THE LANGUAGE of SQL

THIRD EDITION

LARRY ROCKOFF

The Language of SQL

Third Edition

Larry Rockoff

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Looking Ahead

This chapter introduced the topic of how to apply selection criteria to queries. We introduced several basic operators, such as equals and greater than. The ability to specify these types of basic selection criteria goes a long way toward making the SELECT statement truly useful. We also covered the related topic of limiting the number of rows returned in a query. The ability to limit rows in combination with an ORDER BY clause allows for a useful Top N type of data selection.

Next, we discussed how to match words or phrases via a specified pattern. Matching by patterns is a significant and widely used function of SQL. Any time you enter a word in a search box and attempt to retrieve all entities containing that word, you are utilizing pattern matching. We concluded the chapter with a look at matching by sound, a less common practice than matching by word patterns. The technology exists, but there is an inherent difficulty in translating words to sounds, due to the many quirks and exceptions in the English language.

In the next chapter, “Boolean Logic,” we’ll greatly enhance our selection criteria capabilities by introducing several new keywords that add sophisticated logic to the WHERE clause. At present, we can do such things as select all customers from the state of New York. In the real world, however, more selection criteria are typically required. Boolean logic will allow us to formulate a query that will select customers who are in New York or California but not in Los Angeles or Albuquerque.

7 Boolean Logic

Keywords Introduced

AND · OR · NOT · BETWEEN · IN · IS NULL

We introduced the concept of selection criteria in the previous chapter, but only in its simplest form. We'll now expand on that concept to greatly enhance our ability to specify the rows returned from a SELECT. This is where the pure logic of SQL comes into play. In this chapter, we'll introduce several operators that will allow you to create complex logical expressions.

Given these new capabilities, if someone should request a list of all female customers who live in zip codes 60601 through 62999 but excluding anyone under the age of 30 or who doesn't have an email address, that will be something you can provide.

Complex Logical Conditions

The WHERE clause introduced in the previous chapter used only simple selection criteria. We saw clauses such as:

```
WHERE QuantityPurchased = 5
```

The condition expressed in this WHERE clause is quite basic. It specifies merely to return all rows for which the QuantityPurchased column has a value of 5. In the real world, the selection of data is often much less straightforward. Accordingly, let's now turn our attention to methods of specifying some more complex logical conditions in selection criteria.

The ability to devise complex logical conditions is sometimes called *Boolean logic*. This term, taken from mathematics, refers to the ability to formulate complex conditions that are evaluated as either true or false. In the example, the condition QuantityPurchased = 5 is evaluated as either true or false for each row in the table. Obviously, we want to see only rows where the condition evaluates as true.

The principal keywords used to create complex Boolean logic are AND, OR, and NOT. These three operators are used to provide additional functionality to the WHERE clause. In proper combination, the AND, OR, and NOT operators, along with parentheses, can specify just about any logical expression that can be imagined.

The AND Operator

The following examples will be taken from this Purchases table:

PurchaseID	CustomerName	State	QuantityPurchased	PricePerItem
1	Kim Chiang	IL	4	2.50
2	Sandy Harris	CA	10	1.25
3	James Turban	NY	5	4.00

Here's an example of a WHERE clause that uses the AND operator:

```
SELECT
CustomerName,
QuantityPurchased
FROM Purchases
WHERE QuantityPurchased > 3
AND QuantityPurchased < 7
```

The AND clause means that all conditions must evaluate to true for the row to be selected. This SELECT specifies that the only rows to be retrieved are those for which the QuantityPurchased is both greater than 3 and less than 7. Therefore, only these two rows are returned:

CustomerName	QuantityPurchased
Kim Chiang	4
James Turban	5

Notice that the row for Sandy Harris is not returned. Why? Sandy purchased a quantity of 10, which, in fact, does satisfy the first condition (QuantityPurchased > 3). However, the second condition (QuantityPurchased < 7) is not satisfied and therefore is not true. When using the AND operator, all conditions specified must be true for the row to be selected.

The OR Operator

Let's now look at the OR operator. The AND operator meant that *all* conditions must evaluate to true for the row to be selected. The OR operator means that the row will be selected if *any* of the conditions are determined to be true.

Here's an example, taken from the same table:

```
SELECT
CustomerName,
QuantityPurchased,
PricePerItem
FROM Purchases
WHERE QuantityPurchased > 8
OR PricePerItem > 3
```

This SELECT returns this data:

CustomerName	QuantityPurchased	PricePerItem
Sandy Harris	10	1.25
James Turban	5	4.00

Why are the rows for Sandy Harris and James Turban displayed, and not the row for Kim Chiang? The row for Sandy Harris is selected because it meets the requirements of the first condition (QuantityPurchased > 8). It doesn't matter that the second condition (PricePerItem > 3) isn't true, because only one condition needs to be true for an OR condition.

Likewise, the row for James Turban is selected because the second condition (PricePerItem > 3) is true for that row. The row for Kim Chiang isn't selected because it doesn't satisfy either of the two conditions.

Using Parentheses

Suppose we are interested only in orders from customers from either the state of Illinois or the state of California. Additionally, we want to see orders only where the quantity purchased is greater than 8. To attempt to satisfy this request, we might put together this SELECT statement:

```
SELECT
CustomerName,
State,
QuantityPurchased
FROM Purchases
WHERE State = 'IL'
OR State = 'CA'
AND QuantityPurchased > 8
```

We would expect this statement to return only one row of data, for Sandy Harris. Although we have two rows for customers in Illinois or California (Chiang and Harris), only one of those (Harris) has a quantity purchased greater than 8. However, when this statement is executed, we get the following:

CustomerName	State	QuantityPurchased
Kim Chiang	IL	4
Sandy Harris	CA	10

We see two rows instead of the expected one row. What went wrong? The answer lies in how SQL interprets the WHERE clause, which happens to contain both AND and OR operators. Like other computer languages, SQL has a predetermined order of evaluation that specifies the order in which various operators are interpreted. Unless told otherwise, SQL always processes the

AND operator before the OR operator. So in the previous statement, it first looks at the AND and evaluates the condition:

```
State = 'CA'
AND QuantityPurchased > 8
```

The row that satisfies that condition is for Sandy Harris. SQL then evaluates the OR operator, which allows for rows where the State equals IL. That adds the row for Kim Chiang. The result is that SQL determines that both the Kim Chiang and the Sandy Harris rows meet the condition.

Obviously, this isn't what was intended. This type of problem often comes up when AND and OR operators are combined in a single WHERE clause. The way to resolve the ambiguity is to use parentheses to specify the desired order of evaluation. Anything in parentheses is always evaluated first.

Here's how parentheses can be added to the previous SELECT to correct the situation:

```
SELECT
CustomerName,
State,
QuantityPurchased
FROM Purchases
WHERE (State = 'IL'
OR State = 'CA')
AND QuantityPurchased > 8
```

When this is executed, we see this data:

CustomerName	State	QuantityPurchased
Sandy Harris	CA	10

The parentheses in the SELECT statement force the OR expression (State = 'IL' OR State = 'CA') to be evaluated first. This produces the intended result.

Multiple Sets of Parentheses

Let's say we want to select two different sets of rows from the Purchases table: first, rows for customers in New York, and second, rows for customers in Illinois who have made a purchase with a quantity between 3 and 10. The following SELECT accomplishes this requirement:

```
SELECT
CustomerName,
State,
QuantityPurchased
FROM Purchases
WHERE State = 'NY'
OR (State = 'IL'
AND (QuantityPurchased >= 3
AND QuantityPurchased <= 10))
```