0.02
E9973110	Theresa	Beck	(720) 320-2234	TBeck@bigco.com	E9884325	

Product

ProdNo	ProdName	ProdMfg	ProdQOH	ProdPrice	ProdNextShipDate
P0036566	17 inch Color Monitor	ColorMeg, Inc.	12	$169.00	2/20/2007
P0036577	19 inch Color Monitor	ColorMeg, Inc.	10	$319.00	2/20/2007
P1114590	R3000 Color Laser Printer	Connex	5	$699.00	1/22/2007
P1412138	10 Foot Printer Cable	Ethlite	100	$12.00	
P1445671	8-Outlet Surge Protector	Intersafe	33	$14.99	
P1556678	CVP Ink Jet Color Printer	Connex	8	$99.00	1/22/2007
P3455443	Color Ink Jet Cartridge	Connex	24	$38.00	1/22/2007
P4200344	36-Bit Color Scanner	UV Components	16	$199.99	1/29/2007
P6677900	Black Ink Jet Cartridge	Connex	44	$25.69	
P9995676	Battery Back-up System	Cybercx	12	$89.00	2/1/2007

OrdLine

OrdNo	ProdNo	Qty
O1116324	P1445671	1
O1231231	P0036566	1
O1231231	P1445671	1
O1241518	P0036577	1
O1455122	P4200344	1
O1579999	P1556678	1
O1579999	P6677900	1
O1579999	P9995676	1
O1615141	P0036566	1
O1615141	P1445671	1
O1615141	P4200344	1
O1656777	P1445671	1
O1656777	P1556678	1
O2233457	P0036577	1
O2233457	P1445671	1
O2334661	P0036566	1
O2334661	P1412138	1
O2334661	P1556678	1
O3252629	P4200344	1
O3252629	P9995676	1
O3331222	P1412138	1
O3331222	P1556678	1
O3331222	P3455443	1
O3377543	P1445671	1
O3377543	P9995676	1
O4714645	P0036566	1
O4714645	P9995676	1
O5511365	P1412138	1
O5511365	P1445671	1
O5511365	P1556678	1
O5511365	P3455443	1
O5511365	P6677900	1
O6565656	P0036566	10
O7847172	P1556678	1
O7847172	P6677900	1
O7959898	P1412138	5
O7959898	P1556678	5
O7959898	P3455443	5
O7959898	P6677900	5
O7989497	P1114590	2
O7989497	P1412138	2
O7989497	P1445671	3
O8979495	P1114590	1
O8979495	P1412138	1
O8979495	P1445671	1
O9919699	P0036577	1
O9919699	P1114590	1
O9919699	P4200344	1

```
CREATE TABLE Customer
(   CustNo              CHAR(8),
    CustFirstName       VARCHAR2(20) CONSTRAINT CustFirstNameRequired NOT NULL,
    CustLastName        VARCHAR2(30) CONSTRAINT CustLastNameRequired NOT NULL,
    CustStreet          VARCHAR2(50),
    CustCity            VARCHAR2(30),
    CustState           CHAR(2),
    CustZip             CHAR(10),
    CustBal             DECIMAL(12,2) DEFAULT 0,
  CONSTRAINT PKCustomer PRIMARY KEY (CustNo)    )
```

```
CREATE TABLE OrderTbl
(                   OrdNo     CHAR(8),
                    OrdDate   DATE      CONSTRAINT OrdDateRequired NOT NULL,
                    CustNo    CHAR(8) CONSTRAINT CustNoRequired NOT NULL,
                    EmpNo     CHAR(8),
                    OrdName   VARCHAR2(50),
                    OrdStreet VARCHAR2(50),
                    OrdCity   VARCHAR2(30),
                    OrdState  CHAR(2),
                    OrdZip    CHAR(10),
CONSTRAINT PKOrderTbl PRIMARY KEY (OrdNo) ,
CONSTRAINT FKCustNo FOREIGN KEY (CustNo) REFERENCES Customer,
CONSTRAINT FKEmpNo FOREIGN KEY (EmpNo) REFERENCES Employee  )
```

```
CREATE TABLE OrdLine
(       OrdNo               CHAR(8),
        ProdNo              CHAR(8),
        Qty                 INTEGER DEFAULT 1,
CONSTRAINT PKOrdLine PRIMARY KEY (OrdNo, ProdNo),
CONSTRAINT FKOrdNo FOREIGN KEY (OrdNo) REFERENCES OrderTbl
      ON DELETE CASCADE,
CONSTRAINT FKProdNo FOREIGN KEY (ProdNo) REFERENCES Product )
```

```
CREATE TABLE Employee
(       EmpNo           CHAR(8),
        EmpFirstName    VARCHAR2(20) CONSTRAINT EmpFirstNameRequired NOT NULL,
        EmpLastName     VARCHAR2(30) CONSTRAINT EmpLastNameRequired NOT NULL,
        EmpPhone        CHAR(15),
        EmpEMail        VARCHAR(50) CONSTRAINT EmpEMailRequired NOT NULL,
        SupEmpNo        CHAR(8),
        EmpCommRate     DECIMAL(3,3),
CONSTRAINT PKEmployee PRIMARY KEY (EmpNo),
CONSTRAINT UNIQUEEMail UNIQUE(EmpEMail),
CONSTRAINT FKSupEmpNo FOREIGN KEY (SupEmpNo) REFERENCES Employee )
```

```
CREATE TABLE Product
(       ProdNo              CHAR(8),
        ProdName            VARCHAR2(50) CONSTRAINT ProdNameRequired NOT NULL,
        ProdMfg             VARCHAR2(20) CONSTRAINT ProdMfgRequired NOT NULL,
        ProdQOH             INTEGER DEFAULT 0,
        ProdPrice           DECIMAL(12,2) DEFAULT 0,
        ProdNextShipDate    DATE,
CONSTRAINT PKProduct PRIMARY KEY (ProdNo)    )
```

Part 1: SELECT

1. List the customer number, the name (first and last), and the balance of customers.
2. List the customer number, the name (first and last), and the balance of customers who reside in Colorado (CustState is CO).
3. List all columns of the *Product* table for products costing more than $50. Order the result by product manufacturer (*ProdMfg*) and product name.
4. List the order number, order date, and shipping name (*OrdName*) of orders sent to addresses in Denver or Englewood.
5. List the customer number, the name (first and last), the city, and the balance of customers who reside in Denver with a balance greater than $150 or who reside in Seattle with a balance greater than $300.
6. List the cities and states where orders have been placed. Remove duplicates from the result.
7. List all columns of the *OrderTbl* table for Internet orders placed in January 2007. An Internet order does not have an associated employee.
8. List all columns of the *OrderTbl* table for phone orders placed in February 2007. A phone order has an associated employee.
9. List all columns of the *Product* table that contain the words *Ink Jet* in the product name.
10. List the order number, order date, and customer number of orders placed after January 23, 2007, shipped to Washington recipients.
11. List the order number, order date, customer number, and customer name (first and last) of orders placed in January 2007 sent to Colorado recipients.
12. List the order number, order date, customer number, and customer name (first and last) of orders placed in January 2007 placed by Colorado customers (*CustState*) but sent to Washington recipients (*OrdState*).
13. List the customer number, name (first and last), and balance of Washington customers who have placed one or more orders in February 2007. Remove duplicate rows from the result.
14. List the order number, order date, customer number, customer name (first and last), employee number, and employee name (first and last) of January 2007 orders placed by Colorado customers.
15. List the employee number, name (first and last), and phone of employees who have taken orders in January 2007 from customers with balances greater than $300. Remove duplicate rows in the result.
16. List the product number, name, and price of products ordered by customer number C0954327 in January 2007. Remove duplicate products in the result.
17. List the customer number, name (first and last), order number, order date, employee number, employee name (first and last), product number, product name, and order cost *(OrdLine.Qty * ProdPrice)* for products ordered on January 23, 2007, in which the order cost exceeds $150.
18. List the average balance of customers by city. Include only customers residing in Washington state (WA).

19. List the average balance of customers by city and short zip code (the first five digits of the zip code). Include only customers residing in Washington State (WA). In Microsoft Access, the expression *left(CustZip, 5)* returns the first five digits of the zip code. In Oracle, the expression *substr(CustZip, 1, 5)* returns the first five digits.
20. List the average balance and number of customers by city. Only include customers residing in Washington State (WA). Eliminate cities in the result with less than two customers.
21. List the number of unique short zip codes and average customer balance by city. Only include customers residing in Washington State (WA). Eliminate cities in the result in which the average balance is less than $100. In Microsoft Access, the expression *left(CustZip, 5)* returns the first five digits of the zip code. In Oracle, the expression *substr(CustZip, 1, 5)* returns the first five digits. (Note: this problem requires two SELECT statements in Access SQL or a nested query in the FROM clause—see Chapter 9).
22. List the order number and total amount for orders placed on January 23, 2007. The total amount of an order is the sum of the quantity times the product price of each product on the order.
23. List the order number, order date, customer name (first and last), and total amount for orders placed on January 23, 2007. The total amount of an order is the sum of the quantity times the product price of each product on the order.
24. List the customer number, customer name (first and last), the sum of the quantity of products ordered, and the total order amount (sum of the product price times the quantity) for orders placed in January 2007. Only include products in which the product name contains the string Ink Jet or Laser. Only include customers who have ordered more than two Ink Jet or Laser products in January 2007.
25. List the product number, product name, sum of the quantity of products ordered, and total order amount (sum of the product price times the quantity) for orders placed in January 2007. Only include products that have more than five products ordered in January 2007. Sort the result in descending order of the total amount.
26. List the order number, the order date, the customer number, the customer name (first and last), the customer state, and the shipping state (*OrdState*) in which the customer state differs from the shipping state.
27. List the employee number, the employee name (first and last), the commission rate, the supervising employee name (first and last), and the commission rate of the supervisor.
28. List the employee number, the employee name (first and last), and total amount of commissions on orders taken in January 2007. The amount of a commission is the sum of the dollar amount of products ordered times the commission rate of the employee.
29. List the union of customers and order recipients. Include the name, street, city, state, and zip in the result. You need to use the concatenation function to combine the first and last names so that they can be compared to the order recipient name. In Access SQL, the & symbol is the concatenation function. In Oracle SQL, the || symbol is the concatenation function.
30. List the first and last name of customers who have the same name (first and last) as an employee.
31. List the employee number and the name (first and last) of second-level subordinates (subordinates of subordinates) of the employee named Thomas Johnson.
32. List the employee number and the name (first and last) of the first- and second-level subordinates of the employee named Thomas Johnson. To distinguish the level of subordinates, include a computed column with the subordinate level (1 or 2).
33. Using a mix of the join operator and the cross product styles, list the names (first and last) of customers who have placed orders taken by Amy Tang. Remove duplicate rows in the result. Note that the join operator style is supported only in Oracle versions 9i and beyond.
34. Using the join operator style, list the product name and the price of all products ordered by Beth Taylor in January 2007. Remove duplicate rows from the result.
35. For Colorado customers, compute the number of orders placed in January 2007. The result should include the customer number, last name, and number of orders placed in January 2007.

36. For Colorado customers, compute the number of orders placed in January 2007 in which the orders contain products made by Connex. The result should include the customer number, last name, and number of orders placed in January 2007.
37. For each employee with a commission rate of less than 0.04, compute the number of orders taken in January 2007. The result should include the employee number, employee last name, and number of orders taken.
38. For each employee with a commission rate greater than 0.03, compute the total commission earned from orders taken in January 2007. The total commission earned is the total order amount times the commission rate. The result should include the employee number, employee last name, and total commission earned.
39. List the total amount of all orders by month in 2007. The result should include the month and the total amount of all orders in each month. The total amount of an individual order is the sum of the quantity times the product price of each product in the order. In Access, the month number can be extracted by the **Month** function with a date as the argument. You can display the month name using the **MonthName** function applied to a month number. In Oracle, the function *to_char(OrdDate, 'M')* extracts the month number from *OrdDate*. Using "MON" instead of "M" extracts the three-digit month abbreviation instead of the month number.
40. List the total commission earned by each employee in each month of 2007. The result should include the month, employee number, employee last name, and the total commission amount earned in that month. The amount of a commission for an individual employee is the sum of the dollar amount of products ordered times the commission rate of the employee. Sort the result by the month in ascending month number and the total commission amount in descending order. In Access, the month number can be extracted by the **Month** function with a date as the argument. You can display the month name using the **MonthName** function applied to a month number. In Oracle, the function *to_char(OrdDate, 'M')* extracts the month number from *OrdDate*. Using "MON" instead of "M" extracts the three-digit month abbreviation instead of the month number.

Part 2: INSERT, UPDATE, and DELETE statements

1. Insert yourself as a new row in the *Customer* table.
2. Insert your roommate, best friend, or significant other as a new row in the *Employee* table.
3. Insert a new *OrderTbl* row with you as the customer, the person from problem 2 (Part 2) as the employee, and your choice of values for the other columns of the *OrderTbl* table.
4. Insert two rows in the *OrdLine* table corresponding to the *OrderTbl* row inserted in problem 3 (Part 2).
5. Increase the price by 10 percent of products containing the words *Ink Jet.*
6. Change the address (street, city, and zip) of the new row inserted in problem 1 (part 2).
7. Identify an order that respects the rules about deleting referenced rows to delete the rows inserted in problems 1 to 4 (Part 2).
8. Delete the new row(s) of the table listed first in the order for problem 7 (Part 2).
9. Delete the new row(s) of the table listed second in the order for problem 7 (Part 2).
10. Delete the new row(s) of the remaining tables listed in the order for problem 7 (Part 2).

References for Further Study

There are many SQL books varying by emphasis on basic coverage, advanced coverage, and product specific coverage. A good summary of SQL books can be found at www.ocelot.ca/books.htm. The DBAZine site (www.dbazine.com) and the DevX.com Database Zone (www.devx.com) have plenty of practical advice about query formulation and SQL. For product-specific SQL advice, the Advisor.com site (www.advisor.com) features technical journals for Microsoft SQL Server and Microsoft Access. Oracle documentation can be found at the Oracle Technet site (www.oracle.com/technology).

Appendix **4.A**

SQL:2003 Syntax Summary

This appendix summarizes SQL:2003 syntax for the SELECT, INSERT, UPDATE, and DELETE statements presented in this chapter. The syntax is limited to the simplified statement structure presented in this chapter. More complex syntax is introduced in Part 5 of this textbook. The conventions used in the syntax notation are identical to those used at the end of Chapter 3.

Simplified SELECT Syntax

```
<Select-Statement>: { <Simple-Select> | <Set-Select> }
                     [ ORDER BY <Sort-Specification>* ]

<Simple-Select>:
                   SELECT  [ DISTINCT ]  <Column-Specification>*
                     FROM <Table-Specification>*
                     [ WHERE <Row-Condition> ]
                     [ GROUP BY ColumnName* ]
                     [ HAVING <Group-Condition> ]

<Column-Specification>: { <Column-List> | <Column-Item> }

<Column-List>: { * | TableName.*  }
                       — * is a literal here not a syntax symbol

<Column-Item>: <Column-Expression> [ AS ColumnName ]

<Column-Expression>:
                      { <Scalar-Expression> | <Aggregate-Expression> }

<Scalar-Expression>:
                     { <Scalar-Item> |
                       <Scalar-Item> <Arith-Operator> <Scalar-Item> }

<Scalar-Item>:
                      { [ TableName.]ColumnName  |
                        Constant   |
                        FunctionName [ (Argument*) ] |
                        <Scalar-Expression>  |
                        ( <Scalar-Expression> ) }

<Arith-Operator>:  { + | – | * | / }
                       — * and + are literals here not syntax symbols

<Aggregate-Expression>:
                     {  SUM ( {<Scalar-Expression> | DISTINCT ColumnName } )  |
                        AVG ( {<Scalar-Expression> | DISTINCT ColumnName } )  |
```

```
                        MIN ( <Scalar-Expression> ) |
                        MAX ( <Scalar-Expression> ) |
                        COUNT ( [ DISTINCT ] ColumnName ) |
                        COUNT ( * ) } — * is a literal symbol here, not a special syntax symbol

<Table-Specification>: { <Simple-Table> |
                         <Join-Operation> }

<Simple-Table>: TableName [ [ AS ] AliasName ]

<Join-Operation>:
                       { <Simple-Table> [INNER] JOIN <Simple-Table>
                          ON <Join-Condition> |
                         { <Simple-Table> | <Join-Operation> } [INNER] JOIN
                          { <Simple-Table> | <Join-Operation> }
                          ON <Join-Condition> |
                         ( <Join-Operation> ) }

<Join-Condition>:    { <Simple-Join-Condition> |
                      <Compound-Join-Condition> }

<Simple-Join-Condition>:
                      <Scalar-Expression> <Comparison-Operator>
                      <Scalar-Expression>

<Compound-Join-Condition>:
                        { NOT <Join-Condition> |
                          <Join-Condition> AND <Join-Condition> |
                          <Join-Condition> OR <Join-Condition> |
                          ( <Join-Condition> )

<Comparison-Operator>: { = | < | > | <= | >= | <> }

<Row-Condition>:
                    { <Simple-Condition> | <Compound-Condition> }

<Simple-Condition>:
                    { <Scalar-Expression> <Comparison-Operator>
                       <Scalar-Experssion> |
                      <Scalar-Expression> [ NOT ] IN ( Constant* ) |
                      <Scalar-Expression> BETWEEN <Scalar-Expression> AND
                       <Scalar-Expression> |
                      <Scalar-Expression> IS [NOT] NULL |
                      ColumnName [ NOT ] LIKE StringPattern }

<Compound-Condition>:
                        { NOT <Row-Condition> |
                          <Row-Condition> AND <Row-Condition> |
                          <Row-Condition> OR <Row-Condition> |
                          ( <Row-Condition> ) }
```

```
<Group-Condition>:
                    { <Simple-Group-Condition> | <Compound-Group-Condition> }

<Simple-Group-Condition>:— permits both scalar and aggregate expressions
                   { <Column-Expression> ComparisonOperator
                       < Column-Experssion>  |
                     <Column-Expression> [ NOT ] IN ( Constant* )  |
                     <Column-Expression> BETWEEN <Column-Expression> AND
                      <Column-Expression> |
                     <Column-Expression> IS [NOT] NULL  |
                     ColumnName [ NOT ] LIKE StringPattern  }

<Compound-Group-Condition>:
                    { NOT <Group-Condition>  |
                      <Group-Condition> AND <Group-Condition>  |
                      <Group-Condition> OR <Group-Condition>  |
                      ( <Group-Condition> )  }

<Sort-Specification>:
                    { ColumnName | ColumnNumber } [ { ASC | DESC } ]

<Set-Select>:
                    { <Simple-Select> | <Set-Select> } <Set-Operator>
                    { <Simple-Select> | <Set-Select> }

<Set-Operator>: { UNION | INTERSECT | EXCEPT } [ ALL ]
```

INSERT Syntax

```
INSERT INTO TableName ( ColumnName* )
                        VALUES ( Constant* )

INSERT INTO TableName [ ( ColumnName* ) ]
                       <Simple-Select>
```

UPDATE Syntax

```
UPDATE TableName
                    SET <Column-Assignment>*
                     [ WHERE <Row-Condition> ]

<Column-Assignment>: ColumnName = <Scalar-Expression>
```

DELETE Syntax

```
DELETE FROM TableName
   [ WHERE  <Row-Condition> ]

DELETE TableName.*     — * is a literal symbol here not a special syntax symbol
   FROM <Join-Operation>
   [ WHERE <Row-Condition> ]
```

Appendix 4.B

Syntax Differences among Major DBMS Products

Table 4B.1 summarizes syntax differences among Microsoft Access (1997 to 2003 versions), Oracle 8i to 10g, Microsoft SQL Server, and IBM's DB2. The differences involve the parts of the SELECT statement presented in the chapter.

TABLE 4B.1 SELECT Syntax Differences among Major DBMS Products

Element\Product	Oracle 8i, 9i, 10g	Access 97/2000/2002/2003	MS SQL Server 2000	DB2
Pattern-matching characters	%, _	*, ? although the % and _ characters can be used in the 2002/2003 versions by setting the query mode	%, _	%, _
Case sensitivity in string matching	Yes	No	Yes	Yes
Date constants	Surround in single quotation marks	Surround in # symbols	Surround in single quotation marks	Surround in single quotation marks
Inequality symbol	< >	< >	!=	< >
Join operator style	No in 8i, Yes in 9i, 10g	Yes	Yes	Yes
Difference operations	MINUS keyword	Not supported	Not supported	EXCEPT keyword