CHAPTER 18

Advanced Retrieval Operations

Abstract

This chapter covers advanced retrieval operations such as unions, negative queries, special operators (EXISTS, EXCEPT, AND INTERSECTION), arithmetic in SQL queries, string manipulation, and date and time manipulation.

Keywords

SQL
SQL retrieval
SQL UNION
SQL EXISTS
SQL EXCEPT
SQL INSTERSECT
SQL arithmetic
SQL string manipulation
SQL date manipulation
SQL time manipulation

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
```

```
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
```

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

```
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);
```

```
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
```

```
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.**

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 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

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	Output from the query using INTERSECT
first_name | last_name ----------+---------- Edna | Hayes Helen | Jerry	first_name | last_name ----------+---------- Franklin | Hayes Janice | Jones

Jane	Doe
Jane	Smith
Janice	Smith
Jon	Jones
Mary	Collins
Peter	Collins

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.

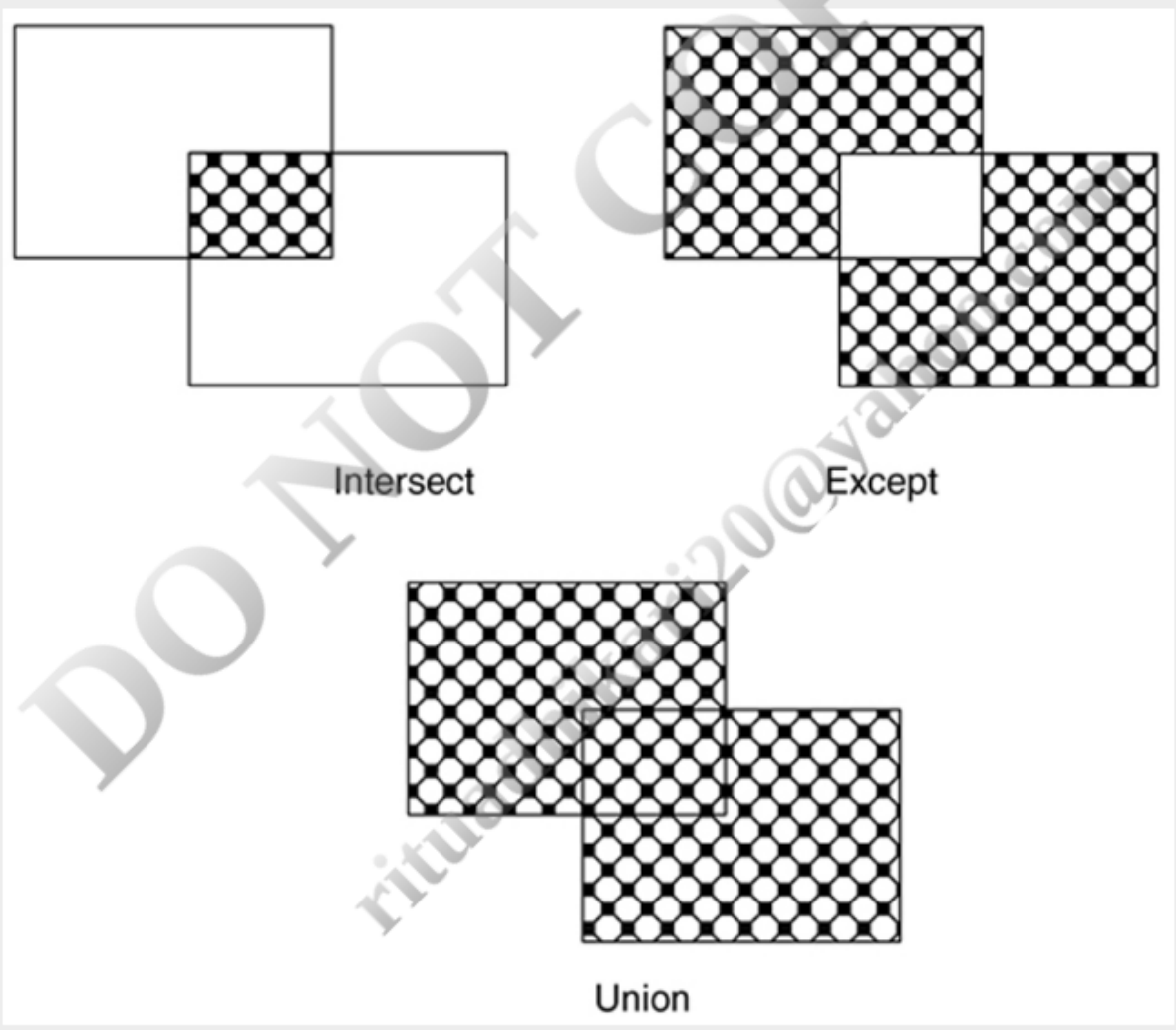

FIGURE 18-6 **Operation of the SQL INTERSECT, EXCEPT, and UNION operators.**

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
```

```
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
```

```
 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.

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

- 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

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.**

[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.)

[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.

[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.