Part IV

Using Interactive SQL to Manipulate a Relational Database

16. Simple SQL Retrieval 323
17. Retrieving Data from More Than One Table 355
18. Advanced Retrieval Operations 375
19. Working With Groups of Rows 399
20. Data Modification 429
21. Creating Additional Structural Elements 437

This part of the book goes in depth into the uses and syntax of interactive SQL. You will learn to create a wide variety of queries, as well as perform data modification. In this part, you will read about the impact that equivalent syntaxes may have on the performance of your queries. Finally, you will learn about creating additional database structural elements that require querying in their syntax.

Page left intentionally blank

Chapter 16 Simple SQL Retrieval

As we've said several times earlier, retrieval is what databases are all about: We put the data in so we can get them out in some meaningful way. There are many ways to formulate a request for data retrieval, but when it comes to relational databases, SQL is the international standard. It may not seem very "modern" because it's a text-based language, but it's the best we have when it comes to something that can be used in many environments.

Note: Yes, SQL is a standard and, in theory, implementations should be the same. However, although the basic syntax is the same, the "devil is in the details," as the old saying goes. For example, some implementations require a semicolon to terminate a statement, but others do not. You will therefore find notes throughout the SQL chapters warning you about areas where implementations may differ. The only solution is to use product documentation to verify the exact syntax used by your DBMS.

SQL has one command for retrieving data: SELECT. This is nowhere as restrictive as it might seem. SELECT contains syntax for choosing columns, choosing rows, combining tables, grouping data, and performing some simple calculations. In fact, a single SELECT statement can result in a DBMS performing any or all of the relational algebra operations.

The basic syntax of the SELECT statement has the following general structure:

```
SELECT column1, column2 …
FROM table1, table2 …
WHERE predicate
```

The SELECT clause specifies the columns you want to see. You specify the tables used in the query in the FROM clause. The optional WHERE

clause can contain a wide variety of criteria that identify which rows you want to retrieve.

Note: Most SQL command processors are not case sensitive when it comes to parts of a SQL statement. SQL keywords, table names, column names, and so on can be in any case you choose. However, most DBMSs are case sensitive when it comes to matching data values. Therefore, whenever you place a value in quotes for SQL to match, you must match the case of the stored data. In this book, SQL keywords will appear in uppercase letters; database components such as column and table names will appear in lowercase letters.

In addition to these basic clauses, SELECT has many other syntax options. Rather than attempt to summarize them all in a single general statement, you will learn to build the parts of a SELECT gradually throughout this and the next few chapters of this book.

Note: The SQL queries you see throughout the book are terminated by a semicolon (;). This is not part of the SQL standard, but is used by many DBMSs so that you can type a command on multiple lines. The SQL command processor doesn't execute the query until it encounters the semicolon.

Note: The examples in this chapter and the remaining SQL retrieval chapters use the rare book store database introduced in Chapter 6.

Revisiting the Sample Data

In Chapter 6 you were introduced to the rare book store database and the sample data with which the tables have been populated. The sample data are repeated here for easier reference when reading Chapters 16–19.

Table 6.1: Publisher

publisher_id	publisher_name
1	Wiley
2	Simon & Schuster
3	Macmillan
4	Tor
5	DAW

Table 6.2: Author

author_numb	author_last_first
1	Bronte, Charlotte
2	Doyle, Sir Arthur Conan
3	Twain, Mark
4	Stevenson, Robert Louis
5	Rand, Ayn
6	Barrie, James
7	Ludlum, Robert
8	Barth, John
9	Herbert, Frank
10	Asimov, Isaac
11	Funke, Cornelia
12	Stephenson, Neal

Table 6.3: Condition codes

condition_code	condition_description
1	New
2	Excellent
3	Fine
4	Good
5	Poor

Choosing Columns

One of the characteristics of a relation is that you can view any of the columns in any order you choose. SQL therefore lets you specify the columns you want to see, and the order in which you want to see them, using the relational algebra project operation to produce the final result table.

Retrieving All Columns

To retrieve all the columns in a table, viewing the columns in the order in which they were defined when the table was created, you can use an asterisk (*) rather than listing each column. For example, to see all the works that the rare book store has handled, you would use

```
SELECT *
FROM work;
```

Table 6.4: Work

work_numb	author_numb	title
1	1	Jane Eyre
2	1	Villette
3	2	Hound of the Baskervilles
4	2	Lost World, The
5	2	Complete Sherlock Holmes
7	3	Prince and the Pauper
8	3	Tom Sawyer
9	3	Adventures of Huckleberry Finn, The
6	3	Connecticut Yankee in King Arthur's Court, A
13	5	Fountainhead, The
14	5	Atlas Shrugged
15	6	Peter Pan
10	7	Bourne Identity, The
11	7	Matarese Circle, The
12	7	Bourne Supremacy, The
16	4	Kidnapped
17	4	Treasure Island
18	8	Sot Weed Factor, The
19	8	Lost in the Funhouse
20	8	Giles Goat Boy
21	9	Dune
22	9	Dune Messiah
23	10	Foundation
24	10	Last Foundation
25	10	I, Robot
26	11	Inkheart
27	11	Inkdeath
28	12	Anathem
29	12	Snow Crash
30	5	Anthem
31	12	Cryptonomicon

Table 6.5: Books

isbn	work_numb	publisher_id	edition	binding	copyright_year
978-1-11111-111-1	1	1	2	Board	1857
978-1-11111-112-1	1	1	1	Board	1847
978-1-11111-113-1	2	4	1	Board	1842
978-1-11111-114-1	3	4	1	Board	1801
978-1-11111-115-1	3	4	10	Leather	1925
978-1-11111-116-1	4	3	1	Board	1805
978-1-11111-117-1	5	5	1	Board	1808
978-1-11111-118-1	5	2	19	Leather	1956
978-1-11111-120-1	8	4	5	Board	1906
978-1-11111-119-1	6	2	3	Board	1956
978-1-11111-121-1	8	1	12	Leath	1982
978-1-11111-122-1	9	1	12	Leather	1982
978-1-11111-123-1	11	2	1	Board	1998
978-1-11111-124-1	12	2	1	Board	1989
978-1-11111-125-1	13	2	3	Board	1965
978-1-11111-126-1	13	2	9	Leath	2001
978-1-11111-127-1	14	2	1	Board	1960
978-1-11111-128-1	16	2	12	Board	1960
978-1-11111-129-1	16	2	14	Leather	2002
978-1-11111-130-1	17	3	6	Leather	1905
978-1-11111-131-1	18	4	6	Board	1957
978-1-11111-132-1	19	4	1	Board	1962
978-1-11111-133-1	20	4	1	Board	1964
978-1-11111-134-1	21	5	1	Board	1964
978-1-11111-135-1	23	5	1	Board	1962
978-1-11111-136-1	23	5	4	Leather	2001
978-1-11111-137-1	24	5	4	Leather	2001
978-1-11111-138-1	23	5	4	Leather	2001
978-1-11111-139-1	25	5	4	Leather	2001
978-1-11111-140-1	26	5	1	Board	2001
978-1-11111-141-1	27	5	1	Board	2005
978-1-11111-142-1	28	5	1	Board	2008
978-1-11111-143-1	29	5	1	Board	1992
978-1-11111-144-1	30	1	1	Board	1952
978-1-11111-145-1	30	5	1	Board	2001
978-1-11111-146-1	31	5	1	Board	1999

Table 6.6: Volume (continues)

inventory_id	isbn	condition_code	date_acquired	asking_price	selling_price	sale_id
1	978-1-11111-111-1	3	12-JUN-19 00:00:00	175.00	175.00	1
2	978-1-11111-131-1	4	23-JAN-20 00:00:00	50.00	50.00	1
7	978-1-11111-137-1	2	20-JUN-19 00:00:00	80.00		
3	978-1-11111-133-1	2	05-APR-18 00:00:00	300.00	285.00	1
4	978-1-11111-142-1	1	05-APR-18 00:00:00	25.95	25.95	2
5	978-1-11111-146-1	1	05-APR-18 00:00:00	22.95	22.95	2
6	978-1-11111-144-1	2	15-MAY-19 00:00:00	80.00	76.10	2
8	978-1-11111-137-1	3	20-JUN-20 00:00:00	50.00		
9	978-1-11111-136-1	1	20-DEC-19 00:00:00	75.00		
10	978-1-11111-136-1	2	15-DEC-19 00:00:00	50.00		
11	978-1-11111-143-1	1	05-APR-20 00:00:00	25.00	25.00	3
12	978-1-11111-132-1	1	12-JUN-20 00:00:00	15.00	15.00	3
13	978-1-11111-133-1	3	20-APR-20 00:00:00	18.00	18.00	3
15	978-1-11111-121-1	2	20-APR-20 00:00:00	110.00	110.00	5
14	978-1-11111-121-1	2	20-APR-20 00:00:00	110.00	110.00	4
16	978-1-11111-121-1	2	20-APR-20 00:00:00	110.00		
17	978-1-11111-124-1	2	12-JAN-21 00:00:00	75.00		
18	978-1-11111-146-1	1	11-MAY-20 00:00:00	30.00	30.00	6
19	978-1-11111-122-1	2	06-MAY-20 00:00:00	75.00	75.00	6
20	978-1-11111-130-1	2	20-APR-20 00:00:00	150.00	120.00	6
21	978-1-11111-126-1	2	20-APR-20 00:00:00	110.00	110.00	6
22	978-1-11111-139-1	2	16-MAY-20 00:00:00	200.00	170.00	6
23	978-1-11111-125-1	2	16-MAY-20 00:00:00	45.00	45.00	7
24	978-1-11111-131-1	3	20-APR-20 00:00:00	35.00	35.00	7
25	978-1-11111-126-1	2	16-NOV-20 00:00:00	75.00	75.00	8
26	978-1-11111-133-1	3	16-NOV-20 00:00:00	35.00	55.00	8
27	978-1-11111-141-1	1	06-NOV-20 00:00:00	24.95		
28	978-1-11111-141-1	1	06-NOV-20 00:00:00	24.95		
29	978-1-11111-141-1	1	06-NOV-20 00:00:00	24.95		
30	978-1-11111-145-1	1	06-NOV-20 00:00:00	27.95		
31	978-1-11111-145-1	1	06-NOV-20 00:00:00	27.95		
32	978-1-11111-145-1	1	06-NOV-20 00:00:00	27.95		
33	978-1-11111-139-1	2	06-OCT-20 00:00:00	75.00	50.00	9
34	978-1-11111-133-1	1	16-NOV-20 00:00:00	125.00	125.00	10
35	978-1-11111-126-1	1	06-OCT-20 00:00:00	75.00	75.00	11
36	978-1-11111-130-1	3	06-DEC-19 00:00:00	50.00	50.00	11
37	978-1-11111-136-1	3	06-DEC-19 00:00:00	75.00	75.00	11
38	978-1-11111-130-1	2	06-APR-20 00:00:00	200.00	150.00	12
39	978-1-11111-132-1	3	06-APR-20 00:00:00	75.00	75.00	12
40	978-1-11111-129-1	1	06-APR-20 00:00:00	25.95	25.95	13
41	978-1-11111-141-1	1	16-MAY-20 00:00:00	40.00	40.00	14
42	978-1-11111-141-1	1	16-MAY-20 00:00:00	40.00	40.00	14
43	978-1-11111-132-1	1	12-NOV-20 00:00:00	17.95		
44	978-1-11111-138-1	1	12-NOV-20 00:00:00	75.95		
45	978-1-11111-138-1	1	12-NOV-20 00:00:00	75.95		
46	978-1-11111-131-1	3	12-NOV-20 00:00:00	15.95		
47	978-1-11111-140-1	3	12-NOV-20 00:00:00	25.95		
48	978-1-11111-123-1	2	16-AUG-20 00:00:00	24.95		
49	978-1-11111-127-1	2	16-AUG-20 00:00:00	27.95		
50	978-1-11111-127-1	2	06-JAN-21 00:00:00	50.00	50.00	15
51	978-1-11111-141-1	2	06-JAN-21 00:00:00	50.00	50.00	15
52	978-1-11111-141-1	2	06-JAN-21 00:00:00	50.00	50.00	16

Table 6.6: Volume (*cont.*)

inventory_id	isbn	condition_ code	date_acquired	asking_price	selling_price	sale_id
53	978-1-11111-123-1	2	06-JAN-21 00:00:00	40.00	40.00	16
54	978-1-11111-127-1	2	06-JAN-21 00:00:00	40.00	40.00	16
55	978-1-11111-133-1	2	06-FEB-21 00:00:00	60.00	60.00	17
56	978-1-11111-127-1	2	16-FEB-21 00:00:00	40.00	40.00	17
57	978-1-11111-135-1	2	16-FEB-21 00:00:00	40.00	40.00	18
59	978-1-11111-127-1	2	25-FEB-21 00:00:00	35.00	35.00	18
58	978-1-11111-131-1	2	16-FEB-21 00:00:00	25.00	25.00	18
60	978-1-11111-128-1	2	16-DEC-20 00:00:00	50.00	45.00	
61	978-1-11111-136-1	3	22-OCT-20 00:00:00	50.00	50.00	19
62	978-1-11111-115-1	2	22-OCT-20 00:00:00	75.00	75.00	20
63	978-1-11111-130-1	2	16-JUL-20 00:00:00	500.00		
64	978-1-11111-136-1	2	06-MAR-20 00:00:00	125.00		
65	978-1-11111-136-1	2	06-MAR-20 00:00:00	125.00		
66	978-1-11111-137-1	2	06-MAR-20 00:00:00	125.00		
67	978-1-11111-137-1	2	06-MAR-20 00:00:00	125.00		
68	978-1-11111-138-1	2	06-MAR-20 00:00:00	125.00		
69	978-1-11111-138-1	2	06-MAR-20 00:00:00	125.00		
70	978-1-11111-139-1	2	06-MAR-20 00:00:00	125.00		
71	978-1-11111-139-1	2	06-MAR-20 00:00:00	125.00		

Because this query is requesting all rows in the table, there is no WHERE clause. As you can see in Figure 16.1, the result table labels each column with its name.

Note: The layouts of the printed output of many SQL queries in this book have been adjusted so that it will fit across the width of the pages. When you actually view listings on the screen, each row will be in a single horizontal line. If a listing is too wide to fit on the screen or a terminal program's window, either each line will wrap as it reaches the right edge of the display, or you will need to scroll.

Table 6.7: Customer

customer_ numb	first_ name	last_ name	street	city	state_ province	zip_ postcode	contact_ phone
1	Janice	Jones	125 Center Road	Anytown	NY	11111	518-555-1111
2	Jon	Jones	25 Elm Road	Next Town	NJ	18888	209-555-2222
3	John	Doe	821 Elm Street	Next Town	NJ	18888	209-555-3333
4	Jane	Doe	852 Main Street	Anytown	NY	11111	518-555-4444
5	Jane	Smith	1919 Main Street	New Village	NY	13333	518-555-5555
6	Janice	Smith	800 Center Road	Anytown	NY	11111	518-555-6666
7	Helen	Brown	25 Front Street	Anytown	NY	11111	518-555-7777
8	Helen	Jerry	16 Main Street	Newtown	NJ	18886	518-555-8888
9	Mary	Collins	301 Pine Road, Apt. 12	Newtown	NJ	18886	518-555-9999
10	Peter	Collins	18 Main Street	Newtown	NJ	18886	518-555-1010
11	Edna	Hayes	209 Circle Road	Anytown	NY	11111	518-555-1110
12	Franklin	Hayes	615 Circle Road	Anytown	NY	11111	518-555-1212
13	Peter	Johnson	22 Rose Court	Next Town	NJ	18888	209-555-1212
14	Peter	Johnson	881 Front Street	Next Town	NJ	18888	209-555-1414
15	John	Smith	881 Manor Lane	Next Town	NJ	18888	209-555-1515

Table 6.8: Sale

sale_id	customer_ numb	sale_date	sale_total_ amt	credit_card_ numb	exp_month	exp_year
3	1	15-JUN-21 00:00:00	58.00	1234 5678 9101 1121	10	18
4	4	30-JUN-21 00:00:00	110.00	1234 5678 9101 5555	7	17
5	6	30-JUN-21 00:00:00	110.00	1234 5678 9101 6666	12	17
6	12	05-JUL-21 00:00:00	505.00	1234 5678 9101 7777	7	16
7	8	05-JUL-21 00:00:00	80.00	1234 5678 9101 8888	8	16
8	5	07-JUL-21 00:00:00	90.00	1234 5678 9101 9999	9	15
9	8	07-JUL-21 00:00:00	50.00	1234 5678 9101 8888	8	16
10	11	10-JUL-21 00:00:00	125.00	1234 5678 9101 1010	11	16
11	9	10-JUL-21 00:00:00	200.00	1234 5678 9101 0909	11	15
12	10	10-JUL-21 00:00:00	200.00	1234 5678 9101 0101	10	15
13	2	10-JUL-21 00:00:00	25.95	1234 5678 9101 2222	2	15
14	6	10-JUL-21 00:00:00	80.00	1234 5678 9101 6666	12	17
15	11	12-JUL-21 00:00:00	75.00	1234 5678 9101 1231	11	17
16	2	25-JUL-21 00:00:00	130.00	1234 5678 9101 2222	2	15
17	1	25-JUL-21 00:00:00	100.00	1234 5678 9101 1121	10	18
18	5	22-AUG-21 00:00:00	100.00	1234 5678 9101 9999	9	15
2	1	05-JUN-21 00:00:00	125.00	1234 5678 9101 1121	10	18
1	1	29-MAY-21 00:00:00	510.00	1234 5678 9101 1121	10	18
19	6	01-SEP-21 00:00:00	95.00	1234 5678 9101 7777	7	16
20	2	01-SEP-21 00:00:00	75.00	1234 5678 9101 2222	2	15

Using the * operator to view all columns is a convenient shorthand for interactive SQL when you want a quick overview of data. However, it can be troublesome when used in embedded SQL (SQL inside a program of some sort) if the columns in the table are changed. In particular, if a column is added to the table and the application is not modified to handle the new column, then the application may not work properly. The * operator also displays the columns in the order in which they appeared in the CREATE statement that set up the table. This may simply not be the order you want.

Retrieving Specific Columns

In most SQL queries, you will want to specify exactly which column or columns you want retrieved and perhaps the order in which you want them to appear. To specify columns, you list them following SELECT in the order in which you want to see them. For example, a query to view the names and phone numbers of all of our store's customers is written

```
SELECT first_name,last_name,contact_phone
FROM customer;
```

work_numb	author_numb	title
1	1	Jane Eyre
2	1	Villette
3	2	Hound of the Baskervilles
4	2	Lost World, The
5	2	Complete Sherlock Holmes
7	3	Prince and the Pauper
8	3	Tom Sawyer
9	3	Adventures of Huckleberry Finn, The
6	3	Connecticut Yankee in King Arthur's Court, A
13	5	Fountainhead, The
14	5	Atlas Shrugged
15	6	Peter Pan
10	7	Bourne Identity, The
11	7	Matarese Circle, The
12	7	Bourne Supremacy, The
16	4	Kidnapped
17	4	Treasure Island
18	8	Sot Weed Factor, The
19	8	Lost in the Funhouse
20	8	Giles Goat Boy
21	9	Dune
22	9	Dune Messiah
23	10	Foundation
24	10	Last Foundation
25	10	I, Robot
26	11	Inkheart
27	11	Inkdeath
28	12	Anathem
29	12	Snow Crash
30	5	Anthem
31	12	Cryptonomicon

■ **FIGURE 16.1 Viewing all columns in a table.**

The result (see Figure 16.2) shows all rows in the table for just the three columns specified in the query. The order of the columns in the result table matches the order in which the columns appeared after the SELECT keyword.

Removing Duplicates

Unique primary keys ensure that relations have no duplicate rows. However, when you view only a portion of the columns in a table, you may end up with duplicates. For example, executing the following query produced the result in Figure 16.3:

```
SELET customer_numb, credit_card_numb
FROM sale;
```

first_name	last_name	contact_phone
Janice	Jones	518-555-1111
Jon	Jones	209-555-2222
John	Doe	209-555-3333
Jane	Doe	518-555-4444
Jane	Smith	518-555-5555
Janice	Smith	518-555-6666
Helen	Brown	518-555-7777
Helen	Jerry	518-555-8888
Mary	Collins	518-555-9999
Peter	Collins	518-555-1010
Edna	Hayes	518-555-1110
Franklin	Hayes	518-555-1212
Peter	Johnson	209-555-1212
Peter	Johnson	209-555-1414
John	Smith	209-555-1515

■ **FIGURE 16.2 Choosing specific columns.**

customer_numb	credit_card_numb
1	1234 5678 9101 1121
1	1234 5678 9101 1121
1	1234 5678 9101 1121
1	1234 5678 9101 1121
2	1234 5678 9101 2222
2	1234 5678 9101 2222
2	1234 5678 9101 2222
4	1234 5678 9101 5555
5	1234 5678 9101 9999
5	1234 5678 9101 9999
6	1234 5678 9101 6666
6	1234 5678 9101 7777
6	1234 5678 9101 6666
8	1234 5678 9101 8888
8	1234 5678 9101 8888
9	1234 5678 9101 0909
10	1234 5678 9101 0101
11	1234 5678 9101 1231
11	1234 5678 9101 1010
12	1234 5678 9101 7777

■ **FIGURE 16.3 A result table with duplicate rows.**

Duplicates appear because the same customer used the same credit card number for more than one purchase. Keep in mind that, although this table with duplicate rows is not a legal relation, that doesn't present a problem for the database because it is not stored in the base tables.

To remove duplicates from a result table, you insert the keyword DISTINCT following SELECT:

```
SELECT DISTINCT customer_numb, credit_card_numb
FROM sale;
```

The result is a table without the duplicate rows (see Figure 16.4). Although a legal relation has no duplicate rows, most DBMS vendors have implemented SQL retrieval so that it leaves the duplicates. As you read earlier the primary reason is performance. To remove duplicates, a DBMS must sort the result table by every column in the table. It must then scan the table from top to bottom, looking at every "next" row to identify duplicate rows that are next to one another. If a result table is large, the sorting and scanning can significantly slow down the query. It is therefore up to the user to decide whether to request unique rows.

```
 customer_numb |   credit_card_numb
---------------+---------------------
             1 | 1234 5678 9101 1121
             2 | 1234 5678 9101 2222
             4 | 1234 5678 9101 5555
             5 | 1234 5678 9101 9999
             6 | 1234 5678 9101 6666
             6 | 1234 5678 9101 7777
             8 | 1234 5678 9101 8888
             9 | 1234 5678 9101 0909
            10 | 1234 5678 9101 0101
            11 | 1234 5678 9101 1010
            11 | 1234 5678 9101 1231
            12 | 1234 5678 9101 7777
```

■ **FIGURE 16.4 The result table in Figure 16.3 with the duplicates removed.**

Ordering the Result Table

The order in which rows appear in the result table may not be what you expect. In some cases, rows will appear in the order in which they are physically stored. However, if the query optimizer uses an index to process the query, then the rows will appear in index key order. If you want row ordering to be consistent and predictable, you will need to specify how you want the rows to appear.

```
 author_numb |           author_last_first
-------------+-------------------------------------------------------
           1 | Bronte, Charlotte
           2 | Doyle, Sir Arthur Conan
           3 | Twain, Mark
           4 | Stevenson, Robert Louis
           5 | Rand, Ayn
           6 | Barrie, James
           7 | Ludlum, Robert
           8 | Barth, John
           9 | Herbert, Frank
          10 | Asimov, Isaac
          11 | Funke, Cornelia
          12 | Stephenson, Neal
```

■ **FIGURE 16.5 The unordered result table.**

When you want to control the order of rows in a result table, you add an ORDER BY clause to your SELECT statement.

For example, if you issue the query

```
SELECT *
FROM author;
```

to get an alphabetical list of authors, you will see the listing in Figure 16.5. The author numbers are in numeric order because that was the order in which the rows were added, but the author names aren't alphabetical. Adding the ORDER BY clause sorts the result in alphabetical order by the *author_first_last* column (see Figure 16.6):

```
SELECT *
FROM author
ORDER BY author_last_first;
```

The keywords ORDER BY are followed by the column or columns on which you want to sort the result table. When you include more than one column, the first column represents the outer sort, the next column a sort within it. For example, assume that you issue the query

```
SELECT zip_postcode, last_name, first_name
FROM customer
ORDER BY zip_postcode, last_name;
```

author_numb	author_last_first
10	Asimov, Isaac
6	Barrie, James
8	Barth, John
1	Bronte, Charlotte
2	Doyle, Sir Arthur Conan
11	Funke, Cornelia
9	Herbert, Frank
7	Ludlum, Robert
5	Rand, Ayn
12	Stephenson, Neal
4	Stevenson, Robert Louis
3	Twain, Mark

■ **FIGURE 16.6 The result table from Figure 16.5 sorted in alphabetical order by author name.**

zip_postcode	last_name	first_name
11111	Brown	Helen
11111	Doe	Jane
11111	Hayes	Edna
11111	Hayes	Franklin
11111	Jones	Janice
11111	Smith	Janice
13333	Smith	Jane
18886	Collins	Mary
18886	Collins	Peter
18886	Jerry	Helen
18888	Doe	John
18888	Johnson	Peter
18888	Johnson	Peter
18888	Jones	Jon
18888	Smith	John

■ **FIGURE 16.7 Sorting output by two columns.**

The result (see Figure 16.7) first orders by the zip code, and then sorts by the customer's last name within each zip code. If we reverse the order of the columns on which the output is to be sorted, as in

```
SELECT zip_postcode, last_name, first_name
FROM customer
ORDER BY last_name, zip_postcode;
```

```
 zip_postcode | last_name | first_name
--------------+-----------+------------
 11111        | Brown     | Helen
 18886        | Collins   | Peter
 18886        | Collins   | Mary
 11111        | Doe       | Jane
 18888        | Doe       | John
 11111        | Hayes     | Franklin
 11111        | Hayes     | Edna
 18886        | Jerry     | Helen
 18888        | Johnson   | Peter
 18888        | Johnson   | Peter
 11111        | Jones     | Janice
 18888        | Jones     | Jon
 11111        | Smith     | Janice
 13333        | Smith     | Jane
 18888        | Smith     | John
```

■ **FIGURE 16.8 Reversing the sort order of the query in Figure 16.7.**

the output (see Figure 16.8) then sorts first by last name, and then by zip code within each last name.

Choosing Rows

As well as viewing any columns from a relation, you can also view any rows you want. We specify row selection criteria in a SELECT statement's WHERE clause.

In its simplest form, a WHERE clause contains a logical expression against which each row in a table is evaluated. If a row meets the criteria in the expression, then it becomes a part of the result table. If the row does not meet the criteria, then it is omitted. The trick to writing row selection criteria is, therefore, knowing how to create logical expressions against which data can be evaluated.

Predicates

As you read earlier, a logical expression that follows WHERE is known as a predicate. It uses a variety of operators to represent row selection criteria. If a row meets the criteria in a predicate (in other words, the logical expression evaluates as true), then the row is included in the result table. If the row doesn't meet the criteria (the logical expression evaluates as false), then the row is excluded.

Relationship Operators

In Table 16.1 you can see the six operators used to express data relationships. To write an expression using one of the operators, you surround it with two values. In database queries, such expressions have either a column name on one side and a literal value on the other, as in

```
cost > 1.95
```

or column names on both sides:

```
numb_on_hand <= reorder_point
```

Table 16.1 The Relationship Operators

Operator	Meaning	Examples
=	Equal to	`cost = 1.95` `numb_in_stock = reorder_point`
<	Less than	`cost < 1.95` `numb_in_stock < reorder_point`
<=	Less than or equal to	`cost <= 1.95` `numb_in_stock <= reorder_point`
>	Greater than	`cost > 1.95` `numb_in_stock >= reorder_point`
>=	Greater than or equal to	`cost >= 1.95` `numb_in_stock >= reorder_point`
!= or < > [1]	Not equal to	`cost != 1.95` `numb_in_stock != reorder_point`

[1] *Check the documentation that accompanies your DBMS to determine whether the "not equal to" operator is != or < >.*

The first expression asks the question "Is the cost of the item greater than 1.95?" The second asks "Is the number of items in stock less than or equal to the reorder point?"

The way in which you enter literal values into a logical expression depends on the data type of the column to which you are comparing the value:

- Numbers: type numbers without any formatting. In other words, leave out dollar signs, commas, and so on. You should, however, put decimal points in the appropriate place in a number with a factional portion.
- Characters: type characters surrounded by quotation marks. Most DBMSs accept pairs of either single or double quotes. If your

characters include an apostrophe (a single quote), then you should use double quotes. Otherwise, use single quotes.
- Dates: type dates in the format used to store them in the database. This will vary from one DBMS to another.
- Times: type times in the format used to store them in the database. This will vary from one DBMS to another.

When you are using two column names, keep in mind that the predicate is applied to each row in the table individually. The DBMS substitutes the values stored in the columns in the same row, when making its evaluation of the criteria. You can, therefore, use two column names when you need to examine data that are stored in the same row but in different columns. However, you cannot use a simple logical expression to compare the same column in two or more rows.

The DBMS also bases the way it evaluates data on the type of data:

- Comparisons involving numeric data are based on numerical order.
- Comparisons involving character data are based on alphabetical order.
- Comparisons involving dates and times are based on chronological order.

Logical Operators

Sometimes a simple logical expression is not enough to identify the rows you want to retrieve; you need more than one criterion. In that case, you can chain criteria together with logical operators. For example, assume that you want to retrieve volumes that you have in stock that cost more than $75, and that are in excellent condition. The conditions are coded (2 = excellent). The predicate you need is, therefore, made up of two simple expressions:

```
condition_code = 2
asking_price > 75
```

A row must meet both of these criteria to be included in the result table. You therefore connect the two expressions with the logical operator AND into a single complex expression:

```
condition_code = 2 AND asking_price > 75
```

Whenever you connect two simple expressions with AND, a row must meet *both* of the conditions to be included in the result.

You can use the AND operators to create a predicate that includes a range of dates. For example, if you want to find all sales that were made in Aug. and Sep. of 2019, the predicate would be written:[1]

```
sale_date >= '01-Aug-2019' AND sale_date <= '30-Sep-2019'
```

The same result would be generated by

```
Sale_date > '31-Jul-2019' AND sale_date < '1-Oct-2019'
```

To be within the interval, a sale date must meet both individual criteria.

You will find a summary of the action of the AND operators in Table 16.2. The labels in the columns and rows represent the result of evaluating the single expressions on either side of the AND. As you can see, the only way to get a true result for the entire expression is for both simple expressions to be true.

Table 16.2 AND Truth Table

AND	True	False
True	True	False
False	False	False

If you want to create an expression from which a row needs to meet only one condition from several conditions, then you connect simple expressions with the logical operator OR. For example, if you want to retrieve volumes that cost more than $125 or less than $50, you would use the predicate

```
asking_price > 125 OR asking_price < 50
```

Whenever you connect two simple expressions with OR, a row needs to meet only one of the conditions to be included in the result of the query. When you want to create a predicate that looks for dates outside an interval,

[1]This date format is a fairly generic one that is recognized by many DMBSs. However, you should consult the documentation for your DBMS to determine exactly what will work with your product.

you use the OR operator. For example, to see sales that occurred prior to Mar. 1, 2019 or after Dec. 31, 2019, the predicate is written

```
sale_date < '01-Mar-2019' OR sale_date > '31-Dec-2019'
```

You can find a summary of the OR operation in Table 16.3. Notice that the only way to get a false result is for both simple expressions surrounding OR to be false.

Table 16.3 OR Truth Table

OR	True	False
True	True	True
False	True	False

There is no limit to the number of simple expression you can connect with AND and OR. For example, the following expression is legal:

```
condition_code >= 3 AND selling_price < asking_price
     AND selling_price > 75
```

Negation

The logical operator NOT (or !) inverts the result of logical expression. If a row meets the criteria in a predicate, then placing NOT in front of the criteria *excludes* the row from the result. By the same token, if a row does not meet the criteria in a predicate, then placing NOT in front of the expression *includes* the row in the result. For example,

```
NOT (asking_price <= 50)
```

retrieves all rows where the cost is not less than or equal to $50 (in other words, greater than $50[2]). First the DBMS evaluates the value in the *asking_price* column against the expression *asking_price* <= 50. If the row meets the criteria, then the DBMS does nothing. If the row does not meet the criteria, it includes the row in the result.

[2]It is actually faster for the DBMS to evaluate *asking_price* > 50 because it involves only one action (is the asking price greater than 50) rather than three: Is the asking price equal to 50? Is the asking price less than 50? Take the opposite of the result of the first two questions.

The parentheses in the preceding example group the expression to which NOT is to be applied. In the following example, the NOT operator applies only to the expression *asking_price* <= 50.

```
NOT (asking_price <= 50) AND selling_price < asking_price
```

NOT can be a bit tricky when it is applied to complex expressions. As an example, consider this expression:

```
NOT (asking_price <= 50 AND selling_price < asking_price)
```

Rows that have both an asking price of less than or equal to $50, and a selling price that was less than the asking price will meet the criteria within parentheses. However, the NOT operator excludes them from the result. Those rows that have either an asking price of more than $50 or a selling price greater than or equal to the asking price will fail the criteria within the parentheses, but will be included in the result by the NOT. This means that the expression is actually the same as

```
asking_price > 50 OR selling_price >= asking_price
```

or

```
NOT (asking_price <= 50) OR NOT (selling_price < asking_price)
```

Precedence ad Parentheses

When you create an expression with more than one logical operation, the DBMS must decide on the order in which it will process the simple expressions. Unless you tell it otherwise, a DBMS uses a set of default *rules of precedence*. In general, a DBMS evaluates simple expressions first, followed by any logical expressions. When there is more than one operator of the same type, evaluation proceeds from left to right.

As a first example, consider this expression:

```
asking_price < 50 OR condition_code = 2 AND selling_price >
     asking_price
```

If the asking price of a book is $25, its condition code is 3, and the selling price was $20, the DBMS will exclude the row from the result. The first simple expression is true; the second is false. An OR between the first two produces a true result, because at least one of the criteria is true. Then the DBMS performs an AND between the true result of the first portion and the result of the third simple expression (false). Because we are combining a true result and a false result with AND, the overall result is false. The row is therefore excluded from the result.

We can change the order in which the DBMS evaluates the logical operators and, coincidentally, the result of the expression, by using parentheses to group the expressions that are to have higher precedence:

```
asking_price < 50 OR (condition_code = 2 AND selling_price >
     asking_price)
```

A DBMS gives highest precedence to the parts of the expression within parentheses. Using the same sample data from the preceding paragraph, the expression within parentheses is false (both simple expressions are false). However, the OR with the first simple expression produces true, because the first simple expression is true. Therefore, the row is included in the result.

Special Operators

SQL predicates can include a number of special operators that make writing logical criteria easier. These include BETWEEN, LIKE, IN, and IS NULL.

Note: There are additional operators that are used primarily with subqueries, SELECT statements in which you embed one complete SELECT within another. You will be introduced to them in Chapter 17.

Between

The BETWEEN operator simplifies writing predicates that look for values that lie within an interval. Remember the example you saw earlier in this chapter using AND to generate a date interval? Using the BETWEEN operator, you could rewrite that predicate as

```
sale_date BETWEEN '01-Aug-2021' AND '30-Sep-2021'
```

Any row with a sale date of Aug. 1, 2021 through Sep. 30, 2021 will be included in the result.

If you negate the BETWEEN operator, the DBMS returns all rows that are outside the interval. For example,

```
sale_date NOT BETWEEN '01-Aug-2021' AND '30-Sep-2021'
```

retrieves all rows with dates *prior* to Aug. 1, 2019 and *after* Sep. 31, 2021. It does not include 01-Aug-2021 or 30-Sep-2021. NOT BETWEEN is therefore a shorthand for the two simple expressions linked by OR that you saw earlier in this chapter.

Like

The LIKE operator provides a measure of character string pattern matching by allowing you to use placeholders (wildcards) for one or more characters. Although you may find that the wildcards are different in your particular DBMS, in most case, % stands for zero, one, or more characters and _ stands for zero or one character.

The way in which the LIKE operator works is summarized in Table 16.4. As you can see, you can combine the two wildcards to produce a variety of begins with, ends with, and contains expressions.

Table 16.4 Using the LIKE Operator

Expression	Meaning
`LIKE 'Sm%'`	Begins with *Sm*
`LIKE '%ith'`	Ends with *ith*
`LIKE '%ith%'`	Contains *ith*
`LIKE 'Sm_'`	Begins with *Sm* and is followed by at most one character
`LIKE '_ith'`	Ends with *ith* and is preceded by at most one character
`LIKE '_ith_'`	Ontains *ith* and begins and ends with at most one additional character
`LIKE '%ith_'`	Contains *ith*, begins with any number of characters, and ends with at most one additional character
`LIKE '_ith%'`	Contains *ith*, begins with at most one additional character, and ends with any number of characters

As with BETWEEN you can negate the LIKE operator:

```
last_name NOT LIKE 'Sm%'
```

Rows that are like the pattern are therefore excluded from the result.

One of the problems you may run into when using LIKE is that you need to include the wildcard characters as part of your data. For example, what can you do if you want rows that contain 'nd_by'? The expression you want is

```
column_name LIKE '%nd_by%'
```

The problem is that the DBMS will see the _ as a wildcard, rather than as characters in your search string. The solution was introduced in SQL-92, providing you with the ability to designate an *escape character*.

An escape character removes the special meaning of the character that follows. Because many programming languages use \ as the escape character, it is a logical choice for pattern matching, although it can be any character that is not part of your data. To establish the escape character, you add the keyword ESCAPE, followed by the escape character, to your expression:

```
column_name LIKE '%nd\_by%' ESCAPE '\'
```

In

The IN operator compares the value in a column against a set of values. IN returns true if the value is within the set. For example, assume that a store employee is checking the selling price of a book and wants to know if it is $25, $50, or $60. Using the IN operator, the expression would be written:

```
selling_price IN (25,50,60)
```

This is shorthand for

```
selling_price = 25 OR selling_price = 50 OR selling_price = 60
```

Therefore, any row whose price is one of those three values will be included in the result. Conversely, if you write the predicate

```
selling_price NOT IN (25,50,60)
```

the DBMS will return the rows with prices other than those in the set of values. The preceding expression is therefore the same as

```
selling_price != 25 AND selling_price != 50 AND selling_price !=60
```

or

```
NOT (selling_price = 25 OR selling_price = 50 OR selling_proce = 60)
```

Note: The most common use of IN and NOT IN is with a subquery, where the set of values to which data are compared are generated by an embedded SELECT. You will learn about this is Chapter 17.

Is Null

As you know, null is a specific indicator in a database. Although columns that contain nulls appear empty when you view them, the database actually stores a value that represents null, so that an unknown value can be distinguished from, for example, a string value containing a blank. As a user, however, you will rarely know exactly what a DBMS is using internally for null. This means that you need some special way to identify null in a predicate, so you can retrieve rows containing nulls. That is where the IS NULL operator comes in.

For example, an expression to identify all rows for volumes that have not been sold is written as

```
sale_date IS NULL
```

Conversely, to find all volumes that have been sold, you could use

```
sale_date IS NOT NULL
```

Performing Row Selection Queries

To perform SQL queries that select specific rows, you place a predicate after the SQL keyword WHERE. Depending on the nature of the predicate, the intention of the query may be to retrieve one or more rows. In this section, you therefore will see some SELECT examples that combine a variety of row selection criteria. You will also see how those criteria are combined in queries with column selection, and with sorting of the output.

Using a Primary Key Expression to Retrieve One Row

A common type of SQL retrieval query uses a primary key expression in its predicate to retrieve exactly one row. For example, if someone at the rare book store wants to see the name and telephone number of customer number 6, then the query is written

```
SELECT first_name, last_name, contact_phone
FROM customer
WHERE customer_numb = 6;
```

The result is the single row requested by the predicate.

```
first_name | last_name | contact_phone
-----------+-----------+---------------
Janice     | Smith     | 518-555-6666
```

If a table has a concatenated primary key, such as the employee number and child name for the *dependents* table you saw earlier in the book, then a primary key expression needs to include a complex predicate in which each column of the primary key appears in its own simple logical expression. For example, if you wanted to find the birthdate of employee number 0002's son John, you would use following query:

```
SELECT child_birth_date
FROM dependents
WHERE employee_number = '0002' AND child_name = 'John';
```

In this case, the result is simply

```
child_birth_date
----------------
2-Dec-1999
```

Retrieving Multiple Rows

Although queries with primary key expressions are written with the intention of retrieving only one row, more commonly SQL queries are designed to retrieve multiple rows.

Using Simple Predicates

When you want to retrieve data based on a value in a single column, you construct a predicate that includes just a simple logical expression. For

example, to see all the books ordered on sale number 6, an application program being used by a store employee would use

```
SELECT isbn
FROM volume
WHERE sale_id = 6;
```

The output (see Figure 16.9) displays a single column for rows where the *sale_id* is 6.

```
isbn
-------------------
978-1-11111-146-1
978-1-11111-122-1
978-1-11111-130-1
978-1-11111-126-1
978-1-11111-139-1
```

■ **FIGURE 16.9 Displaying a single column from multiple rows using a simple predicate.**

Using Complex Predicates

When you want to see rows that meet two or more simple conditions, you use a complex predicate in which the simple conditions are connected by AND or OR. For example, if someone wanted to see the books on order number 6 that sold for less than the asking price, the query would be written

```
SELECT isbn
FROM volume
WHERE sale_id = 6 AND selling_price < asking_price;
```

Only two rows meet the criteria:

```
isbn
-------------------
978-1-11111-130-1
978-1-11111-139-1
```

By the same token, if you wanted to see all sales that took place prior to Aug. 1, 2021, and for which the total amount of the sale was less than $100, the query would be written[3]

[3]Whether you need quotes around date expressions depends on the DBMS.

```
SELECT sale_id, sale_total_amt
FROM sale
WHERE sale_date < '1-Aug-2021' AND sale_total_amt < 100;
```

It produces the result in Figure 16.10.

```
 sale_id | sale_total_amt
---------+----------------
       3 |          58.00
       7 |          80.00
       8 |          90.00
       9 |          50.00
      13 |          25.95
      14 |          80.00
      15 |          75.00
```

■ **FIGURE 16.10 Retrieving rows using a complex predicate including a date.**

Note: Don't forget that the date format required by your DMBS may be different from the one used in examples in this book.

Alternatively, if you needed information about all sales that occurred prior to or on Aug. 1, 2021 that totaled more than 100, along with sales that occurred after Aug. 1, 2019 that totaled less than 100, you would write the query

```
SELECT sale_id, sale_date, sale_total_amt
FROM sale
WHERE (sale_date <= '1-Aug-2021' AND sale_total_amt > 100) OR
(sale_date > '1-Aug-2021' AND sale_total_amt < 100);
```

Notice that although the AND operator has precedence over OR and, therefore, the parentheses are not strictly necessary, the predicate in this query includes parentheses for clarity. Extra parentheses are never a problem—as long as you balance every opening parenthesis with a closing parenthesis—and you should feel free to use them whenever they help make it easier to understand the meaning of a complex predicate. The result of this query can be seen in Figure 16.11.

Note: The default output format for a column of type date in PostgreSQL (the DBMS used to generate the sample queries) includes the time. Because no time was entered (just a date), the time displays as all 0s.

Using Between and Not Between

As an example of using one of the special predicate operators, consider a query where someone wants to see all sales that occurred between Jul. 1, 2019 and Aug. 31, 2019. The query would be written

sale_id	sale_date	sale_total_amt
4	30-JUN-21 00:00:00	110.00
5	30-JUN-21 00:00:00	110.00
6	05-JUL-21 00:00:00	505.00
10	10-JUL-21 00:00:00	125.00
11	10-JUL-21 00:00:00	200.00
12	10-JUL-21 00:00:00	200.00
16	25-JUL-21 00:00:00	130.00
2	05-JUN-21 00:00:00	125.00
1	29-MAY-21 00:00:00	510.00
19	01-SEP-21 00:00:00	95.00
20	01-SEP-21 00:00:00	75.00

■ **FIGURE 16.11 Using a complex predicate that includes multiple logical operators.**

```
SELECT sale_id, sale_date, sale_total_amt
FROM sale
WHERE sale_date BETWEEN '1-Jul-2021' AND '31-Aug-2021';
```

It produces the output in Figure 16.12.

sale_id	sale_date	sale_total_amt
6	05-JUL-21 00:00:00	505.00
7	05-JUL-21 00:00:00	80.00
8	07-JUL-21 00:00:00	90.00
9	07-JUL-21 00:00:00	50.00
10	10-JUL-21 00:00:00	125.00
11	10-JUL-21 00:00:00	200.00
12	10-JUL-21 00:00:00	200.00
13	10-JUL-21 00:00:00	25.95
14	10-JUL-21 00:00:00	80.00
15	12-JUL-21 00:00:00	75.00
16	25-JUL-21 00:00:00	130.00
17	25-JUL-21 00:00:00	100.00
18	22-AUG-21 00:00:00	100.00

■ **FIGURE 16.12 Using BETWEEN to retrieve rows in a date range.**

The inverse query retrieves all orders not placed between Jul. 1, 2019 and Aug. 31, 2019 is written

```
SELECT sale_id, sale_date, sale_total_amt
FROM sale
WHERE sale_date NOT BETWEEN '1-Jul-2019' AND '31-Aug-2019';
```

and produces the output in Figure 16.13.

sale_id	sale_date	sale_total_amt
3	15-JUN-21 00:00:00	58.00
4	30-JUN-21 00:00:00	110.00
5	30-JUN-21 00:00:00	110.00
2	05-JUN-21 00:00:00	125.00
1	29-MAY-21 00:00:00	510.00
19	01-SEP-21 00:00:00	95.00
20	01-SEP-21 00:00:00	75.00

■ **FIGURE 16.13 Using NOT BETWEEN to retrieve rows outside a date range.**

If we want output that is easier to read, we might ask the DBMS to sort the result by sale date:

```
SELECT sale_id, sale_date, sale_total_amt
FROM sale
WHERE sale_date NOT BETWEEN '1-Jul-2021' AND '31-Aug-2021'
ORDER BY sale_date;
```

producing the result in Figure 16.14.

sale_id	sale_date	sale_total_amt
1	29-MAY-21 00:00:00	510.00
2	05-JUN-21 00:00:00	125.00
3	15-JUN-21 00:00:00	58.00
5	30-JUN-21 00:00:00	110.00
4	30-JUN-21 00:00:00	110.00
19	01-SEP-21 00:00:00	95.00
20	01-SEP-21 00:00:00	75.00

■ **FIGURE 16.14 Output sorted by date.**

Nulls and Retrieval: Three-Valued Logic

The predicates you have seen to this point omit one important thing: the presence of nulls. What should a DBMS do when it encounters a row that contains null, rather than a known value? As you know, the relational data model doesn't have a specific rule as to what a DBMS should do, but it does require that the DBMS act consistently when it encounters nulls.

Consider the following query as an example:

```
SELECT inventory_id, selling_price
FROM volume
WHERE selling_price < 100;
```

The result can be found in Figure 16.15. Notice that every row in the result table has a value of selling price, which means that rows for unsold items—those with null in the selling price column—are omitted.

inventory_id	selling_price
2	50.00
4	25.95
5	22.95
6	76.10
11	25.00
12	15.00
13	18.00
18	30.00
19	75.00
23	45.00
24	35.00
25	75.00
26	55.00
33	50.00
35	75.00
36	50.00
37	75.00
39	75.00
40	25.95
41	40.00
42	40.00
50	50.00
51	50.00
52	50.00
53	40.00
54	40.00
55	60.00
56	40.00
57	40.00
59	35.00
58	25.00
60	45.00
61	50.00
62	75.00

■ **FIGURE 16.15 Retrieval based on a column that includes rows with nulls.**

The DBMS can't ascertain what the selling price for unsold items will be: maybe it will be less than $100, or maybe it will be greater than or equal to $100.

The policy of most DBMSs is to exclude rows with nulls from the result. For rows with null in the selling price column, the *maybe* answer to "Is selling price less than 100" becomes *false*. This seems pretty straightforward, but what happens when you have a complex logical expression of which one portion returns *maybe*? The operation of AND, OR, and NOT must be expanded to take into account that they may be operating on a *maybe*.

The three-valued logic table for AND can be found in Table 16.5. Notice that something important hasn't changed: the only way to get a true result is for both simple expressions linked by AND to be true. Given that most DBMSs exclude rows where the predicate evaluates to *maybe*, the presence of nulls in the data will not change what an end user sees.

Table 16.5 Three-Valued AND Truth Table

AND	True	False	Maybe
True	True	False	Maybe
False	False	False	False
Maybe	Maybe	False	Maybe

The same is true when you look at the three-valued truth table for OR (see Table 16.6). As long as one simple expression is true, it does not matter whether the second returns true, false, or maybe. The result will always be true.

Table 16.6 Three-Valued OR Truth Table

OR	True	False	Maybe
True	True	True	True
False	True	False	Maybe
Maybe	True	Maybe	Maybe

If you negate an expression that returns *maybe*, the NOT operator has no effect. In other words, NOT (MAYBE) is still *maybe*.

To see the rows that return *maybe*, you need to add an expression to your query that uses the IS NULL operator. For example, the easiest way to see which volumes have not been sold is to write a query like:

```
SELECT inventory_id, isbn, selling_price
FROM volume
WHERE selling_price IS NULL;
```

The result can be found in Figure 16.16. Note that the selling price column is empty in each row. (Remember that you typically can't see any special value for null.) Notice also that the rows in this result table are all those excluded from the query in Figure 16.15.

inventory_id	isbn	selling_price
7	978-1-11111-137-1	
8	978-1-11111-137-1	
9	978-1-11111-136-1	
10	978-1-11111-136-1	
16	978-1-11111-121-1	
17	978-1-11111-121-1	
27	978-1-11111-141-1	
28	978-1-11111-141-1	
29	978-1-11111-141-1	
30	978-1-11111-145-1	
31	978-1-11111-145-1	
32	978-1-11111-145-1	
43	978-1-11111-132-1	
44	978-1-11111-138-1	
45	978-1-11111-138-1	
46	978-1-11111-131-1	
47	978-1-11111-140-1	
48	978-1-11111-123-1	
49	978-1-11111-127-1	
63	978-1-11111-130-1	
64	978-1-11111-136-1	
65	978-1-11111-136-1	
66	978-1-11111-137-1	
67	978-1-11111-137-1	
68	978-1-11111-138-1	
69	978-1-11111-138-1	
70	978-1-11111-139-1	
71	978-1-11111-139-1	

■ **FIGURE 16.16 Using IS NULL to retrieve rows containing nulls.**

Four-Valued Logic

Codd's 330 rules for the relational data model include an enhancement to three-valued logic that he called ***four-valued logic***. In four-valued logic, there are actually two types of null: "null and it doesn't matter that it's null" and "null and we've really got a problem because it's null." For example, if a company sells internationally, then it probably has a column for the country of each customer. Because it is essential to know a customer's country, a null in the ***country*** column would fall into the category of "null and we've really got a problem." In contrast, a missing value in a ***company name*** column would be quite acceptable in a customer table for rows that represented individual customers. Then the null would be "null and it doesn't matter that it's null." Four-valued logic remains purely theoretical at this time, however, and hasn't been widely implemented. (In fact, as far as I know, only one DBMS uses four-valued logic: FirstSQL.)

Chapter 17 Retrieving Data from More Than One Table

As you learned in the first part of this book, logical relationships between entities in a relational database are represented by matching primary and foreign key values. Given that there are no permanent connections between tables stored in the database, a DBMS must provide some way for users to match primary and foreign key values when needed using the join operation.

In this chapter, you will be introduced to the syntax for including a join in a SQL query. Throughout this chapter you will also read about the impact joins have on database performance. At the end, you will see how subqueries (SELECTs within SELECTs) can be used to avoid joins and in some cases, significantly decrease the time it takes for a DBMS to complete a query.

SQL Syntax for Inner Joins

There are two types of syntax you can use for requesting the inner join of two tables. The first, which we have been calling the "traditional" join syntax, is the only way to write a join in the SQL standards through SQL-89. SQL-92 added a join syntax that is both more flexible and easier to use. (Unfortunately, it hasn't been implemented by some commonly used DBMSs.)

Traditional SQL Joins

The traditional SQL join syntax is based on the combination of the product and restrict operations that you read about in Chapter 6. It has the following general form:

```
SELECT columns
FROM table1, table2
WHERE table1.primary_key = table2.foreign_key
```

Listing the tables to be joined after FROM requests the product. The join condition in the WHERE clause's predicate requests the restrict that identifies the rows that are part of the joined tables. Don't forget that if you leave

off the join condition in the predicate, then the presence of the two tables after FROM simply generates a product table.

Note: If you really, really, really want a product, use the CROSS JOIN operator in the FROM clause.

Note: Even if you use the preceding syntax, it is highly unlikely today that the DBMS will actually perform a product followed by a restrict. Most current DMBSs have far faster means of processing a join. Nonetheless, a join is still just about the slowest common SQL operation. This is really unfortunate, because a relation database environment will, by its very nature, require a lot of joins.

For example, assume that someone wanted to see all the orders placed by a customer whose phone number is 518-555-1111. The phone number is part of the *customer* table; the purchase information is in the *sale* table. The two relations are related by the presence of the customer number in both (primary key of the *customer* table; foreign key in *sale*). The query to satisfy the information request, therefore, requires an equi-join of the two tables over the customer number, the result of which can be seen in Figure 17.1:

```
SELECT first_name, last_name, sale_id, sale_date
FROM customer, sale
WHERE customer.customer_numb = sale.customer_numb
      AND contact_phone = '518-555-1111';
```

```
first_name  | last_name | sale_id |     sale_date
------------+-----------+---------+--------------------
 Janice     | Jones     |       3 | 15-JUN-21 00:00:00
 Janice     | Jones     |      17 | 25-JUL-21 00:00:00
 Janice     | Jones     |       2 | 05-JUN-21 00:00:00
 Janice     | Jones     |       1 | 29-MAY-21 00:00:00
```

■ **FIGURE 17.1 Output from a query containing an equi-join between a primary key and a foreign key.**

There are two important things to notice about the preceding query:

- The join is between a primary key in one table and a foreign key in another. As you will remember, equi-joins that don't meet this pattern are frequently invalid.
- Because the *customer_numb* column appears in more than one table in the query, it must be qualified in the WHERE clause by the name of the table from which it should be taken. To add a qualifier, precede the name of a column by its name, separating the two with a period.

Note: With some large DBMSs, you must also qualify the names of tables you did not create with the user ID of the account that did create the table. For example, if user ID DBA created the customer table, then the full name

of the customer number column would the DBA.customer.customer_numb. Check your product documentation to determine whether your DBMS is one of those that requires the user ID qualifier.

How might a SQL query optimizer choose to process this query? Although we cannot be certain because there is more than one order of operations that will work, it is likely that the restrict operation to choose the customer with a telephone number of 518-555-1111 will be performed first. This cuts down on the amount of data that needs to be manipulated for the join. The second step probably will be the join operation because doing the project to select columns for display will eliminate the column needed for the join.

SQL-92 Join Syntax

The SQL-92 standard introduced an alternative join syntax that is both simpler and more flexible than the traditional join syntax. If you are performing a natural equi-join, there are three variations of the syntax you can use, depending on whether the column or columns over which you are joining have the same name and whether you want to use all matching columns in the join.

Note: Despite the length of time that has passed since the introduction of this revised join syntax, not all DBMSs support all three varieties of the syntax. You will need to consult the documentation of your particular product to determine exactly which syntax you can use.

Joins Over All Columns with the Same Name

When the primary key and foreign key columns you are joining have the same name and you want to use all matching columns in the join condition, all you need to do is indicate that you want to join the tables, using the following general syntax:

```
SELECT column(s)
FROM table1 NATURAL JOIN table2
```

The query we used as an example in the preceding section could therefore be written as

```
SELECT first_name, last_name, sale_id, sale_date
FROM customer NATURAL JOIN sale
WHERE contact_phone = '518-555-1111';
```

Note: Because the default is a natural equi-join, you will obtain the same result if you simply use JOIN instead of NATURAL JOIN. In fact, we rarely use the word NATURAL in queries.

The SQL command processor identifies all columns in the two tables that have the same name and automatically performs the join over those columns.

Joins Over Selected Columns

If you don't want to use all matching columns in a join condition, but the columns still have the same name, you specify the names of the columns over which the join is to be made by adding a USING clause:

```
SELECT column(s)
FROM table1 JOIN table2 USING (column)
```

Using this syntax, the sample query would be written

```
SELECT first_name, last_name, sale_id, sale_date
FROM customer JOIN sale USING (customer_numb)
WHERE contact_phone = '518-555-1111';
```

Joins Over Columns with Different Names

When the columns over which you are joining table don't have the same name, then you must use a join condition similar to that used in the traditional SQL join syntax:

```
SELECT column(s)
FROM table1 JOIN table2 ON join_condition
```

Assume that the rare book store database used the name *buyer_numb* in the *sale* table (rather than duplicate *customer_numb* from the customer table). In this case, the sample query will appear as

```
SELECT first_name, last_name, sale_id, sale_date
FROM customer JOIN sale
ON customer.customer_numb = sale.customer_numb
WHERE contact_phone = '518-555-1111';
```

Joining Using Concatenated Keys

All of the joins you have seen to this point have been performed using a single matching column. However, on occasion, you may run into tables where you are dealing with concatenated primary and foreign keys. As an example, we'll return to the four tables from the small accounting firm database that we used in Chapter 6 when we discussed how joins over concatenated keys work:

```
accountant (acct_first_name, acct_last_name, date_hired,
     office_ext)

customer (customer_numb, first_name, last_name, street,
     city, state_province, zip_postcode, contact_phone)

project (tax_year, customer_numb, acct_first_name,
     acct_last_name)

form (tax_year, customer_numb, form_id, is_complete)
```

To see which accountant worked on which forms during which year, a query needs to join the *project* and *form* tables, which are related by a concatenated primary key. The join condition needed is

```
project.tax_year || project.customer_numb =
     form.tax_year || form.customer_numb
```

The || operator represents concatenation in most SQL implementations. It instructs the SQL command processor to view the two columns as if they were one and to base its comparison on the concatenation rather than individual column values.

The following join condition produces the same result because it pulls rows from a product table where *both* the customer ID numbers and the tax years are the same:

```
project.tax_year = form.tax_year AND
     project.customer_numb = form.customer_numb
```

You can therefore write a query using the traditional SQL join syntax in two ways:

```
SELECT acct_first_name, acct_last_name, form.tax_year,
     form.form_ID
FROM project, form
WHERE project.tax_year || project.customer_numb =
     form.tax_year || form.customer_numb;
```

or

```
SELECT acct_first_name, acct_last_name, form.tax_year,
     form.form_ID
FROM project, form
WHERE project.tax_year = form.tax_year
     AND project.customer_numb = form.customer_numb;
```

If the columns have the same names in both tables and are the only matching columns, then the SQL-92 syntax

```
SELECT acct_first_name, acct_last_name, form.tax_year,
     form.form_ID
FROM project JOIN form;
```

has the same effect as the preceding two queries.

When the columns have the same names but aren't the only matching columns, then you must specify the columns in a USING clause:

```
SELECT acct_first_name, acct_last_name, form.tax_year,
     form.form_ID
FROM project JOIN form USING (tax_year, form_ID);
```

Alternatively, if the columns don't have the same name you can use the complete join condition, just as you would if you were using the traditional join syntax. For this example only, let's assume that the accounting firm prefaces each duplicated column name with an identifier for its table. The relations in the sample query would therefore look something like this:

```
account (acct_first_name, acct_last_name, …)
project (p_customer_numb, p_tax_year, …)
form (f_customer_numb, f_taxYear, form_ID, …)
```

The sample query would then be written:

```
SELECT acct_first_name, acct_last_name, form.f_tax_year,
     form.form_ID
FROM project JOIN form ON
     project.p_tax_year || project.p_customer_numb =
     form.f_tax_year || form.f_customer_numb;
```

or

```
SELECT acct_first_name, acct_last_name, form.f_tax_year,
      form.form_ID
FROM project JOIN form ON p_project.tax_year = f_form.tax_year
      AND p_project.customer_numb = f_form.customer_numb;
```

Notice that in all forms of the query, the tax year and form ID columns in the SELECT clause are qualified by a table name. It really doesn't matter from which the data are taken, but because the columns appear in both tables the SQL command processor needs to be told which pair of columns to use.

Joining More Than Two Tables

What if you need to join more than two tables in the same query? For example, someone at the rare book store might want to see the names of the people who have purchased a volume with the ISBN of 978-1-11111-146-1. The query that retrieves that information must join *volume* to *sale* to find the sales on which the volume was sold. Then, the result of the first join must be joined again to *customer* to gain access to the names.

Using the traditional join syntax, the query is written

```
SELECT first_name, last_name
FROM customer, sale, volume
WHERE volume.sale_id = sale.sale_id AND
      sale.customer_numb = customer.customer_numb
AND isbn = '978-1-11111-136-1';
```

With the simplest form of the SQL-92 syntax, the query becomes

```
SELECT first_name, last_name
FROM customer JOIN sale JOIN volume
WHERE isbn = '978-1-11111-136-1';
```

Both syntaxes produce the following result:

```
first_name | last_name
-----------+------------
Mary       | Collins
Janice     | Smith
```

Keep in mind that the join operation can work on only two tables at a time. If you need to join more than two tables, you must join them in pairs. Therefore, a join of three tables requires two joins, a join of four tables requires three joins, and so on.

SQL-92 Syntax and Multiple-Table Join Performance

Although the SQL-92 syntax is certainly simpler than the traditional join syntax, it has another major benefit: It gives you control over the order in which the joins are performed. With the traditional join syntax, the query optimizer is in complete control of the order of the *joins*. However, in SQL-92, the joins are performed from left to right, following the order in which the *joins* are placed in the FROM clause.

This means that you sometimes can affect the performance of a query by varying the order in which the joins are performed.[1] Remember that the less data the DBMS has to manipulate, the faster a query including one or more joins will execute. Therefore, you want to perform the most discriminatory joins first.

As an example, consider the sample query used in the previous section. The *volume* table has the most rows, followed by *sale* and then *customer*. However, the query also contains a highly discriminatory *restrict* predicate that limits the rows from that table. Therefore, it is highly likely that the DBMS will perform the restrict on *volume* first. This means that the query is likely to execute faster if you write it so that *sale* is joined with *volume* first, given that this join will significantly limit the rows from *sale* that need to be joined with *customer*.

[1]This holds true only if a DBMS has implemented the newer join syntax according to the SQL standard. A DBMS may support the syntax without its query optimizer using the order of tables in the FROM clause to determine join order.

In contrast, what would happen if there was no restrict predicate in the query and you wanted to retrieve the name of the customer for every book ordered in the database? The query would appear as

```
SELECT first_name, last_name
FROM customer JOIN sale JOIN volume;
```

First, keep in mind that this type of query, which is asking for large amounts of data, will rarely execute as quickly as one that contains predicates to limit the number of rows. Nonetheless, if will execute a bit faster if *customers* is joined to *sale* before joining to *volume*. Why? Because the joins manipulate fewer rows in that order.

Assume that there are 20 customers, 100 sales, and 300 volumes sold. Every sold item in *volume* must have a matching row in *sale*. Therefore, the result from that join will be at least 300 rows long. Those 300 rows must be joined to the 20 rows in *customer*. However, if we reverse the order, then the 20 rows in *customer* are joined to 100 rows in *sale*, producing a table of 100 rows, which can then be joined to *volume*. In either case, we are stuck with a join of 100 rows to 300 rows, but when the *customer* table is handled first, the other join is 20 to 100 rows, rather than 20 to 300 rows.

Finding Multiple Rows in One Table: Joining a Table to Itself

One of the limitations of a restrict operation is that its predicate is applied to only one row in a table at a time. This means that a predicate such as

```
isbn = '0-131-4966-9' AND isbn = '0-191-4923-8'
```

and the query

```
SELECT first_name, last_name
FROM customer JOIN sale JOIN volume
WHERE isbn = '978-1-11111-146-1'
      AND isbn = '978-1-11111-122-1';
```

will always return 0 rows. No row can have more than one value in the *isbn* column!

What the preceding query is actually trying to do is locate customers who have purchased two specific books. This means that there must be at least two rows for a customer's purchases in *volume*, one for each for each of the books in question.

Given that you cannot do this type of query with a simple restrict predicate, how can you retrieve the data? The technique is to join the *volume* table to itself over the sale ID. The result table will have two columns for the book's ISBN, one for

each copy of the original table. Those rows that have both the ISBNs that we want will finally be joined to the *sale* table (over the sale ID) and *customer* (over customer number) tables so that the query can project the customer's name.

Before looking at the SQL syntax, however, let's examine the relational algebra of the joins so you can see exactly what is happening. Assume that we are working with the subset of the *volume* table in Figure 17.2. (The sale ID and the ISBN are the only columns that affect the relational algebra; the rest have been left off for simplicity.) Notice first that the result of our sample query should display the first and last names of the customer who made purchase number 6. (It is the only order that contains both of the books in question.)

sale_id	isbn
1	978-1-11111-111-1
1	978-1-11111-133-1
1	978-1-11111-131-1
2	978-1-11111-142-1
2	978-1-11111-144-1
2	978-1-11111-146-1
3	978-1-11111-133-1
3	978-1-11111-132-1
3	978-1-11111-143-1
4	978-1-11111-121-1
5	978-1-11111-121-1
6	978-1-11111-139-1
6	978-1-11111-146-1
6	978-1-11111-122-1
6	978-1-11111-130-1
6	978-1-11111-126-1
7	978-1-11111-125-1
7	978-1-11111-131-1
8	978-1-11111-126-1
8	978-1-11111-133-1
9	978-1-11111-139-1
10	978-1-11111-133-1

■ **FIGURE 17.2 A subset of the *volume* table.**

The first step in the query is to join the table in Figure 17.2 to itself over the sale ID, producing the result table in Figure 17.3. The columns that come from the first copy have been labeled T1; those that come from the second copy are labeled T2.

The two rows in black are those that have the ISBNs for which we are searching. Therefore, we need to follow the join with a restrict that says something like

```
WHERE isbn (from table 1) = '978-1-11111-146-1'
     AND isbn (from table 2) = '978-1-11111-122-1'
```

The result will be a table with one row in it (the second of the two black rows in Figure 17.3).

sale_id (T1)	isbn	sale_id (T2)	isbn
1	978-1-11111-111-1	1	978-1-11111-133-1
1	978-1-11111-111-1	1	978-1-11111-131-1
1	978-1-11111-111-1	1	978-1-11111-111-1
1	978-1-11111-131-1	1	978-1-11111-133-1
1	978-1-11111-131-1	1	978-1-11111-131-1
1	978-1-11111-131-1	1	978-1-11111-111-1
1	978-1-11111-133-1	1	978-1-11111-133-1
1	978-1-11111-133-1	1	978-1-11111-131-1
1	978-1-11111-133-1	1	978-1-11111-111-1
2	978-1-11111-142-1	2	978-1-11111-144-1
2	978-1-11111-142-1	2	978-1-11111-146-1
2	978-1-11111-142-1	2	978-1-11111-142-1
2	978-1-11111-146-1	2	978-1-11111-144-1
2	978-1-11111-146-1	2	978-1-11111-146-1
2	978-1-11111-146-1	2	978-1-11111-142-1
2	978-1-11111-144-1	2	978-1-11111-144-1
2	978-1-11111-144-1	2	978-1-11111-146-1
2	978-1-11111-144-1	2	978-1-11111-142-1
3	978-1-11111-143-1	3	978-1-11111-133-1
3	978-1-11111-143-1	3	978-1-11111-132-1
3	978-1-11111-143-1	3	978-1-11111-143-1
3	978-1-11111-132-1	3	978-1-11111-133-1
3	978-1-11111-132-1	3	978-1-11111-132-1
3	978-1-11111-132-1	3	978-1-11111-143-1
3	978-1-11111-133-1	3	978-1-11111-133-1
3	978-1-11111-133-1	3	978-1-11111-132-1
3	978-1-11111-133-1	3	978-1-11111-143-1
5	978-1-11111-121-1	5	978-1-11111-121-1
4	978-1-11111-121-1	4	978-1-11111-121-1
6	978-1-11111-146-1	6	978-1-11111-139-1
6	978-1-11111-146-1	6	978-1-11111-126-1
6	978-1-11111-146-1	6	978-1-11111-130-1

■ **FIGURE 17.3 The result of joining the table in Figure 17.2 to itself (continues).**

At this point, the query can join the table to *sale* over the sale ID to provide access to the customer number of the person who made the purchase. The result of that second join can then be joined to *customer* to obtain the customer's name (Franklin Hayes). Finally, the query projects the columns the user wants to see.

Correlation Names

The challenge facing a query that needs to work with multiple copies of a single table is to tell the SQL command processor to make the copies of the table. We do this by placing the name of the table more than once on the FROM line, associating each instance of the name with a different alias. Such aliases for table names are known as *correlation names* and take the syntax

```
FROM table_name AS correlation_name
```

6 | 978-1-11111-146-1 |	6 | 978-1-11111-122-1
6 | 978-1-11111-146-1 |	6 | 978-1-11111-146-1
6 | 978-1-11111-122-1 |	6 | 978-1-11111-139-1
6 | 978-1-11111-122-1 |	6 | 978-1-11111-126-1
6 | 978-1-11111-122-1 |	6 | 978-1-11111-130-1
6 | 978-1-11111-122-1 |	6 | 978-1-11111-122-1
6 | 978-1-11111-122-1 |	6 | 978-1-11111-146-1
6 | 978-1-11111-130-1 |	6 | 978-1-11111-139-1
6 | 978-1-11111-130-1 |	6 | 978-1-11111-126-1
6 | 978-1-11111-130-1 |	6 | 978-1-11111-130-1
6 | 978-1-11111-130-1 |	6 | 978-1-11111-122-1
6 | 978-1-11111-130-1 |	6 | 978-1-11111-146-1
6 | 978-1-11111-126-1 |	6 | 978-1-11111-139-1
6 | 978-1-11111-126-1 |	6 | 978-1-11111-126-1
6 | 978-1-11111-126-1 |	6 | 978-1-11111-130-1
6 | 978-1-11111-126-1 |	6 | 978-1-11111-122-1
6 | 978-1-11111-126-1 |	6 | 978-1-11111-146-1
6 | 978-1-11111-139-1 |	6 | 978-1-11111-139-1
6 | 978-1-11111-139-1 |	6 | 978-1-11111-126-1
6 | 978-1-11111-139-1 |	6 | 978-1-11111-130-1
6 | 978-1-11111-139-1 |	6 | 978-1-11111-122-1
6 | 978-1-11111-139-1 |	6 | 978-1-11111-146-1
7 | 978-1-11111-125-1 |	7 | 978-1-11111-131-1
7 | 978-1-11111-125-1 |	7 | 978-1-11111-125-1
7 | 978-1-11111-131-1 |	7 | 978-1-11111-131-1
7 | 978-1-11111-131-1 |	7 | 978-1-11111-125-1
8 | 978-1-11111-126-1 |	8 | 978-1-11111-133-1
8 | 978-1-11111-126-1 |	8 | 978-1-11111-126-1
8 | 978-1-11111-133-1 |	8 | 978-1-11111-133-1
8 | 978-1-11111-133-1 |	8 | 978-1-11111-126-1
9 | 978-1-11111-139-1 |	9 | 978-1-11111-139-1
10 | 978-1-11111-133-1 |	10 | 978-1-11111-133-1

■ **FIGURE 17.3** (***cont.***)

For example, to instruct SQL to use two copies of the *volume* table you might use

```
FROM volume AS T1, volume AS T2
```

The AS is optional. Therefore, the following syntax is also legal:

```
FROM volume T1, volume T2
```

In the other parts of the query, you refer to the two copies using the correlation names rather than the original table name.

Note: You can give any table a correlation name; its use is not restricted to queries that work with multiple copies of a single table. In fact, if a table name is difficult to type and appears several times in a query, you can save yourself some typing and avoid problems with typing errors by giving the table a short correlation name.

Performing the Same-Table Join

The query that performs the same-table join needs to specify all of the relational algebra operations you read about in the preceding section. It can be written using the traditional join syntax, as follows:

```
SELECT first_name, last_name
FROM volume T1, volume T2, sale, customer
WHERE t1.isbn = '978-1-11111-146-1'
      AND T2.isbn = '978-1-11111-122-1'
      AND T1.sale_id = T2.sale_id
      AND T1.sale_id = sale.sale_id
      AND sale.customer_numb = customer.customer_numb;
```

There is one very important thing to notice about this query. Although our earlier discussion of the relational algebra indicated that the same-table join would be performed first, followed by a restrict and the other two joins, there is no way using the traditional syntax to indicate the joining of an intermediate result table (in this case, the same-table join). Therefore, the query syntax must join *sale* to either T1 or T2. Nonetheless, it is likely that the query optimizer will determine that performing the same-table join followed by the restrict is a more efficient way to process the query than joining *sale* to T1 first.

If you use the SQL-92 join syntax, then you have some control over the order in which the joins are performed:

```
SELECT first_name, last_name
FROM volume T1 JOIN volume T2 ON (T1.sale_id = T2.sale_id)
      JOIN sale JOIN customer
WHERE t1.isbn = '978-1-11111-146-1'
      AND T2.isbn = '978-1-11111-122-1';
```

The SQL command processor will process the multiple joins in the FROM clause from left to right, ensuring that the same-table join is performed first.

You can extend the same table join technique you have just read about to find as many rows in a table you need. Create one copy of the table with a correlation name for the number of rows the query needs to match in the FROM clause and join those tables together. In the WHERE clause, use a predicate that includes one restrict for each copy of the table. For example, to retrieve data that have four specified rows in a table, you need four copies of the table, three joins, and four expressions in the restrict predicate. The general format of such a query is

```
SELECT column(s)
FROM table_name T1 JOIN table_name T2
      JOIN table_name T3 JOIN table_name T4
WHERE T1.column_name = value AND T2.column_name = value
      AND T3.column_name = value
      AND T3.column_name = value
```

Outer Joins

As you read in Chapter 6, an outer join is a join that includes rows in a result table even though there may not be a match between rows in the two tables being joined. Whenever the DBMS can't match rows, it places nulls in the columns for which no data exist. The result may therefore not be a legal relation because it may not have a primary key. However, because a query's result table is a virtual table that is never stored in the database, having no primary keys doesn't present a data integrity problem.

To perform an outer join using the SQL-92 syntax, you indicate the type of join in the FROM clause. For example, to perform a left outer join between the *customer* and *sale* tables, you could type

```
SELECT first_name, last_name, sale_id, sale_date
FROM customer LEFT OUTER JOIN sale;
```

The result appears in Figure 17.4. Notice that five rows appear to be empty in the *sale_id* and *sale_date* columns. These five customers haven't

first_name	last_name	sale_id	sale_date
Janice	Jones	1	29-MAY-21 00:00:00
Janice	Jones	2	05-JUN-21 00:00:00
Janice	Jones	17	25-JUL-21 00:00:00
Janice	Jones	3	15-JUN-21 00:00:00
Jon	Jones	20	01-SEP-21 00:00:00
Jon	Jones	16	25-JUL-21 00:00:00
Jon	Jones	13	10-JUL-21 00:00:00
John	Doe		
Jane	Doe	4	30-JUN-21 00:00:00
Jane	Smith	18	22-AUG-21 00:00:00
Jane	Smith	8	07-JUL-21 00:00:00
Janice	Smith	19	01-SEP-21 00:00:00
Janice	Smith	14	10-JUL-21 00:00:00
Janice	Smith	5	30-JUN-21 00:00:00
Helen	Brown		
Helen	Jerry	9	07-JUL-21 00:00:00
Helen	Jerry	7	05-JUL-21 00:00:00
Mary	Collins	11	10-JUL-21 00:00:00
Peter	Collins	12	10-JUL-21 00:00:00
Edna	Hayes	15	12-JUL-21 00:00:00
Edna	Hayes	10	10-JUL-21 00:00:00
Franklin	Hayes	6	05-JUL-21 00:00:00
Peter	Johnson		
Peter	Johnson		
John	Smith		

■ **FIGURE 17.4 The result of an outer join.**

made any purchases. Therefore, the columns in question are actually null. However, most DBMSs have no visible indicator for null; it looks as if the values are blank. It is the responsibility of the person viewing the result table to realize that the empty spaces represent nulls rather than blanks.

The SQL-92 outer join syntax for joins has the same options as the inner join syntax:

- If you use the syntax in the preceding example, the DBMS will automatically perform the outer join on all matching columns between the two tables.
- If you want to specify the columns over which the outer join will be performed, and the columns have the same names in both tables, add a USING clause:

```
SELECT first_name, last_name, sale_id, sale_date
FROM customer LEFT OUTER JOIN sale USING (customer_numb);
```

- If the columns over which you want to perform the outer join do not have the same name, then append an ON clause that contains the join condition:

```
SELECT first_name, last_name
FROM customer T1 LEFT OUTER JOIN sale T2
    ON (T1.customer_numb = T2.customer_numb);
```

Note: The SQL standard also includes an operation known as the UNION JOIN. It performs a FULL OUTER JOIN on two tables and then throws out the rows that match, placing all those that don't match in the result table. The UNION JOIN hasn't been widely implemented.

Table Constructors in Queries

SQL standards from SQL-92 forward allow the table on which a SELECT is performed to be a virtual table rather than just a base table. This means that a DBMS should allow a complete SELECT (what is known as a *subquery*) to be used in a FROM clause to prepare the table on which the remainder of the query will operate. Expressions that create tables for use in SQL statements in this way are known as *table constructors*.

Note: When you join tables in the FROM clause you are actually generating a source for a query on the fly. What is described in this section is just an extension of that principle.

For example, the following query lists all volumes that were purchased by customers 6 and 10:

```
SELECT isbn, first_name, last_name
FROM volume JOIN (SELECT first_name, last_name, sale_id
   FROM sale JOIN customer
WHERE customer.customer_numb = 6 or customer.customer_numb = 10)
```

The results can be found in Figure 17.5. Notice that the row selection is being performed in the subquery that is part of the FROM clause. This forces the SQL command processor to perform the subquery prior to performing the join in the outer query. Although this query could certainly be written in another way, using the subquery in the FROM clause gives a programmer using a DBMS with a query optimizer that uses the FROM clause order additional control over the order in which the relational algebra operations are performed.

```
       isbn        | first_name | last_name
-------------------+------------+-----------
 978-1-11111-121-1 | Janice     | Smith
 978-1-11111-130-1 | Peter      | Collins
 978-1-11111-132-1 | Peter      | Collins
 978-1-11111-141-1 | Janice     | Smith
 978-1-11111-141-1 | Janice     | Smith
 978-1-11111-128-1 | Janice     | Smith
 978-1-11111-136-1 | Janice     | Smith
```

■ **FIGURE 17.5 Using a table constructor in a query's FROM clause.**

Avoiding Joins with Uncorrelated Subqueries

As we discussed earlier in this chapter, with some DBMSs you can control the order in which joins are performed by using the SQL-92 syntax and being careful with the order in which you place joins in the FROM clause. However, there is a type of SQL syntax—a *subquery*—that you can use with any DBMS to obtain the same result, but often avoid performing a join altogether.[2]

A subquery (or *subselect*) is a complete SELECT statement embedded within another SELECT. The result of the inner SELECT becomes data used by the outer.

Note: Subqueries have other uses besides avoiding joins, which you will see throughout the rest of this book.

[2]Even a subquery may not avoid joins. Some query optimizers actually replace subqueries with joins when processing a query.

A query containing a subquery has the following general form:

```
SELECT column(s)
FROM table
WHERE operator (SELECT column(s))
      FROM table
      WHERE …)
```

There are two general types of subqueries. In an *uncorrelated subquery*, the SQL command processor is able to complete the processing of the inner SELECT before moving to the outer. However, in a *correlated subquery*, the SQL command processor cannot complete the inner query without information from the outer. Correlated subqueries usually require that the inner SELECT be performed more than once and therefore can execute relatively slowly. The same is not true for uncorrelated subqueries which can be used to replace join syntax and therefore may produce faster performance.

Note: You will see examples of correlated subqueries beginning in Chapter 18.

Using the IN Operator

As a first example, consider the following query

```
SELECT sale_date, customer_numb
FROM sale JOIN volume
WHERE isbn = '978-1-11111-136-1';
```

which produces the following output:

```
  sale_date          | customer_numb
---------------------+---------------
 10-JUL-21 00:00:00 |             9
 01-SEP-21 00:00:00 |             6
```

When looking at the preceding output, don't forget that by default the DBMS adds the time to the display of a date column. In this case, there is no time included in the stored data, so the time appears as all zeros.

We can rewrite the query using subquery syntax as

```
SELECT sale_date, customer_numb
FROM sale
WHERE sale_id IN (SELECT sale_id
      FROM volume
      WHERE isbn = '978-1-11111-136-1');
```

The inner SELECT retrieves data from the *volume* table, and produces a set of sale IDs. The outer SELECT then retrieves data from *sale* where the sale ID is in the set of values retrieved by the subquery.

The use of the IN operator is actually exactly the same as the use you read about in Chapter 16. The only difference is that, rather than placing the set of values in parentheses as literals, the set is generated by a SELECT.

When processing this query, the DBMS never joins the two tables. It performs the inner SELECT first and then uses the result table from that query when processing the outer SELECT. In the case in which the two tables are very large, this can significantly speed up processing the query.

Note: You can also use NOT IN with subqueries. This is a very powerful syntax that you will read about in Chapter 18.

Using the ANY Operator

Like IN, the ANY operator searches a set of values. In its simplest form, ANY is equivalent to IN:

```
SELECT sale_date, customer_numb
FROM sale
WHERE sale_id = ANY (SELECT sale_id
      FROM volume
      WHERE isbn = '978-1-11111-136-1');
```

This syntax tells the DBMS to retrieve rows from *sale*, where the sale ID is "equal to any" of those retrieved by the SELECT in the subquery.

What sets ANY apart from IN is that the = can be replaced with any other relationship operator (for example, < and >). For example, you could use it to create a query that asked for all customers who had purchased a book with a price greater than the average cost of a book. Because queries of this type require the use of SQL summary functions, we will leave their discussion until Chapter 19.

Nesting Subqueries

The SELECT that you use as a subquery can have a subquery. In fact, if you want to rewrite a query that joins more than two tables, you will need to nest subqueries in this way. As an example, consider the following query that you saw earlier in this chapter:

```
SELECT first_name, last_name
FROM customer, sale, volume
WHERE volume.sale_id = sale.sale_id AND
      sale.customer_numb = customer.customer_numb
      AND isbn = '978-1-11111-136-1';
```

It can be rewritten as

```
SELECT first_name, last_name
FROM customer
WHERE customer_numb IN
      (SELECT customer_numb
      FROM sale
      WHERE sale_id = ANY
            (SELECT sale_id
            FROM volume
            WHERE isbn = '978-1-11111-136-1'));
```

Note that each subquery is surrounded completely by parentheses. The end of the query therefore contains two closing parentheses next to each other. The rightmost) closes the outer subquery; the) to its left closes the inner subquery.

The DBMS processes the innermost subquery first, returning a set of sale IDs that contains the sales on which the ISBN in question appears. The middle SELECT (the outer subquery) returns a set of customer numbers for rows where the sale ID is any of those in the set returned by the innermost subquery. Finally, the outer query displays information about customers whose customer numbers are in the set produced by the outer subquery.

In general, the larger the tables in question (in other words, the more rows they have), the more performance benefit you will see if you assemble queries using subqueries rather than joins. How many levels deep can you nest subqueries? There is no theoretical limit. However, once a query becomes more than a few levels deep, it may become hard to keep track of what is occurring.

Replacing a Same-Table Join with Subqueries

The same-table join that you read about earlier in this chapter can also be replaced with subqueries. As you will remember, that query required a join between *sale* and *customer* to obtain the customer name, a join between *sale* and *volume*, and a join of the *volume* table to itself to find all sales that contained two desired ISBNs. Because there were three joins in the original query, the rewrite will require one nested subquery for each join.

```
SELECT last_name, first_name
FROM customer
WHERE customer_numb IN
      (SELECT customer_numb
      FROM sale
      WHERE sale_id IN
            (SELECT sale_id
            FROM volume
            WHERE isbn = '978-1-11111-146-1'
            AND sale_id IN
                  (SELECT sale_id
                  FROM volume
                  WHERE isbn = '978-1-11111-122-1')));
```

The innermost subquery retrieves a set of sale IDs for the rows on which an ISBN of '978-1-11111-122-1' appears. The next level subquery above it retrieves rows from *volume* where the sale ID appears in the set retrieved by the innermost subquery and the ISBN is '978-1-11111-146-1'. These two subqueries, therefore, replace the same-table join.

The set of sale IDs is then used by the outermost subquery to obtain a set of customer numbers for the sales whose numbers appear in the result set of the two innermost subqueries. Finally, the outer query displays customer information for the customers whose numbers are part of the outermost subquery's result set.

Notice that the two innermost subqueries are based on the same table. To process this query, the DBMS makes two passes through the *volume* table—one for each subquery—rather than joining a copy of the table to itself. When a table is very large, this syntax can significantly speed up performance because the DBMS does not need to create and manipulate a duplicate copy of the large table in main memory.

Page left intentionally blank

Chapter 18 Advanced Retrieval Operations

To this point, the queries you have read about combine and extract data from relations in relatively straightforward ways. However, there are additional operations you can perform on relations that, for example, answer questions such as "show me the data that are not …" or "show me the combination of data that are …". In this chapter, you will read about the implementation of additional relational algebra operations in SQL that will perform such queries, as well as performing calculations and using functions that you can use to obtain information about the data you retrieve.

Union

Union is one of the few relational algebra operations whose name can be used in a SQL query. When you want to use a union, you write two individual SELECT statements joined by the keyword UNION:

```
SELECT column(s)
FROM table(s)
WHERE predicate
UNION
SELECT column(s)
FROM table(s)
WHERE predicate
```

The columns retrieved by the two SELECTs must have the same data types and sizes and be in the same order. For example, the following is legal as long as the customer numbers are the same data type (for example, integer), and the customer names are the same data type and length (for example, 30-character strings):

```
SELECT customer_numb, customer_first, customer_last
FROM some_table
UNION
SELECT cust_no, first_name, last_name
FROM some_other_table
```

Notice that the source tables of the two SELECTs don't need to be the same, nor do the columns need to have the same names. However, the following is not legal:

```
SELECT customer_first, customer_last
FROM some_table
UNION
SELECT cust_no, cust_phone
FROM some_table
```

Although both SELECTS are taken from the same table and the two base tables are therefore union compatible, the result tables returned by the two SELECTs are *not* union compatible[1] and the union therefore cannot be performed. The *cust_no* column has a domain of INT and therefore doesn't match the CHAR domain of the *customer_first* column. The *customer_last* and *cust_phone* columns do have the same data type, but they don't have the same size: *customer_last* has space for more characters. If even one corresponding set of columns don't match, the tables aren't union compatible.

Performing Union Using the Same Source Tables

A typical use of UNION in interactive SQL is a replacement for a predicate with an OR. As an example, consider this query:

```
SELECT first_name, last_name
FROM customer JOIN sale JOIN volume
WHERE isbn = '978-1-11111-128-1'
UNION
SELECT first_name, last_name
FROM customer JOIN sale JOIN volume
WHERE isbn = '978-1-11111-143-1';
```

It produces the following output:

```
first_name  | last_name
-----------+-----------
Janice      | Jones
Janice      | Smith
```

[1]Don't forget that SQL's definition of union compatibility is different from the relational algebra definition. SQL unions require that the tables have the same number of columns. The columns must match in data type and in size. Relational algebra, however, requires that the columns be defined over the same domains, without regard to size (which is an implementation detail.)

The DBMS processes the query by performing the two SELECTs. It then combines the two individual result tables into one, eliminating duplicate rows. To remove the duplicates, the DBMS sorts the result table by every column in the table and then scans it for matching rows placed next to one another. (That is why the rows in the result are in alphabetical order by the author's first name.) The information returned by the preceding query is the same as the following:

```
SELECT first_name, last_name
FROM customer JOIN sale JOIN volume
WHERE isbn = '978-1-11111-128-1'
      OR isbn = '978-1-11111-143-1';
```

However, there are two major differences. First, when you use the complex predicate that contains OR, most DBMSs retain the duplicate rows. In contrast, the query with the UNION operator removes them automatically.

The second difference is in how the queries are processed. The query that performs a union makes two passes through the *volume* table, one for each of the individual SELECTs, making only a single comparison with the ISBN value in each row. The query that uses the OR in its predicate makes only one pass through the table, but must make two comparisons when testing most rows.[2]

Which query will execute faster? If you include a DISTINCT in the query with an OR predicate, then it will return the same result as the query that performs a union. However, if you are using a DBMS that does not remove duplicates automatically and you can live with the duplicate rows, then the query with the OR predicate will be faster.

Note: If you want a union to retain all rows—including the duplicates—use UNION ALL instead of UNION.

Performing Union Using Different Source Tables

Another common use of UNION is to pull together data from different source tables into a single result table. Suppose, for example, we wanted to obtain a list of books published by Wiley and books that have been purchased by customer number 11. A query to obtain this data can be written as

[2]Some query optimizers do not behave in this way. You will need to check with either a DBA or a system programmer (someone who knows a great deal about the internals of your DBMS) to find out for certain.

```
SELECT author_last_first, title
FROM work, book, author, publisher
WHERE work.author_numb = author.author_numb
      AND work.work_numb = book.work_numb
      AND book.publisher_id = publisher.publisher_id
      AND publisher_name = 'Wiley'
UNION
SELECT author_last_first, title
FROM work, book, author, sale, volume
WHERE customer_numb = 11
      AND work.author_numb = author.author_numb
      AND work.work_numb = book.work_numb
      AND book.isbn = volume.isbn
      AND volume.sale_id = sale.sale_id;
```

To process this query, the result of which appear in Figure 18.1, the DBMS performs each separate SELECT and then combines the individual result tables.

```
          author_last_first       |             title
----------------------------------+-----------------------------------
 Barth, John                      | Giles Goat Boy
 Bronte, Charlotte                | Jane Eyre
 Funke, Cornelia                  | Inkdeath
 Rand, Ayn                        | Anthem
 Rand, Ayn                        | Atlas Shrugged
 Twain, Mark                      | Adventures of Huckleberry Finn, The
 Twain, Mark                      | Tom Sawyer
```

■ **FIGURE 18.1 The result of a union between result tables coming from different source tables.**

Alternative SQL-92 Union Syntax

The SQL-92 standard introduced an alternative means of making two tables union compatible: the CORRESPONDING BY clause. This syntax can be used when the two source tables have some columns with the same names. However, the two source tables need not have completely the same structure.

To use CORRESPONDING BY, you SELECT * from each of the source tables, but then indicate the columns to be used for the union in the CORRESPONDING BY clause:

```
SELECT *
FROM table1
WHERE predicate
UNION CORRESPONDING BY (columns_for_union)
SELECT *
FROM table2
WHERE predicate
```

For example, the query to retrieve the names of all customers who have ordered two specific books could be rewritten

```
SELECT *
FROM volume JOIN sale JOIN customer
WHERE isbn = '978-1-11111-128-1'
UNION CORRESPONDING BY (first_name, last_name)
SELECT *
FROM volume JOIN sale JOIN customer
WHERE isbn = '978-1-11111-128-1';
```

To process this query, the DBMS performs the two SELECTs, returning all columns in the tables. However, when the time comes to perform the union, it throws away all columns except those in the parentheses following BY.

Negative Queries

Among the most powerful database queries are those phrased in the negative, such as "show me all the customers who have not made a purchase in the past year." This type of query is particularly tricky, because it is asking for data that are not in the database. (The rare book store has data about customers who *have* purchased, but not those who have not.) The only way to perform such a query is to request the DBMS to use the difference operation.

Traditional SQL Negative Queries

The traditional way to perform a query that requires a difference is to use subquery syntax with the NOT IN operator. To do so, the query takes the following general format:

```
SELECT column(s)
FROM table(s)
WHERE column NOT IN (SELECT column
      FROM table(s)
      WHERE predicate)
```

The outer query retrieves a list of all things of interest; the subquery retrieves those that meet the necessary criteria. The NOT IN operator then acts to include all those from the list of all things that *are not* in the set of values returned by the subquery.

As a first example, consider the query that retrieves all books that are not in stock (no rows exist in *volume*):

```
SELECT title
FROM book, work
WHERE book.work_numb = work.work_numb
      AND isbn NOT IN (SELECT isbn
      FROM volume);
```

The outer query selects those rows in *books* (the list of all things) whose ISBNs are not in *volume* (the list of things that *are*). The result in Figure 18.2 contains the nine books that do not appear at least once in the *volume* table.

```
title
--------------------------------------------------
 Jane Eyre
 Villette
 Hound of the Baskervilles
 Lost World, The
 Complete Sherlock Holmes
 Complete Sherlock Holmes
 Tom Sawyer
 Connecticut Yankee in King Arthur's Court, A
 Dune
```

■ **FIGURE 18.2 The result of the first SELECT that uses a NOT IN subquery.**

As a second example, we will retrieve the titles of all books for which we don't have a new copy in stock, the result of which can be found in Figure 18.3:

```
SELECT title
FROM work, book
WHERE work.work_numb = book.work_numb
AND book.isbn NOT IN (SELECT isbn
      FROM volume
      WHERE condition_code = 1);
```

In this case, the subquery contains a restrict predicate in its WHERE clause, limiting the rows retrieved by the subquery to new volumes (those with a

title	isbn
Jane Eyre	978-1-11111-111-1
Jane Eyre	978-1-11111-112-1
Villette	978-1-11111-113-1
Hound of the Baskervilles	978-1-11111-114-1
Hound of the Baskervilles	978-1-11111-115-1
Lost World, The	978-1-11111-116-1
Complete Sherlock Holmes	978-1-11111-117-1
Complete Sherlock Holmes	978-1-11111-118-1
Tom Sawyer	978-1-11111-120-1
Connecticut Yankee in King Arthur's Court, A	978-1-11111-119-1
Tom Sawyer	978-1-11111-121-1
Adventures of Huckleberry Finn, The	978-1-11111-122-1
Matarese Circle, The	978-1-11111-123-1
Bourne Supremacy, The	978-1-11111-124-1
Fountainhead, The	978-1-11111-125-1
Atlas Shrugged	978-1-11111-127-1
Kidnapped	978-1-11111-128-1
Treasure Island	978-1-11111-130-1
Sot Weed Factor, The	978-1-11111-131-1
Dune	978-1-11111-134-1
Foundation	978-1-11111-135-1
Last Foundation	978-1-11111-137-1
I, Robot	978-1-11111-139-1
Inkheart	978-1-11111-140-1
Anthem	978-1-11111-144-1

■ **FIGURE 18.3 The result of the second SELECT that uses a NOT IN subquery.**

condition code value of 1). The outer query then copies a book to the result table if the ISBN is *not* in the result of the subquery.

Notice that in both of the sample queries there is no explicit syntax to make the two tables union compatible, something required by the relational algebra *difference* operation. However, the outer query's WHERE clause contains a predicate that compares a column taken from the result of the outer query with the same column taken from the result of the subquery. These two columns represent the union compatible tables.

As a final example, consider a query that retrieves the names of all customers who have not made a purchase after 1-Aug-2021. When you are putting together a query of this type, your first thought might be to write the query as follows:

```
SELECT first_name, last_name
FROM customer JOIN sale
WHERE sale_date < '1-Aug-2021';
```

This query, however, won't work as you intend. First of all, the join eliminates all customers who have no purchases in the *sale* table, even though they should be included in the result. Second, the retrieval predicate identifies those customers who placed orders prior to 1-Aug-2021, but says nothing about who may or may not have made a purchase after that date. Customers may have made a purchase prior to 1-Aug-2021, on 1-Aug-2021, after 1-Aug-2021, or any combination of the preceding.

The typical way to perform this query correctly is to use a difference: the difference between all customers and those who *have* made a purchase after 1-Aug-2021. The query—the result of which can be found in Figure 18.4—appears as follows:

```
SELECT first_name, last_name
FROM customer
WHERE customer_numb NOT IN (SELECT customer_numb
      FROM sale
      WHERE sale_date >= '1-Aug-2021')
```

first_name	last_name
Janice	Jones
John	Doe
Jane	Doe
Helen	Brown
Helen	Jerry
Mary	Collins
Peter	Collins
Edna	Hayes
Franklin	Hayes
Peter	Johnson
Peter	Johnson
John	Smith

■ **FIGURE 18.4 The result of the third query using a NOT IN subquery.**

Negative Queries Using the EXCEPT Operator

The SQL-92 standard added an operator—EXCEPT—that performs a difference operation directly between two union compatible tables. Queries using EXCEPT look very much like a union:

```
SELECT first_name, last_name
FROM customer
EXCEPT
SELECT first_name, last_name
FROM customer, sale
WHERE customer.customer_numb = sale.customer_numb
AND sale_date >= '1-Aug-2021';
```

or

```
SELECT *
FROM customer
EXCEPT CORRESPONDING BY (first_name, last_name)
SELECT *
FROM customer, sale
WHERE customer.customer_numb = sale.customer_numb
AND sale_date >= '1-Aug-2021';
```

Using the first syntax, you include two complete SELECT statements that are joined by the keyword EXCEPT. The SELECTs must return union compatible tables. The first SELECT retrieves a list of all things (in this example, all customers); the second retrieves the things that *are* (in this example, customers with sales after 1-Aug-2021). The EXCEPT operator then removes all rows from the first table that appear in the second.

The second syntax retrieves all columns from both source tables but uses the CORRESPONDING BY clause to project the columns to make the two tables union compatible.

The EXISTS Operator

The EXISTS operator check the number of rows returned by a subquery. If the subquery contains one or more rows, then the result is true and a row is placed in the result table; otherwise, the result is false and no row is added to the result table.

For example, suppose the rare book store wants to see the titles of books that have been sold. To write the query using EXISTS, you would use

```
SELECT title
FROM book t1, work
WHERE t1.work_numb = work.work_numb
AND EXISTS (SELECT *
      FROM volume
      WHERE t1.isbn = volume.isbn
      AND selling_price > 0);
```

The preceding is a *correlated subquery*. Rather than completing the entire subquery and then turning to the outer query, the DBMS processes the query in the following manner:

1. Look at a row in *book*.
2. Use the ISBN from that row in the subquery's WHERE clause.
3. If the subquery finds at least one row in *volume* with the same ISBN, place a row in the intermediate result table. Otherwise, do nothing.
4. Repeat steps 1 through 3 for all rows in the *book* table.
5. Join the intermediate result table to *work*.
6. Project the *title* column.

The important thing to recognize here is that the DBMS repeats the subquery for every row in *book*. It is this repeated execution of the subquery that makes this a correlated subquery.

When you are using the EXISTS operator, it doesn't matter what follows SELECT in the subquery. EXISTS is merely checking to determine whether any rows are present in the subquery's result table. Therefore, it is easiest simply to use * rather than to specify individual columns.[3]

How will this query perform? It will probably perform better than a query that joins *book* and *volume*, especially if the two tables are large. If you were to write the query using an IN subquery—

```
SELECT title
FROM work, book
WHERE work.work_numb = book.work_numb
      AND isbn IN (SELECT isbn
      FROM volume);
```

—you would be using an uncorrelated subquery that returned a set of ISBNs that the outer query searches. The more rows returned by the uncorrelated subquery, the closer the performance of the EXISTS and IN queries will be. However, if the uncorrelated subquery returns only a few rows, it will probably perform better than the query containing the correlated subquery.

The EXCEPT and INTERSECT Operators

INTERSECT operates on the results of two independent tables and must be performed on union compatible tables. In most cases, the two source tables are each generated by a SELECT. INTERSECT is the relational algebra

[3]Depending on your DBMS, you may get better performance using 1 instead of *. This holds true for DB2 and just might work with others.

intersect operation, which returns all rows the two tables have in common. It is the exact opposite of EXCEPT.

As a first example, let's prepare a query that lists all of the rare book store's customers *except* those who have made purchases with a total cost of more than $500. One way to write this query is

```
SELECT first_name, last_name
FROM customer
EXCEPT
SELECT first_name, last_name
FROM customer JOIN sale
WHERE sale_total_amt > 500;
```

Note that those customers who have made multiple purchases, some of which are less than $500 and some of which are greater than $500, will be excluded from the result.

If we replace the EXCEPT with an INTERSECT—

```
SELECT first_name, last_name
FROM customer
INTERSECT
SELECT first_name, last_name
FROM customer JOIN sale
WHERE sale_total_amt > 500;
```

—the query returns the names of those who *have* made a purchase of over $500. As you can see in Figure 18.5, the query results are quite different.

Output from the query using EXCEPT

```
 first_name | last_name
------------+-----------
 Edna       | Hayes
 Helen      | Jerry
 Jane       | Doe
 Jane       | Smith
 Janice     | Smith
 Jon        | Jones
 Mary       | Collins
 Peter      | Collins
```

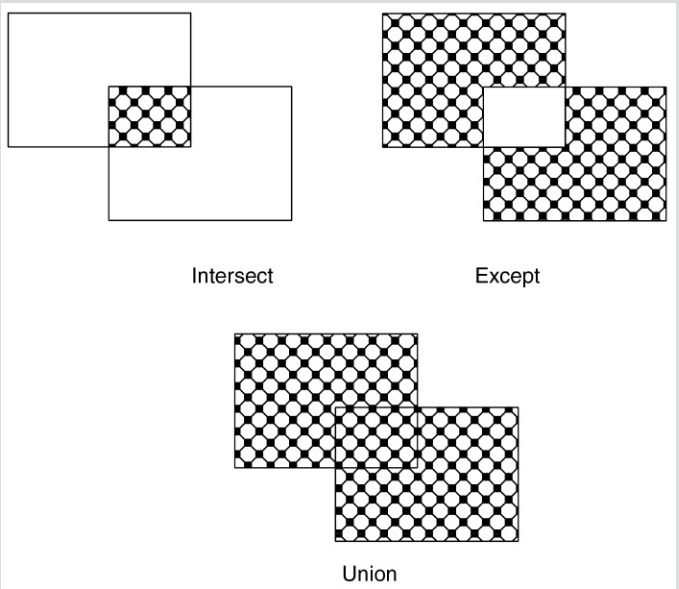
Output from the query using INTERSECT

```
 first_name | last_name
------------+-----------
 Franklin   | Hayes
 Janice     | Jones
```

■ **FIGURE 18.5 Output of queries using EXCEPT and INTERSECT.**

UNION Versus EXCEPT Versus INTERSET

One way to compare the operation of UNION, EXCEPT, and INTERSECT is to look at graphic representations, as in Figure 18.6. Each rectangle represents a table of data; the dark areas where the images overlap represent the rows returned by a query using the respective operation. As you can see, INTERSECT returns the area of overlap, EXCEPT returns everything EXCEPT the overlap, and UNION returns everything.



■ **FIGURE 18-6 Operation of the SQL INTERSECT, EXCEPT, and UNION operators.**

Performing Arithmetic

Although SQL is not a complete programming language, it can perform some calculations. SQL recognizes simple arithmetic expressions involving column names and literal values. (When you are working with embedded SQL, you can also use host language variables. See Appendix B for details.) For example, if you wanted to compute a discounted price for a volume, the computation could be written

```
asking_price *.9
```

You could then incorporate this into a query as

```
SELECT isbn, asking_price,
      asking_price * .9 AS discounted_price
FROM volume
where sale_id = 6;
```

The result of the preceding query can be found in Figure 18.7.

```
isbn               | asking_price | discounted_price
-------------------+--------------+-----------------
 978-1-11111-146-1 |        30.00 |          27.000
 978-1-11111-122-1 |        75.00 |          67.500
 978-1-11111-130-1 |       150.00 |         135.000
 978-1-11111-126-1 |       110.00 |          99.000
 978-1-11111-139-1 |       200.00 |         180.000
```

■ **FIGURE 18.7 Output of a query that includes a computed column.**

Arithmetic Operators

SQL recognizes the arithmetic operators in Table 18.1. Compared with a general-purpose programming language, this list is fairly limited. For example, there are no operators for exponentiation or modulo division. This means that if you need more sophisticated arithmetic manipulations, you will probably need to use embedded SQL to retrieve the data into host language variables and perform the arithmetic using the host programming language.

Table 18.1 SQL Arithmetic Operations

Operator	Meaning	Example
+	Unary +: preserve the sign of the value	`+balance`
−	Unary −: change the sign of the value	`-balance`
*	Multiplication: multiply two values	`balance * tax_rate`
/	Division: divide one value by another	`balance / numb_items`
+	Addition: add two values	`balance + new_charge`
−	Subtraction: subtract one value from another	`balance - payment`

Operator Precedence

The rows in Table 18.1 appear in the general order of the operators' precedence. (Both unary operators have the same precedence, followed by

multiplication and division. Addition and subtraction have the lowest precedence.) This means that when multiple operations appear in the same expression, the DBMS evaluates them according to their precedence. For example, because the unary operators have the highest precedence, for the expression

```
-balance * tax_rate
```

the DBMS will first change the sign of the value in the *balance* column and then multiply it by the value in the *tax_rate* column.

When more than one operator of the same precedence appears in the same expression, they are evaluated from left to right. Therefore, in the expression

```
balance + new_charges – payments
```

the DBMS will first add the new charges to the balance and then subtract the payments from the sum.

Sometimes, the default precedence can produce unexpected results. Assume that you want to evaluate the expression

```
12 / 3 * 2
```

When the operators are evaluated from left to right, the DBMS divides 12 by 3 and then multiplies the 4 by 2, producing an 8. However, what if you really wanted to perform the multiplication first, followed by the division? (The result would be 2.)

To change the order of evaluation, you use parentheses to surround the operations that should be performed first:

```
12 / (3 * 2)
```

Just as when you use parentheses to change the order of evaluation of logical operators, whenever the DBMS sees a set of parentheses it knows to evaluate what is inside the parentheses first, regardless of the precedence of the operators.

Keep in mind that you can nest one set of parentheses within another:

```
12 / (3 * (1 + 2))
```

In this example, the DBMS evaluates the innermost parentheses first (the addition), moves to the outer set of parentheses (the multiplication), and finally evaluates the division.

There is no limit to how deep you can nest parentheses. However, be sure that each opening parenthesis is paired with a closing parenthesis.

String Manipulation

The SQL core standard contains one operator and several functions for manipulating character strings.

Concatenation

As you saw when we were discussing joins using concatenated foreign keys, the concatenation operator—| |—pastes one string on the end of another. It can be used to format output as well as to concatenate keys for searching. For example, the rare book store could get an alphabetical list of customer names formatted as *last, first* (see Figure 18.8) with:

```
SELECT last_name || ', ' || first_name AS cat_name
FROM customer
ORDER BY last_name, first_name;
```

```
cat_name
-----------------
 Brown, Helen
 Collins, Mary
 Collins, Peter
 Doe, Jane
 Doe, John
 Hayes, Edna
 Hayes, Franklin
 Jerry, Helen
 Johnson, Peter
 Johnson, Peter
 Jones, Janice
 Jones, Jon
 Smith, Jane
 Smith, Janice
 Smith, John
```

■ **FIGURE 18.8 The result of a concatenation.**

Notice that the concatenation includes a literal string to place the comma and space between the last and first names. The concatenation operation knows nothing about normal English spacing; it simply places one string on the end of another. Therefore, it is up to the user to include any necessary spacing and punctuation.

UPPER and LOWER

When a DBMS evaluates a literal string against stored data, it performs a case-sensitive search. This means that upper- and lowercase letters are different:

'JONES' is not the same as 'Jones.' You can get around such problems using the UPPER and LOWER functions to convert stored data to a single case.

For example, assume that someone at the rare book store is not certain of the case in which customer names are stored. To perform a case-insensitive search for customers with a specific last name, the person could use

```
SELECT customer_numb, first_name, last_name
FROM customer
WHERE UPPER(last_name) = 'SMITH';
```

The result—

```
 customer_numb | first_name | last_name
---------------+------------+-----------
             5 | Jane       | Smith
             6 | Janice     | Smith
            15 | John       | Smith
```

—includes rows for customers whose last names are made up of the characters S-M-I-T-H, regardless of case. The UPPER function converts the data stored in the database to uppercase before making the comparison in the WHERE predicate. You obtain the same effect by using LOWER instead of UPPER and placing the characters to be matched in lower case.

Mixed Versus Single Case in Stored Data

There is always the temptation to require that text data be stored as all uppercase letters to avoid the need to use UPPER and LOWER in queries. For the most part, this isn't a good idea. First, text in all uppercase is difficult to read. Consider the following two lines of text:

```
WHICH IS EASIER TO READ? ALL CAPS OR MIXED CASE?

Which is easier to read? All caps or mixed case?
```

Our eyes have been trained to read mixed upper- and lowercase letters. In English, for example, we use letter case cues to locate the start of sentences and to identify proper nouns. Text in all caps removes those cues, making the text more difficult to read. The "sameness" of all uppercase also makes it more difficult to differentiate letters and, thus, to understand the words.

Second, because professional documents are typed/printed in mixed case, all uppercase looks less professional and isn't suitable for most business documents. Finally, Internet text communications in all uppercase are considered to be shouting and this feeling that all uppercase is impolite carries over into any displayed or printed output.

TRIM

The TRIM function removes leading and/or trailing characters from a string. The various syntaxes for this function and their effects are summarized in Table 18.2. The blanks in the first four examples can be replaced with any characters you need to remove, as can the * in the last example.

Table 18.2 The Various Forms of the SQL TRIM Function

Function Format	Result	Action of the Sample
`TRIM (' word ')`	`'word'`	Default: removes both leading and trailing blanks
`TRIM (BOTH ' ' FROM ' word ')`	`'word'`	Removes leading and trailing blanks
`TRIM (LEADING ' ' FROM ' word ')`	`'word'`	Removes leading blanks
`TRIM (TRAILING ' ' FROM ' word ')`	`'word'`	Removes trailing blanks
`TRIM (BOTH '*' FROM '*word*')`	`'word'`	Removes leading and trailing *

You can place TRIM in any expression that contains a string. For example, if you are using characters to store a serial number with leading 0s (for example, 0012), you can strip those 0s when performing a search:

```
SELECT item_description
FROM items
WHERE TRIM (LEADING '0' FROM item_numb) = '25';
```

SUBSTRING

The SUBSTRING function extracts portions of a string. It has the following general syntax:

```
SUBSTRING (source_string, FROM starting_position
     FOR number_of_characters)
```

For example, if the rare book store wanted to extract the first character of a customer's first name, the function call would be written

```
SUBSTRING (first_name FROM 1 FOR 1)
```

The substring being created begins at the first character of the column and is one character long.

You could then incorporate this into a query with

```
SELECT SUBSTRING (first_name FROM 1 FOR 1) || '. '
      || last_name AS whole_name
FROM customer;
```

The results can be found in Figure 18.9.

```
whole_name
------------
 J. Jones
 J. Jones
 J. Doe
 J. Doe
 J. Smith
 J. Smith
 H. Brown
 H. Jerry
 M. Collins
 P. Collins
 E. Hayes
 F. Hayes
 P. Johnson
 P. Johnson
 J. Smith
```

■ **FIGURE 18.9 Output of a query including the SUBSTRING function.**

Date and Time Manipulation

SQL DBMSs provide column data types for dates and times. When you store data using these data types, you make it possible for SQL to perform chronological operations on those values. You can, for example, subtract two dates to find out the number of days between them or add an interval to a date to advance the date a specified number of days. In this section, you will read about the types of date manipulations that SQL provides, along with a simple way to get current date and time information from the computer.

The core SQL standard specifies four column data types that relate to dates and times (jointly referred to as *datetime* data types):

- DATE: a date only,
- TIME: a time only.
- TIMESTAMP: a combination of date and time,
- INTERVAL: the interval between two of the preceding data types.

As you will see in the next two sections, these can be combined in a variety of ways.

Date and Time System Values

To help make date and time manipulations easier, SQL lets you retrieve the current date and/or time with the following three keywords:

- CURRENT_DATE: returns the current system date,
- CURRENT_TIME: returns the current system time

- CURRENT_TIMESTAMP: returns a combination of the current system date and time.

For example, to see all sales made on the current day, someone at the rare book store uses the following query:

```
SELECT first_name, last_name, sale_id
FROM customer JOIN sale
WHERE sale_date = CURRENT_DATE;
```

You can also use these system date and time values when performing data entry.

Date and Time Interval Operations

SQL dates and times can participate in expressions that support queries such as "how many days/months/years in between?" and operations such as "add 30 days to the invoice date." The types of date and time manipulations available with SQL are summarized in Table 18.3. Unfortunately, expressions involving these operations aren't as straightforward as they might initially appear. When you work with date and time intervals, you must also specify the portions of the date and/or time that you want.

Table 18.3 Datetime Arithmetic

Expression	Result
DATE ± integer	DATE
DATE ± time_interval	TIMESTAMP
DATE + time	TIMESTAMP
INVERVAL ± INTERVAL	INTERVAL
TIMESTAMP ± INTERVAL	TIMESTAMP
TIME ± time_interval	TIME
DATE - DATE	integer
TIME - TIME	INTERVAL
integer * INTERVAL	INTERVAL

Each datetime column will include a selection of the following fields:

- MILLENNIUM
- CENTURY
- DECADE
- YEAR
- QUARTER

- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- MILLISECONDS
- MICROSECONDS

When you write an expression that includes an interval, you can either indicate that you want the interval expressed in one of those fields (for example, DAY for the number of days between two dates) or specify a range of fields (for example, YEAR TO MONTH to give you an interval in years and months). The *start field* (the first field in the range) can be only YEAR, DAY, HOUR, or MINUTE. The second field in the range (the *end field*) must be a chronologically smaller unit than the start field.

Note: There is one exception to the preceding rule. If the start field is YEAR, then the end field must be MONTH.

To see the number of years between a customer's orders and the current date, someone at the rare book store might use

```
SELECT CURRENT_DATE - sale_date YEAR
FROM sale
WHERE customer_numb = 6;
```

To see the same interval expressed in years and months, the query would be rewritten as

```
SELECT CURRENT_DATE - sale_date YEAR TO MONTH
FROM sale
WHERE customer_numb = 6;
```

To add 7 days to an order date to give a customer an approximate delivery date, someone at the rare book store would write a query like

```
SELECT sale_date + INTERVAL '7' DAY
FROM sale
WHERE sale_id = 12;
```

Notice that when you include an interval as a literal you precede it with the keyword INTERVAL, put the interval's value in single quotes, and follow it with the datetime unit in which the interval is expressed.

OVERLAPS

The SQL OVERLAPS operator is a special-purpose keyword that returns true or false, depending on whether two datetime intervals overlap. This operator

might be used in applications such as hotel booking systems to determine room availability: Does one customer's planned stay overlap another's?

The operator has the following general syntax:

```
SELECT (start_date1, end_date1)
OVERLAPS
(start_date2, end_date2)
```

An expression such as

```
SELECT (DATE '16-Aug-2021', DATE '31-Aug-2021')
OVERLAPS
(DATE '18-Aug-2021', DATE '9-Sep-2021');
```

produces the following result:

```
overlaps
----------
t
```

Notice that the dates being compared are preceded by the keyword DATE and surrounded by single quotes. Without the specification of the type of data in the operation, SQL doesn't know how to interpret what is within the quotes.

The two dates and/or times that are used to specify an interval can be either DATE, TIME, or TIMESTAMP values or they can be intervals. For example, the following query checks to see whether the second range of dates is within 90 days of the first start date and returns false:

```
SELECT (DATE '16-Aug-2021', INTERVAL '90 DAYS')
OVERLAPS
(DATE '12-Feb-2021', DATE '4-Jun-2021');
```

Note: Because the OVERLAPS operator returns a Boolean, it can be used as the logical expression in a CASE statement, about which you will read shortly.

EXTRACT

The EXTRACT operator pulls out a part of a date and/or time. It has the following general format:

```
EXTRACT (datetime_field FROM datetime_value)
```

For example, the query

```
SELECT EXTRACT (YEAR FROM CURRENT_DATE);
```

returns the current year.

In addition to the datetime fields you saw earlier in this section, EXTRACT also can provide the day of the week (DOW) and the day of the year (DOY).

CASE Expressions

The SQL CASE expression, much like a CASE in a general purpose programming language, allows a SQL statement to pick from among a variety of actions based on the truth of logical expressions. Like arithmetic and string operations, the CASE statement generates a value to be displayed and therefore is part of the SELECT clause.

The CASE expression has the following general syntax:

```
CASE
      WHEN logical condition THEN action
      WHEN logical condition THEN action
      :
      :
      ELSE default action
END
```

It fits within a SELECT statement in this way:

```
SELECT column1, column2,
CASE
      WHEN logical condition THEN action
      WHEN logical condition THEN action
      :
      :
      ELSE default action
END
FROM table(s)
WHERE predicate;
```

The CASE does not necessarily need to be the last item in the SELECT clause. The END keyword can be followed by a comma and other columns or computed quantities.

As an example, assume that the rare book store wants to offer discounts to users based on the price of a book. The more the asking price for the book, the greater the discount. To include the discounted price in the output of a query, you could use

```
SELECT isbn, asking_price,
CASE
      WHEN asking_price < 50 THEN asking_price * .95
      WHEN asking_price < 75 THEN asking_price * .9
      WHEN asking_price < 100 THEN asking_price * .8
      ELSE asking_price * .75
END
FROM volume;
```

The preceding query displays the ISBN and the asking price of a book. It then evaluates the first CASE expression following WHEN. If that condition is true, the query performs the computation, displays the discounted price, and exits the CASE. If the first condition is false, the query proceeds to the second WHEN, and so on. If none of the conditions are true, the query executes the action following ELSE. (The ELSE is optional.)

The first portion of the output of the example query appears in Figure 18.10. Notice that the value returned by the CASE construct appears in a column named *case*. You can, however, rename the computed column just as you would rename any other computed column by adding AS followed by the desired name.

```
        isbn         | asking_price |   case
---------------------+--------------+----------
 978-1-11111-111-1 |       175.00 | 131.2500
 978-1-11111-131-1 |        50.00 |   45.000
 978-1-11111-137-1 |        80.00 |   64.000
 978-1-11111-133-1 |       300.00 | 225.0000
 978-1-11111-142-1 |        25.95 |  2465.25
 978-1-11111-146-1 |        22.95 |  2180.25
 978-1-11111-144-1 |        80.00 |   64.000
 978-1-11111-137-1 |        50.00 |   45.000
 978-1-11111-136-1 |        75.00 |   60.000
 978-1-11111-136-1 |        50.00 |   45.000
 978-1-11111-143-1 |        25.00 |  2375.00
 978-1-11111-132-1 |        15.00 |  1425.00
 978-1-11111-133-1 |        18.00 |  1710.00
 978-1-11111-121-1 |       110.00 |  82.5000
 978-1-11111-121-1 |       110.00 |  82.5000
 978-1-11111-121-1 |       110.00 |  82.5000
```

■ **FIGURE 18.10 Default output of a SELECT statement containing CASE.**

The output of the modified statement—

```
SELECT isbn, asking_price,
CASE
      WHEN asking_price < 50 THEN asking_price * .95
      WHEN asking_price < 75 THEN asking_price * .9
      WHEN asking_price < 100 THEN asking_price * .8
      ELSE asking_price * .75
END AS discounted_price
FROM volume;
```

—can be found in Figure 18.11

isbn	asking_price	discounted_price
978-1-11111-111-1	175.00	131.2500
978-1-11111-131-1	50.00	45.000
978-1-11111-137-1	80.00	64.000
978-1-11111-133-1	300.00	225.0000
978-1-11111-142-1	25.95	2465.25
978-1-11111-146-1	22.95	2180.25
978-1-11111-144-1	80.00	64.000
978-1-11111-137-1	50.00	45.000
978-1-11111-136-1	75.00	60.000
978-1-11111-136-1	50.00	45.000
978-1-11111-143-1	25.00	2375.00
978-1-11111-132-1	15.00	1425.00
978-1-11111-133-1	18.00	1710.00
978-1-11111-121-1	110.00	82.5000
978-1-11111-121-1	110.00	82.5000
978-1-11111-121-1	110.00	82.5000

■ **FIGURE 18.11 CASE statement output using a renamed column for the CASE value.**

Chapter 19 Working With Groups of Rows

The queries you have seen so far in this book for the most part operate on one row at a time. However, SQL also includes a variety of keywords and functions that work on groups of rows—either an entire table or a subset of a table. In this chapter, you will read about what you can do to and with grouped data.

Note: Many of the functions that you will be reading about in this chapter are often referred to as SQL's OLAP functions.

Set Functions

The basic SQL *set,* or *aggregate*, *functions* (summarized in Table 19.1) compute a variety of measures based on values in a column in multiple rows. The result of using one of these set functions is a computed column that appears only in a result table.

The basic syntax for a set function is

```
function_name (input_argument)
```

You place the function call following SELECT, just as you would an arithmetic calculation. What you use for an input argument depends on which function you are using.

Note: For the most part, you can count on a SQL DBMS supporting COUNT, SUM, AVG, MIN, and MAX. In addition, many DBMSs provide additional aggregate functions for measures such as standard deviation and variance. Consult the DBMS's documentation for details.

COUNT

The COUNT function is somewhat different from other SQL set functions in that instead of making computations based on data values, it counts the

Table 19.1 SQL Set Functions

Function	Meaning
Functions implemented by most DBMSs	
COUNT	Returns the number of rows
SUM	Returns the total of the values in a column from a group of rows
AVG	Returns the average of the values in a column from a group of rows
MIN	Returns the minimum value in a column from among a group of rows
MAX	Returns the maximum value in a column from among a group of rows
Less widely implemented functions	
COVAR_POP	Returns a population's covariance
COVAR_SAMP	Returns the covariance of a sample
REGR_AVGX	Returns the average of an independent variable
REGR_AVGY	Returns the average of a dependent variable
REGR_COUNT	Returns the number of independent/dependent variable pairs that remain in a population after any rows that have null in either variable have been removed
REGR_INTERCEPT	Returns the Y-intercept of a least-squares-fit linear equation
REGR_R2	Returns the square of a the correlation coefficient R
REGR_SLOPE	Returns the slope of a least-squares-fit linear equation
REGR_SXX	Returns the sum of the squares of the values of an independent variable
REGR_SXY	Returns the product of pairs independent and dependent variable values
REGR_SYY	Returns the sum of the square of the values of a dependent variable
STDDEV_POP	Returns the standard deviation of a population
STDDEV_SAMP	Returns the standard deviation of a sample
VAR_POP	Returns the variance of a population
VAR_SAMP	Returns the variance of a sample

number of rows in a table. To use it, you place COUNT (*) in your query. COUNT's input argument is always an asterisk:

```
SELECT COUNT (*)
FROM volume;
```

The response appears as

```
count

-------

    71
```

To count a subset of the rows in a table, you can apply a WHERE predicate:

```
SELECT COUNT (*)
FROM volume
WHERE isbn = '978-1-11111-141-1';
```

The result—

```
Count
-------
     7
```

—tells you that the store has sold or has in stock seven books with an ISBN of 978-1-11111-141-1. It does not tell you how many copies of the book are in stock or how many were purchased during any given sale because the query is simply counting the number of rows in which the ISBN appears. It does not take into account data in any other column.

Alternatively, the store could determine the number of distinct items contained in a specific order, with a query like

```
SELECT COUNT (*)
FROM volume
WHERE sale_id = 6;
```

When you use * as an input parameter to the COUNT function, the DBMS includes all rows. However, if you wish to exclude rows that have nulls in a particular column, you can use the name of the column as an input parameter. To find out how many volumes are currently in stock, the rare book store could use

```
SELECT COUNT (selling_price)
FROM volume;
```

If every row in the table has a value in the *selling_date* column, then COUNT (*selling_date*) is the same as COUNT (*). However, if any rows contain null, then the count will exclude those rows. There are 71 rows in the *volume* table. However, the count returns a value of 43, indicating that 43 volumes have not been sold and therefore are in stock.

You can also use COUNT to determine how many unique values appear in any given column by placing the keyword DISTINCT in front of the column name used as an input parameter. For example, to find out how many different books appear in the *volume* table, the rare book store would use

```
SELECT COUNT (DISTINCT isbn)
FROM volume;
```

The result—27—is the number of unique ISBNs in the table.

SUM

If someone at the rare book store wanted to know the total amount of an order so that value could be inserted into the *sale* table, then the easiest way to obtain this value is to add up the values in the *selling_price* column:

```
SELECT SUM (selling_price)
FROM volume
WHERE sale_id = 6;
```

The result appears as

```
  sum
--------
 505.00
```

In the preceding example, the input argument to the SUM function was a single column. However, it can also be an arithmetic operation. For example, to find the total of a sale if the books are discounted 15 percent, the rare book store could use the following query:

```
SELECT SUM (selling_price * .85)
FROM volume
WHERE sale_id = 6;
```

The result—

```
   sum
----------
 429.2500
```

—is the total of the multiplication of the selling price times the selling discount.

If we needed to add tax to a sale, a query could then multiply the result of the SUM by the tax rate,

```
SELECT SUM (selling_price * .85) * 1.0725
FROM volume
WHERE sale_id = 6;
```

producing a final result of 429.2500.

Note: Rows that contain nulls in any column involved in a SUM are excluded from the computation.

AVG

The AVG function computes the average value in a column. For example, to find the average price of a book, someone at the rare book store could use a query like

```
SELECT AVG (selling_price)
FROM volume;
```

The result is 68.2313953488372093 (approximately $68.23).

Note: Rows that contain nulls in any column involved in an AVG are excluded from the computation.

MIN and MAX

The MIN and MAX functions return the minimum and maximum values in a column or expression. For example, to see the maximum price of a book, someone at the rare book store could use a query like

```
SELECT MAX (selling_price)
FROM volume;
```

The result is a single value: $205.00.

The MIN and MAX functions are not restricted to columns or expressions that return numeric values. If someone at the rare book store wanted to see the latest date on which a sale had occurred, then

```
SELECT MAX (sale_date)
FROM volume;
```

returns the chronologically latest date (in our particular sample data, 01-Sep-21).

By the same token, if you use

```
SELECT MIN (last_name)
FROM customer;
```

you will receive the alphabetically first customer last name (Brown).

Set Functions in Predicates

Set functions can also be used in WHERE predicates to generate values against which stored data can be compared. Assume, for example, that someone at the rare book store wants to see the titles and cost of all books that were sold that cost more than the average cost of a book.

The strategy for preparing this query is to use a subquery that returns the average cost of a sold book and to compare the cost of each book in the *volume* table to that average:

```
SELECT title, selling_price
FROM work, book, volume
WHERE work.work_numb = book.work_numb
      AND book.isbn = volume.isbn
      AND selling_price > (SELECT AVG (selling_price)
      FROM volume);
```

Although it would seem logical that the DBMS would calculate the average once and use the result of that single computation to compare to rows in the *volume*, that's not what happens. This is actually a correlated subquery; the DBMS recalculates the average for every row in *volume*. As a result, a query of this type will perform relatively slowly on large amounts of data. You can find the result in Figure 19.1.

Changing Data Types: CAST

One of the problems with the output of the SUM and AVG functions that you saw in the preceding section of this chapter is that they give you no control over the *precision* (number of places to the right of the decimal point) of the output. One way to solve that problem is to change the data type of the result to something that has the number of decimal places you want using the CAST function.

title	selling_price
Jane Eyre	175.00
Giles Goat Boy	285.00
Anthem	76.10
Tom Sawyer	110.00
Tom Sawyer	110.00
Adventures of Huckleberry Finn, The	75.00
Treasure Island	120.00
Fountainhead, The	110.00
I, Robot	170.00
Fountainhead, The	75.00
Giles Goat Boy	125.00
Fountainhead, The	75.00
Foundation	75.00
Treasure Island	150.00
Lost in the Funhouse	75.00
Hound of the Baskervilles	75.00

■ **FIGURE 19.1 Output of a query that uses a set function in a subquery.**

CAST has the general syntax

```
CAST (source_data AS new_data_type)
```

To restrict the output of the average price of books to a precision of 2, you could then use

```
CAST (AVG (selling_price) AS DECIMAL (10,2))
```

and incorporate it into a query using

```
SELECT CAST (AVG (selling_price) AS DECIMAL (10,2))
FROM volume;
```

The preceding specifies that the result should be displayed as a decimal number with a maximum of 10 digits (including the decimal point) with two digits to the right of the decimal point. The result is 68.23, a more meaningful currency value than the original 68.2313953488372093.

Note: If you request more digits of precision than are available, the DBMS may add trailing 0s or it may simply show you all digits available without padding the result to the specified length.

CAST also can be used, for example, to convert a string of characters into a date. The expression

```
CAST ('10-Aug-2021' AS DATE)
```

returns a datetime value.

Valid conversions for commonly used data types are represented by the light gray boxes in Table 19.2. Those conversions that may be possible if certain conditions are met are represented by the dark gray boxes. In particular, if you are attempting to convert a character string into a shorter string, the result will be truncated.

Table 19.2 Valid Data Type Conversion for Commonly Used Data Types (Light Gray Boxes are Valid; Dark Gray Boxes May Be Valid)

	New data type						
Original data type	**Integer or fixed point**	**Floating point**	**Variable length character**	**Fixed length character**	**Date**	**Time**	**Timestamp**
Integer or fixed point							
Floating point							
Character (fixed or variable length)							
Date							
Time							
Timestamp							

Grouping Queries

SQL can group rows based on matching values in specified columns and compute summary measures for each group. When these *grouping queries* are combined with the set functions that you saw earlier in this chapter, SQL can provide simple reports without requiring any special programming.

Forming Groups

To form a group, you add a GROUP BY clause to a SELECT statement, followed by the columns whose values are to be used to form the groups. All rows whose values match on those columns will be placed in the same group.

For example, if someone at the rare book store wants to see how many copies of each book edition have been sold, he or she can use a query like

```
SELECT isbn, COUNT(*)
FROM volume
GROUP BY isbn
ORDER BY isbn;
```

The query forms groups by matching ISBNs. It displays the ISBN and the number of rows in each group (see Figure 19.2).

isbn	count
978-1-11111-111-1	1
978-1-11111-115-1	1
978-1-11111-121-1	3
978-1-11111-122-1	1
978-1-11111-123-1	2
978-1-11111-124-1	1
978-1-11111-125-1	1
978-1-11111-126-1	3
978-1-11111-127-1	5
978-1-11111-128-1	1
978-1-11111-129-1	1
978-1-11111-130-1	4
978-1-11111-131-1	4
978-1-11111-132-1	3
978-1-11111-133-1	5
978-1-11111-135-1	1
978-1-11111-136-1	6
978-1-11111-137-1	4
978-1-11111-138-1	4
978-1-11111-139-1	4
978-1-11111-140-1	1
978-1-11111-141-1	7
978-1-11111-142-1	1
978-1-11111-143-1	1
978-1-11111-144-1	1
978-1-11111-145-1	3
978-1-11111-146-1	2

■ **FIGURE 19.2 Counting the members of a group.**

There is a major restriction that you must observe with a grouping query: You can display values only from columns that are used to form the groups. As an example, assume that someone at the rare book store wants to see the

number of copies of each title that have been sold. A working query could be written

```
SELECT title, COUNT (*)
FROM volume, book, work
WHERE volume.isbn = book.isbn
      AND book.work_numb = work.work_numb
GROUP BY title
ORDER BY title;
```

The result appears in Figure 19.3. The problem with this approach is that the same title may have different ISBNs (for different editions), producing multiple entries for the same title. To ensure that you have a count for a title, including all editions, you will need to group by the work number. However, given the restriction as to what can be displayed, you won't be able to display the title.

title	count
Adventures of Huckleberry Finn, The	1
Anathem	1
Anthem	4
Atlas Shrugged	5
Bourne Supremacy, The	1
Cryptonomicon	2
Foundation	11
Fountainhead, The	4
Giles Goat Boy	5
Hound of the Baskervilles	1
I, Robot	4
Inkdeath	7
Inkheart	1
Jane Eyre	1
Kidnapped	2
Last Foundation	4
Lost in the Funhouse	3
Matarese Circle, The	2
Snow Crash	1
Sot Weed Factor, The	4
Tom Sawyer	3
Treasure Island	4

■ **FIGURE 19.3 Grouping rows by book title.**

The solution is to make the DBMS do a bit of extra work: group by both the work number and the title. (Keep in mind that the title is functionally dependent on the work number, so there will always be only one title for each work number.) The DBMS will then form groups that have the same

values in both columns. The result will be the same as that in Figure 19.3, using our sample data. We therefore gain the ability to display the title when grouping by the work number. The query could be written

```
SELECT work.work_numb, title, COUNT (*)
FROM volume, book, work
WHERE volume.isbn = book.isbn
      AND book.work_numb = work.work_numb
GROUP BY work_numb, title
ORDER BY title;
```

As you can see in Figure 19.4, the major difference between the two results is the appearance of the work number column.

work_numb	title	count
9	Adventures of Huckleberry Finn, The	1
28	Anathem	1
30	Anthem	4
14	Atlas Shrugged	5
12	Bourne Supremacy, The	1
31	Cryptonomicon	2
23	Foundation	11
13	Fountainhead, The	4
20	Giles Goat Boy	5
3	Hound of the Baskervilles	1
25	I, Robot	4
27	Inkdeath	7
26	Inkheart	1
1	Jane Eyre	1
16	Kidnapped	2
24	Last Foundation	4
19	Lost in the Funhouse	3
11	Matarese Circle, The	2
29	Snow Crash	1
18	Sot Weed Factor, The	4
8	Tom Sawyer	3
17	Treasure Island	4

■ **FIGURE 19.4 Grouped output using two grouping columns.**

You can use any of the set functions in a grouping query. For example, someone at the rare book store could generate the total cost of all sales with

```
SELECT sale_id, SUM (selling_price)
FROM volume
GROUP BY sale_id;
```

The result can be seen in Figure 19.5. Notice that the last line of the result has nulls for both output values. This occurs because those volumes that haven't been sold have null for the sale ID and selling price. If you wanted to clean up the output, removing rows with nulls, you could add a WHERE clause:

```
SELECT sale_id, SUM (selling_price)
FROM volume
WHERE NOT (sale_id IS NULL)
GROUP BY sale_id;
```

```
 sale_id |  sum
---------+--------
       1 | 510.00
       2 | 125.00
       3 |  58.00
       4 | 110.00
       5 | 110.00
       6 | 505.00
       7 |  80.00
       8 | 130.00
       9 |  50.00
      10 | 125.00
      11 | 200.00
      12 | 225.00
      13 |  25.95
      14 |  80.00
      15 | 100.00
      16 | 130.00
      17 | 100.00
      18 | 100.00
      19 |  95.00
      20 |  75.00
         |
```

■ **FIGURE 19.5 The result of using a set function in a grouping query.**

In an earlier example we included the book title as part of the GROUP BY clause as a trick to allow us to display the title in the result. However, more commonly we use multiple grouping columns to create nested groups. For example, if someone at the rare book store wanted to see the total cost of purchases made by each customer per day, the query could be written

```
SELECT customer.customer_numb, sale_date,
      sum (selling_price)
FROM customer, sale, volume
WHERE customer.customer_numb = sale.customer_numb
      AND sale.sale_id = volume.sale_id
GROUP BY customer.customer_numb, sale_date;
```

Because the *customer_numb* column is listed first in the GROUP BY clause, its values are used to create the outer groupings. The DBMS then

groups order by date *within* customer numbers. The default output (see Figure 19.6) is somewhat hard to interpret because the outer groupings are not in order. However, if you add an ORDER BY clause to sort the output by customer number, you can see the ordering by date within each customer (see Figure 19.7).

customer_numb	sale_date	sum
1	15-JUN-21 00:00:00	58.00
6	01-SEP-21 00:00:00	95.00
2	01-SEP-21 00:00:00	75.00
5	22-AUG-21 00:00:00	100.00
2	25-JUL-21 00:00:00	210.00
1	25-JUL-21 00:00:00	100.00
8	07-JUL-21 00:00:00	50.00
5	07-JUL-21 00:00:00	130.00
12	05-JUL-21 00:00:00	505.00
8	05-JUL-21 00:00:00	80.00
6	10-JUL-21 00:00:00	80.00
2	10-JUL-21 00:00:00	25.95
6	30-JUN-21 00:00:00	110.00
9	10-JUL-21 00:00:00	200.00
10	10-JUL-21 00:00:00	225.00
4	30-JUN-21 00:00:00	110.00
11	10-JUL-21 00:00:00	125.00
11	12-JUL-21 00:00:00	100.00
1	05-JUN-21 00:00:00	125.00
1	29-MAY-21 00:00:00	510.00

■ **FIGURE 19.6 Group by two columns (default row order).**

customer_numb	sale_date	sum
1	29-MAY-21 00:00:00	510.00
1	05-JUN-21 00:00:00	125.00
1	15-JUN-21 00:00:00	58.00
1	25-JUL-21 00:00:00	100.00
2	10-JUL-21 00:00:00	25.95
2	25-JUL-21 00:00:00	130.00
2	01-SEP-21 00:00:00	75.00
4	30-JUN-21 00:00:00	110.00
5	07-JUL-21 00:00:00	130.00
5	22-AUG-21 00:00:00	100.00
6	30-JUN-21 00:00:00	110.00
6	10-JUL-21 00:00:00	80.00
6	01-SEP-21 00:00:00	95.00
8	05-JUL-21 00:00:00	80.00
8	07-JUL-21 00:00:00	50.00
9	10-JUL-21 00:00:00	200.00
10	10-JUL-21 00:00:00	225.00
11	10-JUL-21 00:00:00	125.00
11	12-JUL-21 00:00:00	100.00
12	05-JUL-21 00:00:00	505.00

■ **FIGURE 19.7 Grouping by two columns (rows sorted by outer grouping column).**

Restricting Groups

The grouping queries you have seen to this point include all the rows in the table. However, you can restrict the rows that are included in grouped output using one of two strategies:

- Restrict the rows before groups are formed.
- Allow all groups to be formed and then restrict the groups.

The first strategy is performed with the WHERE clause in the same way we have been restricting rows to this point. The second requires a HAVING clause, which contains a predicate that applies to groups after they are formed.

Assume, for example, that someone at the rare book store wants to see the number of books ordered at each price over $75. One way to write the query is to use a WHERE clause to throw out rows with a selling price less than or equal to $75:

```
SELECT selling_price, count (*)
FROM volume
WHERE selling_price > 75
GROUP BY selling_price;
```

Alternatively, you could let the DBMS form the groups and then throw out the groups that have a cost less than or equal to $75 with a HAVING clause:

```
SELECT selling_price, count (*)
FROM volume
GROUP BY selling_price
HAVING selling_price > 75;
```

The result in both cases is the same (see Figure 19.8). However, the way in which the query is processed is different.

selling_price	count
76.10	1
110.00	3
120.00	1
125.00	1
150.00	1
170.00	1
175.00	1
285.00	1

■ **FIGURE 19.8 Restrict groups to volumes that cost more than $75.**

Windowing and Window Functions

Grouping queries have two major drawbacks: they can't show you individual rows at the same time they show you computations made on groups of rows and you can't see data from non-grouping columns unless you resort to the group making trick shown earlier. The more recent versions of the SQL standard (from SQL:2003 onward), however, include a new way to compute aggregate functions, yet display the individual rows within each group: *windowing*. Each window (or *partition*) is a group of rows that share some criteria, such as a customer number. The window has a *frame* that "slides" to present to the DBMS the rows that share the same value of the partitioning criteria. *Window functions* are a special group of functions that can act only on partitions.

Note: By default, a window frame includes all the rows as its partition. However, as you will see shortly, that can be changed.

Let's start with a simple example. Assume that someone at the rare book store wants to see the volumes that were part of each sale, as well as the average cost of books for each sale. A grouping query version wouldn't be able to show the individual volumes in a sale nor would it be able to display the ISBN or *sale_id* unless those two values were added to the GROUP BY clause. However, if the query were written using windowing—

```
SELECT sale_id, isbn, CAST (AVG(selling_price)
      OVER (PARTITION BY sale_id) AS DECIMAL (7,2))
FROM volume
WHERE sale_id IS NOT NULL;
```

—it would produce the result in Figure 19.9. Notice that the individual volumes from each sale are present, and that the rightmost column contains the average cost for the specific sale on which a volume was sold. This means that the *avg* column in the result table is the same for all rows that come from a given sale.

The query itself includes two new keywords: OVER and PARTITION BY. (The CAST is present to limit the display of the average to a normal money display format and therefore isn't part of the windowing expression.) OVER indicates that the rows need to be grouped in some way. PARTITION BY indicates the criteria by which the rows are to be grouped. This particular example computes the average for groups of rows that are separated by their sale ID.

To help us explore more of what windowing can do, we're going to need a sample table with some different types of data. Figure 19.10a shows you a table that describes sales representatives and the value of product they have sold in specific quarters. The names of the sales reps are stored in the table labeled as Figure 19.10b.

sale_id	isbn	avg
1	978-1-11111-111-1	170.00
1	978-1-11111-131-1	170.00
1	978-1-11111-133-1	170.00
2	978-1-11111-142-1	41.67
2	978-1-11111-146-1	41.67
2	978-1-11111-144-1	41.67
3	978-1-11111-143-1	42.00
3	978-1-11111-132-1	42.00
3	978-1-11111-133-1	42.00
3	978-1-11111-121-1	42.00
5	978-1-11111-121-1	110.00
6	978-1-11111-146-1	101.00
6	978-1-11111-122-1	101.00
6	978-1-11111-130-1	101.00
6	978-1-11111-126-1	101.00
6	978-1-11111-139-1	101.00
7	978-1-11111-125-1	40.00
7	978-1-11111-131-1	40.00
8	978-1-11111-126-1	65.00
8	978-1-11111-133-1	65.00
9	978-1-11111-139-1	50.00
10	978-1-11111-133-1	125.00
11	978-1-11111-126-1	66.67
11	978-1-11111-130-1	66.67
11	978-1-11111-136-1	66.67
12	978-1-11111-130-1	112.50
12	978-1-11111-132-1	112.50
13	978-1-11111-129-1	25.95
14	978-1-11111-141-1	40.00
14	978-1-11111-141-1	40.00
15	978-1-11111-127-1	50.00
15	978-1-11111-141-1	50.00
16	978-1-11111-141-1	43.33
16	978-1-11111-123-1	43.33
16	978-1-11111-127-1	43.33
17	978-1-11111-133-1	50.00
17	978-1-11111-127-1	50.00
18	978-1-11111-135-1	33.33
18	978-1-11111-131-1	33.33
18	978-1-11111-127-1	33.33
19	978-1-11111-128-1	47.50
19	978-1-11111-136-1	47.50
20	978-1-11111-115-1	75.00

■ **FIGURE 19.9 Output of a simple query using windowing.**

(a)

quarterly_sales

id	quarter	year	sales_amt
1	1	2020	518.00
1	2	2020	1009.00
1	3	2020	1206.00
1	4	2020	822.00
1	1	2021	915.00
1	2	2021	1100.00
2	1	2020	789.00
2	2	2020	1035.00
2	3	2020	1235.00
2	4	2020	1355.00
2	1	2021	1380.00
2	2	2021	1400.00
3	3	2020	795.00
3	4	2020	942.00
3	1	2021	1012.00
3	2	2021	1560.00
4	1	2020	1444.00
4	2	2020	1244.00
4	3	2020	987.00
4	4	2020	502.00
5	1	2020	1200.00
5	2	2020	1200.00
5	3	2020	1200.00
5	4	2020	1200.00
5	1	2021	1200.00
5	2	2021	1200.00
6	1	2020	925.00
6	2	2020	1125.00
6	3	2020	1250.00
6	4	2020	1387.00
6	1	2021	1550.00
6	2	2021	1790.00
7	1	2021	2201.00
7	2	2021	2580.00
8	1	2021	1994.00
8	2	2021	2121.00
9	1	2021	502.00
9	2	2021	387.00
10	1	2021	918.00
10	2	2021	1046.00

(b)

rep_names

id	first_name	last_name
1	John	Anderson
2	Jane	Anderson
3	Mike	Baker
4	Mary	Carson
5	Bill	Davis
6	Betty	Esteban
7	Jack	Fisher
8	Jen	Grant
9	Larry	Holmes
10	Lily	Imprego

■ **FIGURE 19.10 Quarterly sales tables for use in windowing examples.**

Note: Every windowing query must have an OVER clause, but you can leave out the PARTITION BY clause—using only OVER—if you want all the rows in the table to be in the same partition.

Ordering the Partitioning

When SQL processes a windowing query, it scans the rows in the order they appear in the table. However, you control the order in which rows are processed by adding an ORDER BY clause to the PARTITION BY expression. As you will see, doing so can alter the result, producing a "running" average or sum.

Consider first a query similar to the first windowing example:

```
SELECT first_name, last_name, quarter, year, sales_amt,
     CAST (AVG (sales_amt
     OVER (PARTITION BY quarterly_sales.sales_id) AS DECIMAL (7,2))
FROM rep_names, quarterly_sales
WHERE rep_names.id = quarterly_sales.id;
```

As you can see in Figure 19.11, the output is what you would expect: Each line displays the average sales for the given sales representative. The DBMS adds up the sales for all quarters for the sales person and divides by the number of quarters. However, if we add an ORDER BY clause to force processing in quarter and year order, the results are quite different.

The query changes only a bit:

```
SELECT first_name, last_name, quarter, year, sales_amt
   CAST (AVG (sales_amt OVER (PARTITION BY quarterly_sales.sales_id
   ORDER BY year, quarter) AS DECIMAL (7,2))
FROM rep_names, quarterly_sales
WHERE rep_names_id = quarterly_sales.id
ORDER BY year, quarter;
```

However, in this case, the ORDER BY clause forces the DBMS to process the rows in year and quarter order. As you can see in Figure 19.12, the average column is now a moving average. What is actually happening is that the window frame is changing in the partition each time a row is scanned. The first row in a partition is averaged by itself. Then, the window frame expands to include two rows and both are included in the average. This process repeats until all the rows in the partition have been included in the average. Therefore, each line in the output of this version of the query gives you the average at the end of that quarter, rather than for all quarters.

Note: If you replace the AVG in the preceding query with the SUM function, you'll get a running total of the sales made by each sales representative.

first_name	last_name	quarter	year	sales_amt	avg
John	Anderson	1	2020	518.00	928.33
John	Anderson	1	2021	915.00	928.33
John	Anderson	2	2020	1009.00	928.33
John	Anderson	2	2021	1100.00	928.33
John	Anderson	3	2020	1206.00	928.33
John	Anderson	4	2020	822.00	928.33
Jane	Anderson	1	2020	789.00	1199.00
Jane	Anderson	1	2021	1380.00	1199.00
Jane	Anderson	2	2020	1035.00	1199.00
Jane	Anderson	2	2021	1400.00	1199.00
Jane	Anderson	3	2020	1235.00	1199.00
Jane	Anderson	4	2020	1355.00	1199.00
Mike	Baker	1	2021	1012.00	1077.25
Mike	Baker	2	2021	1560.00	1077.25
Mike	Baker	3	2020	795.00	1077.25
Mike	Baker	4	2020	942.00	1077.25
Mary	Carson	1	2020	1444.00	1044.25
Mary	Carson	2	2020	1244.00	1044.25
Mary	Carson	3	2020	987.00	1044.25
Mary	Carson	4	2020	502.00	1044.25
Bill	Davis	1	2020	1200.00	1200.00
Bill	Davis	1	2021	1200.00	1200.00
Bill	Davis	2	2020	1200.00	1200.00
Bill	Davis	2	2021	1200.00	1200.00
Bill	Davis	3	2020	1200.00	1200.00
Bill	Davis	4	2020	1200.00	1200.00
Betty	Esteban	1	2020	925.00	1337.83
Betty	Esteban	1	2021	1550.00	1337.83
Betty	Esteban	2	2020	1125.00	1337.83
Betty	Esteban	2	2021	1790.00	1337.83
Betty	Esteban	3	2020	1250.00	1337.83
Betty	Esteban	4	2020	1387.00	1337.83
Jack	Fisher	1	2021	2201.00	2390.50
Jack	Fisher	2	2021	2580.00	2390.50
Jen	Grant	1	2021	1994.00	2057.50
Jen	Grant	2	2021	2121.00	2057.50
Larry	Holmes	1	2021	502.00	444.50
Larry	Holmes	2	2021	387.00	444.50
Lily	Imprego	1	2021	918.00	982.00
Lily	Imprego	2	2021	1046.00	982.00

■ **FIGURE 19.11 Computing the windowed average without ordering the rows.**

first_name	last_name	quarter	year	sales_amt	avg
John	Anderson	1	2020	518.00	518.00
John	Anderson	2	2020	1009.00	763.50
John	Anderson	3	2020	1206.00	911.00
John	Anderson	4	2020	822.00	888.75
John	Anderson	1	2021	915.00	894.00
John	Anderson	2	2021	1100.00	928.33
Jane	Anderson	1	2020	789.00	789.00
Jane	Anderson	2	2020	1035.00	912.00
Jane	Anderson	3	2020	1235.00	1019.67
Jane	Anderson	4	2020	1355.00	1103.50
Jane	Anderson	1	2021	1380.00	1158.80
Jane	Anderson	2	2021	1400.00	1199.00
Mike	Baker	3	2020	795.00	795.00
Mike	Baker	4	2020	942.00	868.50
Mike	Baker	1	2021	1012.00	916.33
Mike	Baker	2	2021	1560.00	1077.25
Mary	Carson	1	2020	1444.00	1444.00
Mary	Carson	2	2020	1244.00	1344.00
Mary	Carson	3	2020	987.00	1225.00
Mary	Carson	4	2020	502.00	1044.25
Bill	Davis	1	2020	1200.00	1200.00
Bill	Davis	2	2020	1200.00	1200.00
Bill	Davis	3	2020	1200.00	1200.00
Bill	Davis	4	2020	1200.00	1200.00
Bill	Davis	1	2021	1200.00	1200.00
Bill	Davis	2	2021	1200.00	1200.00
Betty	Esteban	1	2020	925.00	925.00
Betty	Esteban	2	2020	1125.00	1025.00
Betty	Esteban	3	2020	1250.00	1100.00
Betty	Esteban	4	2020	1387.00	1171.75
Betty	Esteban	1	2021	1550.00	1247.40
Betty	Esteban	2	2021	1790.00	1337.83
Jack	Fisher	1	2021	2201.00	2201.00
Jack	Fisher	2	2021	2580.00	2390.50
Jen	Grant	1	2021	1994.00	1994.00
Jen	Grant	2	2021	2121.00	2057.50
Larry	Holmes	1	2021	502.00	502.00
Larry	Holmes	2	2021	387.00	444.50
Lily	Imprego	1	2021	918.00	918.00
Lily	Imprego	2	2021	1046.00	982.00

■ **FIGURE 19.12 Computing the windowed average with row ordering.**

If you don't want a running sum or average, you can use a *frame clause* to change the size of the window (which rows are included). To suppress the cumulative average in Figure 19.12, you would add ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW following the columns by which the rows within each partition are to be ordered. The window frame clauses are summarized in Table 19.3.

Table 19.3 Window Frame Clauses

Frame clause	Action

`RANGE UNBOUNDED PRECEDING (default)`
`RANGE BETWEEN UNBOUNDED PRECEDING`
`  AND CURRENT ROW`
	Include all rows within the current partition through the current row, based on the ordering specified in the ORDER BY clause. If no ORDER BY clause, include all rows. If there are duplicate rows, include their values only once

`RANGE BETWEEN UNBOUNDED PRECEDING`
`  AND UNBOUNDED FOLLOWING`
`ROWS BETWEEN UNBOUNDED PRECEDING`
`  AND UNBOUNDED FOLLOWING`
	Include all rows in the partition

`ROWS UNBOUNDED PRECEDING`
`ROWS BETWEEN UNBOUNDED PRECEDING`
`  AND CURRENT ROW`
	Include all rows within the current partition through the current row, including duplicate rows

Specific Functions

The window functions built into SQL perform actions that are only meaningful on partitions. Many of them include ways to rank data, something that is difficult to do otherwise. They can also number rows and compute distribution percentages. In this section we'll look at some of the specific functions: what they can do for you, and how they work.

Note: Depending on your DBMS, you may find additional window functions available, some of which are not part of the SQL standard.

RANK

The RANK function orders and numbers rows in a partition based on the value in a particular column. It has the general format

```
RANK () OVER (partition_specifications)
```

For example, if we wanted to see all the quarterly sales data ranked for all the sales representatives, the query could look like the following:

```
SELECT first_name, last_name, quarter, year, sales_amt,
     RANK () OVER (ORDER BY sales_amt desc)
FROM rep_names, quarterly_sales
WHERE rep_names.id = quarterly_sales.id;
```

The output appears in Figure 19.13. Notice that, because there is no PARTITION BY clause in the query, all of the rows in the table are part of a single ranking.

first_name	last_name	quarter	year	sales_amt	rank
Jack	Fisher	2	2021	2580.00	1
Jack	Fisher	1	2021	2201.00	2
Jen	Grant	2	2021	2121.00	3
Jen	Grant	1	2021	1994.00	4
Betty	Esteban	2	2021	1790.00	5
Mike	Baker	2	2021	1560.00	6
Betty	Esteban	1	2021	1550.00	7
Mary	Carson	1	2020	1444.00	8
Jane	Anderson	2	2021	1400.00	9
Betty	Esteban	4	2020	1387.00	10
Jane	Anderson	1	2021	1380.00	11
Jane	Anderson	4	2020	1355.00	12
Betty	Esteban	3	2020	1250.00	13
Mary	Carson	2	2020	1244.00	14
Jane	Anderson	3	2020	1235.00	15
John	Anderson	3	2020	1206.00	16
Bill	Davis	4	2020	1200.00	17
Bill	Davis	3	2020	1200.00	17
Bill	Davis	1	2021	1200.00	17
Bill	Davis	2	2021	1200.00	17
Bill	Davis	1	2020	1200.00	17
Bill	Davis	2	2020	1200.00	17
Betty	Esteban	2	2020	1125.00	23
John	Anderson	2	2021	1100.00	24
Lily	Imprego	2	2021	1046.00	25
Jane	Anderson	2	2020	1035.00	26
Mike	Baker	1	2021	1012.00	27
John	Anderson	2	2020	1009.00	28
Mary	Carson	3	2020	987.00	29
Mike	Baker	4	2020	942.00	30
Betty	Esteban	1	2020	925.00	31
Lily	Imprego	1	2021	918.00	32
John	Anderson	1	2021	915.00	33
John	Anderson	4	2020	822.00	34
Mike	Baker	3	2020	795.00	35
Jane	Anderson	1	2020	789.00	36
John	Anderson	1	2020	518.00	37
Larry	Holmes	1	2021	502.00	38
Mary	Carson	4	2020	502.00	38
Larry	Holmes	2	2021	387.00	40

■ **FIGURE 19.13 Ranking all quarterly sales.**

Alternatively, you could rank each sales representative's sales to identify the quarters in which each representative sold the most. The query would be written

```
SELECT first_name, last_name, quarter, year, sales_amt,
     RANK () OVER (PARTITION BY quarterly_sales.id
          ORDER BY sales_amt DESC)
FROM rep_names, quarterly_sales
WHERE rep_names.id = quarterly_sales.id;
```

The output can be found in Figure 19.14.

Note: When there are duplicate rows, the RANK function includes only one of the duplicates. However, if you want to include the duplicates, use DENSE_RANK instead of RANK.

first_name	last_name	quarter	year	sales_amt	rank
John	Anderson	3	2020	1206.00	1
John	Anderson	2	2021	1100.00	2
John	Anderson	2	2020	1009.00	3
John	Anderson	1	2021	915.00	4
John	Anderson	4	2020	822.00	5
John	Anderson	1	2020	518.00	6
Jane	Anderson	2	2021	1400.00	1
Jane	Anderson	1	2021	1380.00	2
Jane	Anderson	4	2020	1355.00	3
Jane	Anderson	3	2020	1235.00	4
Jane	Anderson	2	2020	1035.00	5
Jane	Anderson	1	2020	789.00	6
Mike	Baker	2	2021	1560.00	1
Mike	Baker	1	2021	1012.00	2
Mike	Baker	4	2020	942.00	3
Mike	Baker	3	2020	795.00	4
Mary	Carson	1	2020	1444.00	1
Mary	Carson	2	2020	1244.00	2
Mary	Carson	3	2020	987.00	3
Mary	Carson	4	2020	502.00	4
Bill	Davis	1	2020	1200.00	1
Bill	Davis	2	2020	1200.00	1
Bill	Davis	3	2020	1200.00	1
Bill	Davis	4	2020	1200.00	1
Bill	Davis	1	2021	1200.00	1
Bill	Davis	2	2021	1200.00	1
Betty	Esteban	2	2021	1790.00	1
Betty	Esteban	1	2021	1550.00	2
Betty	Esteban	4	2020	1387.00	3
Betty	Esteban	3	2020	1250.00	4
Betty	Esteban	2	2020	1125.00	5
Betty	Esteban	1	2020	925.00	6
Jack	Fisher	2	2021	2580.00	1
Jack	Fisher	1	2021	2201.00	2
Jen	Grant	2	2021	2121.00	1
Jen	Grant	1	2021	1994.00	2
Larry	Holmes	1	2021	502.00	1
Larry	Holmes	2	2021	387.00	2
Lily	Imprego	2	2021	1046.00	1
Lily	Imprego	1	2021	918.00	2

■ **FIGURE 19.14 Ranking within partitions.**

Choosing Windowing or Grouping for Ranking

Given the power and flexibility of SQL's windowing capabilities, is there any time that you should use grouping queries instead? Actually, there just might be. Assume that you want to rank all the sales representatives based on their total sales rather than simply ranking within each person's sales. Probably the easiest way to get that ordered result is to use a query like the following:

```
SELECT id, SUM (sales_amt)
FROM quarterly_sales
GROUP BY id
ORDER BY SUM (sales_amt) DESC;
```

You get the following output:

```
 id |   sum
----+--------
  6 | 8027.00
  5 | 7200.00
  2 | 7194.00
  1 | 5570.00
  7 | 4781.00
  3 | 4309.00
  4 | 4177.00
  8 | 4115.00
 10 | 1964.00
  9 |  889.00
```

The highest ranking total sales are at the top of listing, the lowest ranking sales at the bottom. The output certainly isn't as informative as the windowed output because you can't include the names of the sales representatives, but it does provide the required information.

Yes, you could use a windowing function to generate the same output, but it still needs to include the aggregate function SUM to generate the totals for each sales representative:

```
SELECT id, SUM (SUM(sales_amt))
     OVER (PARTITION BY quarterly_sales.id)
FROM quarterly_sales
GROUP BY id
ORDER BY SUM (sales_amt) DESC;
```

It works, but it's more code and the presence of the GROUP BY clause still means that you can't include the names unless they are part of the grouping criteria. Using the GROUP BY and the simple SUM function just seems easier.

PERCENT_RANK

The PERCENT_RANK function calculates the percentage rank of each value in a partition relative to the other rows in the partition. It works in the same way as RANK, but rather than returning a rank as an integer, it returns the percentage point at which a given value occurs in the ranking.

Let's repeat the query used to illustrate RANK using PERCENT_RANK instead:

```
SELECT first_name, last_name, quarter, year, sales_amt,
    PERCENT_RANK () OVER (PARTITION BY quarterly_sales.id
          ORDER BY sales_amt DESC)
FROM rep_names, quarterly_sales
WHERE rep_names.id = quarterly_sales.id;
```

The output can be found in Figure 19.15. As you can see, the result is exactly the same as the RANK result in Figure 19.14, with the exception of the rightmost column, where the integer ranks are replaced by percentage ranks.

ROW_NUMBER

The ROW_NUMBER function numbers the rows within a partition. For example, to number the sales representatives in alphabetical name order, the query could be

```
SELECT first_name, last_name,
     ROW_NUMBER () OVER (ORDER BY last_name, first_name)
     AS row_numb
FROM rep_names;
```

As you can see from Figure 19.16, the result includes all 10 sales representatives, numbered and sorted by name (last name as the outer sort).

Note: The SQL standard allows a named ROW_NUMBER result to be placed in a WHERE clause to restrict the number of rows in a query. However, not all DBMSs allow window functions in WHERE clauses.

CUME_DIST

When we typically think of a cumulative distribution, we think of something like that in Table 19.4, where the actual data values are gathered into ranges. SQL, however, can't discern the data grouping that we would like and therefore must consider each value (whether it be an individual data row or a row of an aggregate function result) as a line in the distribution.

The CUME_DIST function returns a value between 0 and 1, which, when multiplied by 100, gives you a percentage. Each "range" in the distribution, however, is a single value. In other words, the frequency of each group is

first_name	last_name	quarter	year	sales_amt	percent_rank
John	Anderson	3	2020	1206.00	0
John	Anderson	2	2021	1100.00	0.2
John	Anderson	2	2020	1009.00	0.4
John	Anderson	1	2021	915.00	0.6
John	Anderson	4	2020	822.00	0.8
John	Anderson	1	2020	518.00	1
Jane	Anderson	2	2021	1400.00	0
Jane	Anderson	1	2021	1380.00	0.2
Jane	Anderson	4	2020	1355.00	0.4
Jane	Anderson	3	2020	1235.00	0.6
Jane	Anderson	2	2020	1035.00	0.8
Jane	Anderson	1	2020	789.00	1
Mike	Baker	2	2021	1560.00	0
Mike	Baker	1	2021	1012.00	0.333333333333333
Mike	Baker	4	2020	942.00	0.666666666666667
Mike	Baker	3	2020	795.00	1
Mary	Carson	1	2020	1444.00	0
Mary	Carson	2	2020	1244.00	0.333333333333333
Mary	Carson	3	2020	987.00	0.666666666666667
Mary	Carson	4	2020	502.00	1
Bill	Davis	1	2020	1200.00	0
Bill	Davis	2	2020	1200.00	0
Bill	Davis	3	2020	1200.00	0
Bill	Davis	4	2020	1200.00	0
Bill	Davis	1	2021	1200.00	0
Bill	Davis	2	2021	1200.00	0
Betty	Esteban	2	2021	1790.00	0
Betty	Esteban	1	2021	1550.00	0.2
Betty	Esteban	4	2020	1387.00	0.4
Betty	Esteban	3	2020	1250.00	0.6
Betty	Esteban	2	2020	1125.00	0.8
Betty	Esteban	1	2020	925.00	1
Jack	Fisher	2	2021	2580.00	0
Jack	Fisher	1	2021	2201.00	1
Jen	Grant	2	2021	2121.00	0
Jen	Grant	1	2021	1994.00	1
Larry	Holmes	1	2021	502.00	0
Larry	Holmes	2	2021	387.00	1
Lily	Imprego	2	2021	1046.00	0
Lily	Imprego	1	2021	918.00	1

■ **FIGURE 19.15 Percent ranking within partitions.**

first_name	last_name	row_numb
Jane	Anderson	1
John	Anderson	2
Mike	Baker	3
Mary	Carson	4
Bill	Davis	5
Betty	Esteban	6
Jack	Fisher	7
Jen	Grant	8
Larry	Holmes	9
Lily	Imprego	10

■ **FIGURE 19.16 Row numbering.**

Table 19.4 A Cumulative Frequency Distribution

Sales amount	Frequency	Cumulative frequency	Cumulative percentage
$0–1999	2	2	20
$2000–3999	0	0	20
$4000–5999	5	7	70
$6000–7999	2	9	90
> $8000	1	10	100

always 1. As an example, let's create a cumulative frequency distribution of the total sales made by each sales representative. The SQL can be written

```
SELECT id, SUM (sales_amt),
     100 * (CUME_DIST() OVER (ORDER BY SUM (sales_amt)))
     AS cume_dist
FROM quarterly_sales
GROUP BY id
ORDER BY cume_dist;
```

As you can see in Figure 19.17, each range is a group of 1.

```
 id |   sum    | cume_dist
----+----------+-----------
  9 |  889.00 |        10
 10 | 1964.00 |        20
  8 | 4115.00 |        30
  4 | 4177.00 |        40
  3 | 4309.00 |        50
  7 | 4781.00 |        60
  1 | 5570.00 |        70
  2 | 7194.00 |        80
  5 | 7200.00 |        90
  6 | 8027.00 |       100
```

■ **FIGURE 19.17 A SQL-generated cumulative frequency distribution.**

NTILE

NTILE breaks a distribution into a specified number of partitions, and indicates which rows are part of which group. SQL keeps the numbers of rows in each group as equal as possible. To see how this works, consider the following query:

```
SELECT id, SUM (sales_amt),
     NTILE(2) OVER (ORDER BY SUM (sales_amt) DESC) as N2,
          NTILE(3) OVER (ORDER BY SUM (sales_amt) DESC) as N3, NTILE(4)
          OVER (ORDER BY SUM (sales_amt) DESC) as N4
FROM quarterly_sales
GROUP BY id;
```

For the result, see Figure 19.18. The columns labeled n2, n3, and n4 contain the results of the NTILE calls. The highest number in each of those columns corresponds to the number of groups into which the data have been placed, which is the same value used as an argument to the function call.

```
 id |    sum    | n2 | n3 | n4
----+-----------+----+----+----
  6 | 8027.00 |  1 |  1 |  1
  5 | 7200.00 |  1 |  1 |  1
  2 | 7194.00 |  1 |  1 |  1
  1 | 5570.00 |  1 |  1 |  2
  7 | 4781.00 |  1 |  2 |  2
  3 | 4309.00 |  2 |  2 |  2
  4 | 4177.00 |  2 |  2 |  3
  8 | 4115.00 |  2 |  3 |  3
 10 | 1964.00 |  2 |  3 |  4
  9 |  889.00 |  2 |  3 |  4
```

■ **FIGURE 19.18 Using the NTILE function to divide data into groups.**

Inverse Distributions: PERCENTILE_CONT and PERCENTILE_DISC

The SQL standard includes two inverse distribution functions—PERCENTILE_CONT and PERCENTILE_DISC—that are most commonly used to compute the median of a distribution. PERCENTILE_CONT assumes that the distribution is continuous, and interpolates the median as needed. PERCENTILE_DISC, which assumes a discontinuous distribution, chooses the median from existing data values. Depending on the data themselves, the two functions may return different answers.

The functions have the following general format:

```
PERCENTILE_cont/disc (0.5)
     WITHIN GROUP (optional ordering clause)
     OVER (optional partition and ordering clauses)
```

If you replace the 0.5 following the name of the function with another probability between 0 and 1, you will get the nth percentile. For example, 0.9 returns the 90th percentile. The functions examine the percent rank of the values in a partition until it finds the one that is equal to or greater than whatever fraction you've placed in parentheses.

When used without partitions, each function returns a single value. For example,

```
SELECT PERCENTILE_CONT (0.5) WITHIN GROUP
   (ORDER BY SUM (sales_amt) DESC) AS continuous,
      PERCENTILE_DISC (0.5) WITHIN
      GROUP (ORDER BY SUM (sales_amt DESC) as discontinuous
FROM quarterly_sales
GROUP BY id;
```

Given the sales data, both functions return the same value: 1200. (There are 40 values, and the two middle values are 1200. Even with interpolation, the continuous median computes to the same answer.)

If we partition the data by sales representative, then we can compute the median for each sales representative:

```
SELECT first_name, last_name, PERCENTILE_CONT (0.5) WITHIN GROUP
   (ORDER BY SUM (sales_amt) DESC) OVER (PARTITION BY id)
      AS continuous, PERCENTILE_DISC (0.5) WITHIN GROUP
      (ORDER BY SUM (sales_amt DESC) OVER (PARTITION BY id)
      AS discontinuous
FROM quarterly_sales JOIN rep_names
GROUP BY id
ORDER BY last_name, first_name;
```

As you can see in Figure 19.19, the result contains one row for each sales representative, including both medians.

first_name	last_name	continuous	discontinuous
John	Anderson	962.0	915.0
Jane	Anderson	1295.0	1235.0
Mike	Baker	977.0	942.0
Mary	Carson	1115.5	987.0
Bill	Davis	1200.0	1200.0
Betty	Esteban	1318.5	1250.0
Jack	Fisher	2350.5	2201.0
Jen	Grant	2057.5	1994.0
Larry	Holmes	484.5	387.0
Lily	Imprego	982.0	918.0

■ **FIGURE 19.19 Continuous and discontinuous medians for partitioned data.**

Page left intentionally blank

Chapter 20 Data Modification

SQL includes commands for modifying the data in tables: INSERT, UPDATE, and DELETE. In many business environments, application programs provide forms-driven data modification, removing the need for end users to issue SQL data modification statements directly to the DBMS. (As you will see, this is a good thing because using SQL data modification statements are rather clumsy.) Nonetheless, if you are developing and testing database elements and need to populate tables and modify data, you will probably be working at the command line with the SQL syntax.

Note: This chapter is where it will make sense that we covered retrieval before data modification.

Inserting Rows

The SQL INSERT statement has two variations: one that inserts a single row into a table and a second that copies one or more rows from another table.

Inserting One Row

To add one row to a table, you use the general syntax

```
INSERT INTO table_name VALUES (value_list)
```

In the preceding form, the value list contains one value for every column in the table, in the order in which the columns were created. For example, to insert a new row into the *customer* table someone at the rare book store might use

```
INSERT INTO customer VALUES (8,'Helen','Jerry',
     '16 Main Street','Newtown','NJ','18886','209-555-8888');
```

There are two things to keep in mind when inserting data in this way:

- The format of the values in the value list must match the data types of the columns into which the data will be placed. In the current example, the first column requires an integer. The remaining columns all require characters and therefore the values have been surrounded by single quotes.
- When you insert a row of data, the DBMS checks any integrity constraints that you have placed on the table. For the preceding example, it will verify that the customer number is unique and not null. If the constraints are not met, you will receive an error message and the row will not be added to the table.

If you do not want to insert data into every column of a table, you can specify the columns into which data should be placed:

```
INSERT INTO table_name (column_list) VALUES (value_list)
```

There must be a one-to-one correspondence between the columns in the column list and the values in the value list because the DBMS matches them by their relative positions in the lists.

As an example, assume that someone at the rare book store wants to insert a row into the *book* table, but doesn't know the binding type. The SQL would then be written

```
INSERT INTO book (isbn, work_numb, publisher_id,
    edition, copyright_year)
    VALUES ('978-11111-100-1',16,2,12,1960);
```

There are five columns in the column list and therefore five values in the value list. The first value in the list will be inserted into the *isbn* column, the second value into the *work_numb* column, and so on. The column omitted from the lists—*binding*—will remain null. You therefore must be sure to place values at least in primary key columns. Otherwise, the DBMS will not permit the insertion to occur.

Although it is not necessary to list column names when inserting values for every column in a table, there is one good reason to do so, especially when embedding the INSERT statement in an application program. If the structure of the table changes—if columns are added, deleted, or rearranged—then an INSERT without column names will no longer work properly. By always specifying column names, you can avoid unnecessary program modifications as your database changes to meet your changing needs.

Placement of New Rows

Where do new rows go when you add them? That depends on your DBMS. Typically, a DBMS maintains a unique internal identifier for each row that is not accessible to users (something akin to the combination of a row number and a table identifier) to provide information about the row's physical storage location. These identifiers continue to increase in value.

If you were to use the SELECT * syntax on a table, you would see the rows in internal identifier order. At the beginning of a table's life, this order corresponds to the order in which rows were added to the table. New rows appear to go at the "bottom" of the table, after all existing rows. As rows are deleted from the table, there will be gaps in the sequence of row identifiers. However, the DBMS does not reuse them (to "fill in the holes") until it has used up all available identifiers. If a database is very old, very large, and/or very active, the DBMS will run out of new identifiers and will then start to reuse those made available by deleted rows. In that case, new rows may appear anywhere in the table. Given that you can view rows in any order by using the ORDER BY clause, it should make absolutely no difference to an end user or an application program where a new row is added.

Copying Existing Rows

The SQL INSERT statement can also be used to copy one or more rows from one table to another. The rows that will be copied are specified with a SELECT, giving the statement the following general syntax:

```
INSERT INTO table_name SELECT complete_SELECT_statement
```

The columns in the SELECT must match the columns of the table. For the purposes of this example, we will add a simple table to the rare book store database:

```
summary (isbn, how_many)
```

This table will contain summary information gathered from the *volume* table. To add rows to the new table, the INSERT statement can be written:

```
INSERT INTO summary
SELECT isbn, COUNT (*)
FROM volume
GROUP BY isbn;
```

The result is 27 rows copied into the *summary* table, one for each unique ISBN in the *volume* table.

Note: Should you store summary data like that placed in the table created in the preceding example? The answer is "it depends." If it takes a long time to

generate the summary data and you use the data frequently, then storing it probably makes sense. But if you can generate the summary data easily and quickly, then it is just as easy not to store it and to create the data whenever it is needed for output.

Updating Data

Although most of today's end users modify existing data using an on-screen form, the SQL statements to modify the data must nonetheless be issued by the program providing the form. For example, as someone at the rare book store adds volumes to a sale, the *volume* table is updated with the selling price and the sale ID. The selling price is also added to the total amount of the sale in the *sale* table.

The SQL UPDATE statement affects one or more rows in a table, based on row selection criteria in a WHERE predicate. UPDATE has the following general syntax:

```
UPDATE table_name
SET column1 = new_value, column2 = new_value, …
WHERE row_selection_predicate
```

If the WHERE predicate contains a primary key expression, then the UPDATE will affect only one row. For example, to change a customer's address, the rare book store could use

```
UPDATE customer
SET street = '195 Main Street'
      city = 'New Town'
      zip = '11111'
WHERE customer_numb = 5;
```

However, if the WHERE predicate identifies multiple rows, each row that meets the criteria in the predicate will be modified. To raise all $50 prices to $55, someone at the rare book store might write a query as

```
UPDATE books
SET asking_price = 55
WHERE asking_price = 50;
```

Notice that it is possible to modify the value in a column being used to identify rows. The DBMS will select the rows to be modified before making any changes to them.

If you leave the WHERE clause off an UPDATE, the same modification will be applied to every row in the table. For example, assume that we add a column for sales tax to the *sale* table. Someone at the rare book store could use the following statement to compute the tax for every sale:

```
UPDATE sale
SET sales_tax = sale_total_amt * 0.075;
```

The expression in the SET clause takes the current value in the *sale_total_amt* column, multiplies it by the tax rate, and puts it in the *sales_tax* column.

Deleting Rows

Like the UPDATE statement, the DELETE statement affects one or more rows in a table based on row selection criteria in a WHERE predicate. The general syntax for DELETE is

```
DELETE FROM table_name
WHERE row_selection_predicate
```

For example, if a customer decided to cancel an entire purchase, then someone at the rare book store would use something like

```
DELETE FROM sale
WHERE customer_numb = 12 AND sale_date = '05-Jul-2021';
```

Assuming that all purchases on the same date are considered a single sale, the WHERE predicate identifies only one row. Therefore, only one row is deleted.

When the criteria in a WHERE predicate identify multiple rows, all those matching rows are removed. If someone at the rare book store wanted to delete all sales for a specific customer, then the SQL would be written

```
DELETE FROM sale
WHERE customer_numb = 6;
```

In this case, there are multiple rows for customer number 6, all of which will be deleted.

DELETE is a potentially dangerous operation. If you leave off the WHERE clause—

```
DELETE FROM sale
```

—you will delete every row in the table! (The table remains in the database without any rows.)

Deletes and Referential Integrity

The preceding examples of DELETE involve a table that has a foreign key in another table (*sale_id* in *volume*) referencing it. It also has a foreign key of its own (*customer_numb* referencing the primary key of *customer*). You can delete rows containing foreign keys without any effect on the rest of the database, but what happens when you attempt to delete rows that *do* have foreign keys referencing them?

Note: The statement in the preceding paragraph refers to database integrity issues and clearly misses the logical issue of the need to decrement the total sale amount in the sale table whenever a volume is removed from the sale.

Assume, for example, that a customer cancels a purchase. Your first thought might be to delete the row for that sale from the *sale* table. There are, however, rows in the *volume* table that reference that sale and if the row for the sale is removed from *sale*, there will be no primary key for the rows in *volume* to reference and referential integrity will be violated.

As you will remember from our discussion of creating tables with foreign keys, what actually happens in such a situation depends on what was specified when the table containing the primary key being referenced was created (SET NULL, SET DEFAULT, CASCADE, NO ACTION).

Deleting All Rows: TRUNCATE TABLE

The 2008 SQL standard introduced a new command—TRUNCATE TABLE—that removes all rows from a table more quickly than a DELETE, without a WHERE clause. This can be particularly useful when you are developing a database and need to remove test data but leave the table structure untouched. The command's general syntax is

```
TRUNCATE TABLE table_name
```

Like the DELETE without a WHERE clause, the table structure remains intact, and in the data dictionary.

There are some limits to using the command:

- It cannot be used on a table that has foreign keys referencing it.
- It cannot be used on a table on which indexed views are based.
- It cannot activate a trigger (a program module stored in the database that is initiated when a specific event, such as a delete, occurs).

Although DELETE and TRUNCATE TABLE seem to have the same effect, they do work differently. DELETE removes the rows one at a time and writes an entry into the database log file for each row. In contrast, TRUNCATE TABLE deallocates space in the database files, making the space formerly occupied by the truncated table's rows available for other use.

Inserting, Updating, or Deleting on a Condition: MERGE

The SQL:2003 standard introduced a very powerful and flexible way to insert, update, or delete data using the MERGE statement. MERGE[1] includes a condition to be tested and alternative sets of actions that are performed when the condition is or is not met. The model behind this statement is the merging of a table of transactions into a master table.

MERGE has the following general syntax:

```
MERGE INTO target_table_name
      USING source_table_name ON merge_condition
WHEN MATCHED THEN
      update/delete_specification
WHEN NOT MATCHED THEN
      insert specification
```

Notice that when the merge condition is matched (in other words, evaluates as true for a given row), an update and/or delete is performed. When the condition is not matched, an insert is performed. Either the MATCHED or NOT MATCHED clause is optional.

The target table is the table that will be affected by the changes made by the statement. The source table—which can be a base table or a virtual table generated by a SELECT—provides the source of the table. To help you understand how MERGE works, let's use the classic model of applying transactions to a master table. First, we need a transaction table:

```
transaction (sale_id, inventory_id, selling_price,
      sale_date, customer_numb)
```

The *transaction* table contains information about the sale of a single volume. (It really doesn't contain all the necessary columns for the *sale* table but it will do for this example.) If a row for the sale exists in the *sale* table, then the selling price of the volume should be added to existing sale total. However, if the sale is not in the *sale* table, then a new row should be created and the sale total set to the selling price of the volume. A MERGE statement that will do the trick might be written as

```
MERGE INTO sale S USING transaction T ON (S.sale_id = T.sale_id)
WHEN MATCHED THEN
      UPDATE SET sale_total_amt = sale_total_amt + selling_price
WHEN NOT MATCHED
      INSERT (sale_id, customer_numb, sale_date, sale_total_amt)
           VALUES (T.sale_id, T.customer_numb, T.sale_date,
           T.selling_price);
```

[1]Some DBMSs call this functionality UPSERT.

The target table is *sale*; the source table is *transaction*. The merge condition looks for a match between sale IDs. If a match is found, then the UPDATE portion of the command performs the modification of the *sale_total_amt* column. If no match is found, then the insert occurs. Notice that the INSERT portion of the command does not need a table name because the table affected by the INSERT has already been specified as the target table.

As we said earlier, the source table for a merge operation doesn't need to be a base table; it can be a virtual table created on the fly using a SELECT. For example, assume that someone at the rare book store needs to keep a table of total purchases made by each customer. The following table can be used to hold that data:

```
summary_stats (customer_numb, year, total_purchases)
```

You can find the MERGE statement below. The statement assembles the summary data using a SELECT that extracts the year from the sale date and sums the sale amounts. Then, if a summary row for a year already exists in *summary_stats*, the MERGE adds the amount from the source table to what is stored already in the target table. Otherwise, it adds a row to the target table.

```
MERGE INTO summary_stats AS S USING
      (SELECT customer_numb,
      EXTRACT (YEAR FROM sale_date) AS Y, sum
(sale_total_amt AS M) AS T
      FROM sale
      GROUP BY customer_numb, Y)
ON (CAST(S.customer_numb AS CHAR (4)
      || CAST (S.year AS CHAR(4) =
CAST(T.customer_numb AS CHAR (4) || CAST (T.year AS CHAR(4)
WHEN MATCHED
      UPDATE SET total_purchases = T.M
WHEN NOT MATCHED
      INSERT VALUES (customer_numb, Y, M);
```

As powerful as MERGE seems to be, the restriction of UPDATE/DELETE to the matched condition and INSERT to the unmatched prevent it from being able to handle some situations. For example, if someone at the rare book store wanted to archive all orders more than two years old, the process would involve creating a row for each sale that didn't exist in the archive table and then deleting the row from the *sale* table. (We're assuming that the delete cascades, removing all rows from *volume* as well.) The problem is that the delete needs to occur on the unmatched condition, which isn't allowed with the MERGE syntax.

Chapter 21 Creating Additional Structural Elements

Views

As you first read in Chapter 5, views provide a way to give users a specific portion of a larger schema with which they can work. Before you actually can create views, there are two things you should consider: which views you really need and whether the views can be used for updating data.

Deciding Which Views to Create

Views take up very little space in a database, occupying only a few rows in a data dictionary table. That being the case, you can feel free to create views as needed.

A typical database might include the following views:

- One view for every base table that is exactly the same as the base table, but with a different name. Then, you prevent end users from seeing the base tables and do not tell the end users the table names; you give end users access only to the views. This makes it harder for end users to attempt to gain access to the stored tables because they do not know their names. However, as you will see in the next section, it is essential for updating that there be views that do match the base tables.
- One view for each primary key–foreign key relationship over which you join frequently. If the tables are large, the actual syntax of the statement may include structures for avoiding the join operation, but still combining the tables.
- One view for each complex query that you issue frequently.
- Views as needed to restrict user access to specific columns and rows. For example, you might recreate a view for a receptionist that shows employee office numbers and telephone extensions, but leaves off home address, telephone number, and salary.

View Updatability Issues

A database query can apply any operations supported by its DBMS's query language to a view, just as it can to base tables. However, using views for updates is a much more complicated issue. Given that views exist only in main memory, any updates made to a view must be stored in the underlying base tables if the updates are to have any effect on the database.

Not every view is updatable, however. Although the rules for view updatability vary from one DBMS to another, you will find that many DBMSs share the following restrictions:[1]

- A view must be created from one base table or view, or if the view uses joined tables, only one of the underlying base tables can be updated.
- If the source of the view is another view, then the source view must also adhere to the rules for updatability.
- A view must be created from only one query. Two or more queries cannot be assembled into a single view table using operations such as union.
- The view must include the primary key columns of the base table.
- The view must include all columns specified as not null (columns requiring mandatory values).
- The view must not include any groups of data. It must include the original rows of data from the base table, rather than rows based on values common to groups of data.
- The view must not remove duplicate rows.

Creating Views

To create a view whose columns have the same name as the columns in the base tables from which it is derived, you give the view a name and include the SQL query that defines its contents:

```
CREATE VIEW view_name AS
    SELECT …
```

[1]MySQL views are updatable if they adhere to the rules stated in the text, as are many DB2 views; PostgreSQL views aren't updatable at all. However, you can update using views that don't meet the rules for updatability. If you write a trigger—a SQL procedure executed when some modification event occurs—using INSTEAD OF for a view, you can create an alternative for the modification command that works directly on the base tables. The user still performs the update command on the view, but, instead of doing the update, the DBMS activates the trigger to actually do the work. Triggers require SQL programming (see Appendix B). In addition, the syntax for INSTEAD OF triggers is highly implementation-dependent.

For example, if *Antique Opticals* wanted to create a view that included actions films, the SQL is written

```
CREATE VIEW action_films AS
    SELECT item_numb, title
    FROM item
    WHERE genre = 'action';
```

If you want to rename the columns in the view, you include the view's column names in the CREATE VIEW statement:

```
CREATE VIEW action_films (identifier, name)
    AS
    SELECT item_numb, title
    FROM item
    WHERE genre = 'action';
```

The preceding statement will produce a view with two columns named *identifier* and *name*. Note that if you want to change even one column name, you must include *all* the column names in the parentheses following the view name. The DBMS will match the columns following SELECT with the view column names by their positions in the list.

Views can be created from any SQL query, including those that perform joins, unions, and grouping. For example, to simplify looking at customers and their order totals, *Antique Opticals* might create a view like the following:

```
CREATE VIEW sales_summary AS
    SELECT customer_numb, order.order_numb, order.order_date,
      SUM (selling_price)
    FROM order_line JOIN order
    GROUP BY customer_number, orders.order_date,
      orders.order_numb;
```

The view table will then contain grouped data along with a computed column.

Temporary Tables

A temporary table is a base table that is not stored in the database, but instead exists only while the database session in which it was created is active. At first glance, this may sound like a view, but views and temporary tables are somewhat different:

- A view exists only for a single query. Each time you use the name of a view, its table is recreated from existing data.
- A temporary table exists for the entire database session in which it was created.
- A view is automatically populated with the data retrieved by the query that defines it.

- You must add data to a temporary table with SQL INSERT commands.
- Only views that meet the criteria for view updatability can be used for data modifications.
- Because temporary tables are base tables, all of them can be updated. (They can be updated, but keep in mind that any modifications you make won't be retained in the database.)
- Because the contents of a view are generated each time the view's name is used, a view's data are always current.
- The data in a temporary table reflect the state of the database at the time the table was loaded with data. If the data from which the temporary table was loaded are modified after the temporary table has received its data, then the contents of the temporary table may be out of sync with other parts of the database.

If the contents of a temporary table become outdated when source data change, why use a temporary table at all? Wouldn't it be better simply to use a view whose contents are continually regenerated? The answer lies in performance. It takes processing time to create a view table. If you are going to use data only once during a database session, then a view will actually perform better than a temporary table because you don't need to create a structure for it. However, if you are going to be using the data repeatedly during a session, then a temporary table provides better performance because it needs to be created only once. The decision therefore results in a trade-off: Using a view repeatedly takes more time, but provides continuously updated data; using a temporary table repeatedly saves time, but you run the risk that the table's contents may be out of date.

Creating Temporary Tables

Creating a temporary table is very similar to creating a permanent base table. You do, however, need to decide on the *scope* of the table. A temporary table may be *global*, in which case it is accessible to the entire application program that created it.[2] Alternatively, it can be *local*, in which case it is accessible only to the program module in which it was created.

To create a global temporary table, you add the keywords GLOBAL TEMPORARY to the CREATE TABLE statement:

```
CREATE GLOBAL TEMPORARY TABLE
    (remainder of CREATE statement)
```

[2]The term "application program" refers to a program written by programmers for end-users or to a command-line SQL command processor that a user is using interactively.

By the same token, you create a local temporary table with

```
CREATE LOCAL TEMPORARY TABLE
    (remainder of CREATE statement)
```

For example, if *Antique Opticals* was going to use the order summary information repeatedly, it might create the following temporary table instead of using a view:

```
CREATE GLOBAL TEMPORARY TABLE order_summary
    (customer_numb int,
    order_numb int,
    order_date date,
    order_total numeric (6,2),
    PRIMARY KEY (customer_numb, order_numb);
```

Loading Temporary Tables with Data

To place data in a temporary table, you use one or more SQL INSERT statements. For example, to load the order summary table created in the preceding section, you could type

```
INSERT INTO order_summary
    SELECT customer_numb, order.order_numb, order.order_date,
      SUM (selling_price)
    FROM order_line JOIN order
    GROUP BY customer_number, orders.order_date,
      orders.order_numb
```

You can now query and manipulate the *order_summary* table, just as you would a permanent base table.

Disposition of Temporary Table Rows

When you write embedded SQL (SQL statements coded as part of a program written in a high-level language such as C++ or Java), you have control over the amount of work that the DBMS considers to be a unit (a *transaction*). Although we will cover transactions in depth in Chapter 22, at this point you need to know that a transaction can end in one of two ways: It can be *committed* (changes made permanent) or it can be *rolled back* (its changes undone).

By default, the rows in a temporary table are purged whenever a transaction is committed. If you want to use the same temporary tables in multiple transactions, however, you can instruct the DBMS to retain the rows by including ON COMMIT PRESERVE ROWS to the end of the table creation statement:

```
CREATE GLOBAL TEMPORARY TABLE order_summary
    (customer_numb int,
    order_numb int,
    order_date date,
    order_total numeric (6,2),
    PRIMARY KEY (customer_numb, order_numb
    ON COMMIT PRESERVE ROWS);
```

Because a rollback returns the database to the state it was in before the transaction begins, a temporary table will also be restored to its previous state (with or without rows).

Common Table Expressions (CTEs)

A *common table expression* (CTE) is yet another way of extracting a subset of a database for use in another query. CTEs are like views in that they generate virtual tables. However, the definition of a CTE is not stored in the database, and it must be used immediately after it is defined.

To get started, let's look at a very simple example. The general format of a simple CTE is

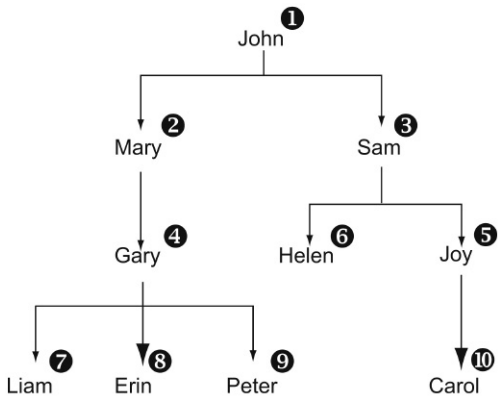
```
WITH CTE_name (columns) AS
      (SELECT_statement_defining_table)
CTE_query
```

For example, a CTE and its query to view all of the rare book store's customers could be written

```
WITH customer_names (first, last) AS
      (SELECT first_name, last_name
       FROM customer)
SELECT *
FROM customer_names;
```

The result is a listing of the first and last names of the customers. This type of structure for a simple query really doesn't buy you much, except that the CTE isn't stored in the database like a view and doesn't require INSERT statements to populate it like a temporary table. However, the major use of CTEs is for *recursive queries*, queries that query themselves. (That may sound a bit circular and it is, intentionally.) The typical application of a recursive query using a CTE is to process hierarchical data, data arranged in a tree structure. It will allow a single query to access every element in the tree or to access subtrees that begin somewhere other than the top of the tree.

As an example, let's create a table that handles the descendants of a single person (in this case, John). As you can see in Figure 21.1, each node in the tree has at most one parent and any number of children. The numbers in the illustration represent the ID of each person.



■ **FIGURE 21.1 A tree structure that can be represented in a relational database, and traversed with a recursive query.**

Relational databases are notoriously bad at handling this type of hierarchically structured data. The typical way to handle it is to create a relation, something like this:

```
genealogy (person_id, parent_id, person_name)
```

Each row in the table represents one node in the tree. For this example, the table is populated with the 10 rows in Figure 21.2. John, the node at the top of the tree, has no parent ID. The *parent_ID* column in the other rows is filled with the person ID of the node above it in the tree. (The order of the rows in the table is irrelevant.)

```
 person_id | parent_id |           person_name
-----------+-----------+---------------------------------
         1 |           | John
         2 |         1 | Mary
         3 |         1 | Sam
         4 |         2 | Gary
         5 |         3 | Joy
         6 |         3 | Helen
         7 |         4 | Liam
         8 |         4 | Erin
         9 |         4 | Peter
        10 |         5 | Carol
```

■ **FIGURE 21.2 Sample data for use with a recursive query.**

We can access every node in the tree by simply accessing every row in the table. However, what can we do if we want to process just the people who are Sam's descendants? There is no easy way with a typical SELECT to do that. However, a CTE used recursively will identify just the rows we want.

The syntax of a recursive query is similar to the simple CTE query, with the addition of the keyword RECURSIVE following WITH. For our particular example, the query will be written:

```
WITH RECURSIVE show_tree AS
      (SELECT
      FROM genealogy
      WHERE person_name = 'Sam'
      UNION ALL
      SELECT g.*
      FROM genealogy AS g, show_tree AS st
      WHERE g.parent_id = st.person_id)
SELECT *
FROM show_tree
ORDER BY person_name;
```

The result is

```
 person_id | parent_id |       person_name
-----------+-----------+--------------------------
        10 |         5 | Carol
         6 |         3 | Helen
         5 |         3 | Joy
         3 |         1 | Sam
```

The query that defines the CTE called *show_tree* has two parts. The first is a simple SELECT that retrieves Sam's row and places it in the result table as well as in an intermediate table that represents the current state of *show_tree*. The second SELECT (below UNION ALL) is the recursive part. It will use the intermediate table in place of *show_tree* each time it executes and add the result of each iteration to the result table. The recursive portion will execute repeatedly until it returns no rows.

Here's how the recursion will work in our example:

1. Join the intermediate result table to *genealogy*. Because the intermediate result table contains just Sam's row, the join will match Helen and Joy.
2. Remove Sam from the intermediate table and insert Helen and Joy.
3. Append Helen and Joy to the result table.

4. Join the intermediate table to *genealogy*. The only match from the join will be Carol. (Helen has no children, and Joy has only one.)
5. Remove Helen and Joy from the intermediate table and insert Carol.
6. Append Carol to the result table.
7. Join the intermediate table to *genealogy*. The result will be no rows, and the recursion stops.

CTEs cannot be reused; the declaration of the CTE isn't saved. Therefore, they don't buy you much for most queries. However, they are enormously useful if you are working with tree-structured data. CTEs and recursion can also be helpful when working with bill of materials data.

Creating Indexes

As you read in Chapter 7, an index is a data structure that provides a fast access path to rows in a table based on the value in one or more columns (the index key). Because an index stores key values in order, the DBMS can use a fast search technique to find the values, rather than being forced to search each row in an unordered table sequentially.

You create indexes with the CREATE INDEX statement:

```
CREATE INDEX index_name
ON table_name (index_key_columns)
```

For example, to create an index on the *title* column in *Antique Opticals item* table, you could use

```
CREATE INDEX item_title_index
ON item (title);
```

By default, the index will allow duplicate entries and keeps the entries in ascending order (alphabetical, numeric, or chronological, whichever is appropriate). To require unique indexes, add the keyword UNIQUE after CREATE:

```
CREATE UNIQUE INDEX item_title_index
ON item (title);
```

To sort in descending order, insert DESC after the column whose sort order you want to change. For example, *Antique Opticals* might want to create an index on the order date in the *order* relation in descending order so that the most recent orders are first:

```
CREATE INDEX order_order_date_index
ON order (order_date DESC);
```

If you want to create an index on a concatenated key, include all the columns that should be part of the index key in the column list. For example, the following creates an index organized by actor and item number:

```
CREATE INDEX actor_actor_item_index
ON actor (actor_numb, item_numb);
```

Although you do not need to access an index directly unless you want to delete it from the database, it helps to give indexes names that will tell you something about their tables and key columns. This makes it easier to remember them should you need to get rid of the indexes.