CHAPTER 17

Retrieving Data from More Than One Table

Abstract

This chapter presents SQL syntax for retrieving data from more than one table in a single query. Techniques include basic joins (original and modern syntax), outer joins, joining a table to itself, table constructors, and uncorrelated subqueries.

Keywords

SQL
SQL SELECT
SQL joins
SQL retrieval
outer join
subqueries
SQL IN operator
SQL ANY operator
multi-table queries

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
```

```
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*.

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

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.

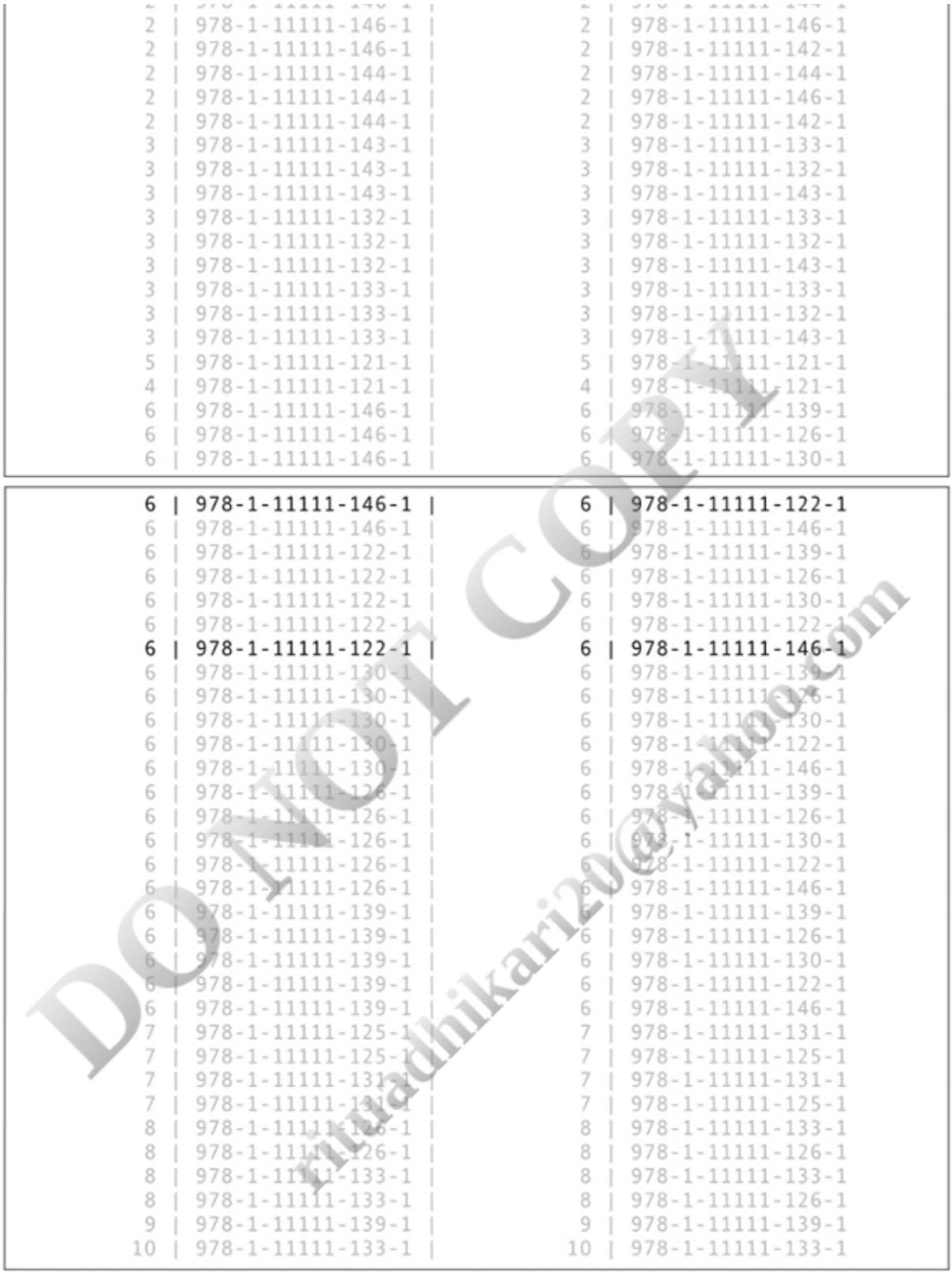
Before looking at the SQL syntax, however, let's examine the relational algebra of the joins so you can see exactly what is happening. Assume that we are working with the subset of the *volume* table in Figure 17.2. (The sale ID and the ISBN are the only columns that affect the relational algebra; the 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.

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
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 **The result of joining the table in Figure 17.2 to itself.**

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).

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
```

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 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.

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.**

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

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.[1]

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.

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
```

```
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'));
```

```
                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.

[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.

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