Ryan Stephens
Ron Plew
Arie D. Jones

The Fifth Edition of
Sams Teach Yourself SQL in 21 Days

More than 48,000 sold!

Sams **Teach Yourself**

SQL

in **One Hour** a Day

SAMS

Sams Teach Yourself SQL in One Hour a Day

ISBN-13: 978-0-672-33025-4
ISBN-10: 0-672-33025-3

Library of Congress Cataloging-in-Publication Data:

Stephens, Ryan K.
Sams teach yourself SQL in one hour a day / Ryan Stephens, Ron Plew, Arie D. Jones. — 1st ed.
p. cm.
Previously published under title: Sams teach yourself SQL in 24 hours.
Includes indexes.
ISBN 978-0-672-33025-4 (pbk.)
1. SQL (Computer program language) I. Plew, Ronald R. II. Jones, Arie. III. Stephens, Ryan K. Sams teach yourself SQL in 24 hours. IV. Title.
QA76.73.S67P554 2009
005.13'3—dc22

2009014482

Printed in the United States of America
First Printing: June 2009

Editor-in-Chief
Mark Taub

Acquisitions Editor
Trina MacDonald

Development Editor
Songlin Qiu

Managing Editor
Kristy Hart

Senior Project Editor
Matthew Purcell

Copy Editor
Seth Kerney

Indexer
Lisa Stumpf

Proofreader
Language Logistics, LLC

Technical Editor
Clayton Rardin
Steven Romero

Publishing Coordinator
Cindy Teeters

Book Designer
Gary Adair

Compositor
Nonie Ratcliff

Introduction

Over the past decade the landscape of information technology has drastically shifted to a data-centric world. More than ever companies are looking for ways in which they can leverage their own data networks to make intelligent business decisions. This includes the ability to gather, store, and report effectively over possibly large sets of data in multiple formats. So the role of database administrators and developers have become strategically important in the proper implementation and care of these systems.

The cornerstone to any database project is the language that will be used in order to interact with the system. Fortunately, a consortium of entities has enacted a standard query language for database environments known as the ANSI SQL standard. This provides a commonality between all database querying languages by following this known standard and allows developers to learn the standard and then work on any given number of database systems with minor adjustments.

This book takes a focused approach on getting the reader the basics of the SQL language in order to allow them to have a solid foundation for future learning. Often in today's business environment, there is very little time to learn new things as our day-to-day functions consume large amounts of our time. By focusing on smaller lesson plans and logically segmenting the sections in a stepping stone fashion, the book allows readers to learn the SQL language at their own pace and within their own schedules.

Who Should Read This Book?

This book is for people who want to learn the fundamentals of Structured Query Language (SQL) quickly. Through the use of countless examples, this book depicts all the major components of SQL, as well as options that are available with various database implementations. You should be able to apply what you learn here to relational databases in a traditional business setting.

How Is This Book Organized?

This book is divided into seven parts, which logically break down the structure of ANSI SQL into easily learnable sections:

- Part I, comprised of the first seven lessons, discusses the basic concepts behind SQL and mainly focuses on the SQL query.

- Part II includes topics on the art of database design, such as creating databases and database objects properly, which is often the foundation of RDBMS application development.
- Part III focuses on data manipulation and using SQL to perform `UPDATE`s, `INSERT`s, and `DELETE`s of data within your database. These will be the staple commands that you will use on a day-to-day basis.
- Part IV is dedicated to database administration, which covers such topics as security, management, and performance, enabling you to maintain the integrity and performance of your database instance.
- Part V focuses on more advanced SQL objects such as triggers and stored procedures. Using these objects will allow you to perform more sophisticated data manipulation techniques that would otherwise be difficult in standard SQL syntax.
- Part VI covers more advanced SQL programming. Advanced SQL programming will allow you to perform more advanced queries and manipulation of the data within your database.
- Part VII presents you with SQL in various database implementations. SQL extensions such as PL/SQL allow you to take advantage of unique attributes within a particular database environment, such as Oracle.
- This book also contains four appendices, which provide you with not only the answers to the exercises in each lesson but also the code examples to create and populate the tables used in the book. Two additional appendices are located at http://www.informit.com/store/products.aspx?isbn=0672330253 under the extras tab.

After studying this book, you should have an excellent understanding of SQL and should know how to apply SQL in the real world.

NOTE

If you are familiar with the basics and history of SQL, we suggest you skim the first lesson and begin in earnest with Lesson 2 "Introducing the Query."

The syntax of SQL is explained and then brought to life in examples using MySQL, which is the closest implementation of the ANSI SQL standard syntax, as well as Oracle Express edition, which demonstrates some of the extensions to ANSI SQL.

Conventions Used in This Book

This book uses the following typeface conventions:

- Menu names are separated from menu options by a comma. For example, File, Open means select the Open option from the File menu.
- New terms appear in italic.
- In some listings, we've included both the input and output (**Input/Output** ▼). For these, all code that you type in (input) appears in **`boldface monospace`**. Output appears in standard `monospace`. The Combination icon indicates that both input and output appear in the code.
- The **Input** ▼ and **Output** ▼ icons also identify the nature of the code.
- Many code-related terms within the text also appear in `monospace`.
- Placeholders in code appear in *`italic monospace`*.
- When a line of code is too long to fit on one line of this book, it is broken at a convenient place and continued to the next line. A code continuation character (➥) precedes the continuation of a line of code. (You should type a line of code that has this character as one long line without breaking it.)
- Paragraphs that begin with the **Analysis** ▼ icon explain the preceding code example.
- The **Syntax** ▼ icon identifies syntax statements.
- Special design features enhance the text material:

NOTE

Notes explain interesting or important points that can help you understand SQL concepts and techniques.

TIP

Tips are little pieces of information that will help you in real-world situations. Tips often offer shortcuts to make a task easier or faster.

CAUTION

Cautions provide information about detrimental performance issues or dangerous errors. Pay careful attention to Cautions.

Using MySQL for Hands-on Exercises

We have chosen to use MySQL for hands-on exercises in this edition. In previous editions, we left it up to the reader to obtain access to any SQL implementation. We decided that it would be better to provide the reader with an open-source SQL database that allowed all readers to start on the same level with the same software. We chose MySQL because it is the most popular open-source database available today, and it is easy to download and use.

Unfortunately, MySQL does have its limitations. There are several features of standard SQL that are not supported by MySQL. We have attempted to distinguish between the exercises that support MySQL and those that do not. Those exercises that do not will mainly focus on using Oracle Enterprise edition, instead. The beauty of SQL is that it is a standard language, although each implementation does have its differences. After using MySQL to understand the basic fundamentals of SQL, you should be able to easily apply the concepts you have learned to any SQL implementation.

About the Book's Source Code

In the appendices, you will find the source code for creating all of the objects used throughout the book. This includes all of the tables and data that is used. Additionally, the source code will be available for download from the publisher's website. This will allow you to simply cut and paste entries into your interface instead of spending the majority of your time typing and enable you to focus more clearly on the material.

LESSON 3
Expressions, Conditions, and Operators

In Lesson 2, "Introducing the Query," you used `SELECT` and `FROM` to manipulate data in interesting (and useful) ways. In this lesson, you learn more about `SELECT` and `FROM`. You will expand the basic query with some new terms, a new clause, and a group of handy items called *operators*. By the end of this lesson, you will

- Know what an expression is and how to use it.
- Know what a condition is and how to use it.
- Be familiar with the basic uses of the `WHERE` clause.
- Be able to use arithmetic, comparison, character, logical, and set operators.
- Have a working knowledge of some miscellaneous operators.

NOTE

We used Oracle and MySQL to generate this lesson's examples. Other implementations of SQL might differ slightly in the way in which commands are entered or output is displayed, but the results are basically the same for all implementations that conform to the ANSI standard.

NOTE

This lesson is one of the longest in the book and also one of the most important as it lays the foundation for most of the other lessons. In this lesson we provide many examples for you to absorb. Do not try to remember every specific example but rather learn the concepts behind them. The lessons to follow will give you plenty of practice in implementing what you will learn.

Working with Query Expressions

The definition of an expression is simple: An *expression* returns a value. Expression types are very broad, covering different data types such as String, Numeric, and Boolean. In fact, pretty much anything following a clause (`SELECT` or `FROM`, for example) is an expression. In the following example, `AMOUNT` is an expression that returns the value contained in the `AMOUNT` column:

Syntax ▼

```
SELECT AMOUNT FROM CHECKS;
```

Of course, the following is also considered a numerical expression. Remember that the key to an expression is that it returns a value.

Syntax ▼

```
SELECT AMOUNT*10 FROM CHECKS;
```

In the following statement, `NAME`, `ADDRESS`, `PHONE`, and `ADDRESSBOOK` are expressions:

Syntax ▼

```
SELECT NAME, ADDRESS, PHONE
FROM ADDRESSBOOK;
```

Now, examine the following `WHERE` clause:

Syntax ▼

```
WHERE NAME = 'BROWN'
```

It contains a condition, `NAME = 'BROWN'`, which is an example of a Boolean expression. `NAME = 'BROWN'` will be either `TRUE` or `FALSE`, depending on the condition `=`.

Placing Conditions on Queries

If you ever want to find a particular item or group of items in your database, you need one or more conditions. Conditions are contained in the `WHERE` clause. In the preceding example, the condition is

Syntax ▼

```
NAME = 'BROWN'
```

To find everyone in your organization who worked more than 100 hours last month, your condition would be

Syntax ▼

```
NUMBEROFHOURS > 100
```

Conditions enable you to make selective queries. In their most common form, conditions comprise a variable, a constant, and a comparison operator. In the first example, the variable is NAME, the constant is 'BROWN', and the comparison operator is =. In the second example, the variable is NUMBEROFHOURS, the constant is 100, and the comparison operator is >. You need to know about two more elements before you can write conditional queries: the WHERE clause and operators.

The syntax of the WHERE clause is

Syntax ▼

```
WHERE <SEARCH CONDITION>
```

SELECT, FROM, and WHERE are the three most frequently used clauses in SQL. WHERE simply causes your queries to be more selective. Without the WHERE clause, the most useful thing you could do with a query is display all records in the selected table(s)—for example,

Input ▼

```
SQL> SELECT * FROM BIKES;
```

lists all rows of data in the table BIKES.

Output ▼

```
NAME            FRAMESIZE COMPOSITION   MILESRIDDEN TYPE
-----------     --------- ------------  ----------- --------
TREK 2300            22.5 CARBON FIBER         3500 RACING
BURLEY                 22 STEEL                2000 TANDEM
GIANT                  19 STEEL                1500 COMMUTER
FUJI                   20 STEEL                 500 TOURING
SPECIALIZED            16 STEEL                 100 MOUNTAIN
CANNONDALE           22.5 ALUMINUM             3000 RACING
6 rows selected.
```

If you wanted a particular bike, you could type

Input ▼

```
SQL> SELECT *
   2 FROM BIKES
   3 WHERE NAME = 'BURLEY';
```

which would yield only one record:

Output ▼

```
NAME            FRAMESIZE COMPOSITION    MILESRIDDEN TYPE
-------------   --------- -----------    ----------- ------
BURLEY                 22 STEEL                 2000 TANDEM
1 rows selected.
```

These simple examples show how you can place a condition on the data that you want to retrieve.

Learning How to Use Operators

Operators are the elements you use inside an expression to articulate how you want specified conditions to retrieve data. Operators fall into six groups: arithmetic, comparison, character, logical, set, and miscellaneous. SQL utilizes three types of operators: arithmetic, comparison, and logical.

Arithmetic Operators

The arithmetic operators are plus (+), minus (-), divide (/), multiply (*), and modulo (%). The first four are self-explanatory. Modulo returns the integer remainder of a division. Here are two examples:

```
5 % 2 = 1
6 % 2 = 0
```

The modulo operator does not work with data types that have decimals, such as Real or Number.

If you place several of these arithmetic operators in an expression without any parentheses, the operators are resolved in this order: multiplication, division, modulo, addition, and subtraction. For example, the expression

```
2*6+9/3
```

equals

```
12 + 3 = 15
```

However, the expression

```
2 * (6 + 9) / 3
```

equals

```
2 * 15 / 3 = 10
```

Watch where you put those parentheses! Sometimes the expression does exactly what you tell it to do, rather than what you want it to do. The same holds true for SQL.

The following sections examine the arithmetic operators in some detail and give you a chance to write some queries.

Plus (+)

You can use the plus sign in several ways. Type the following statement to display the `PRICE` table:

Input/Output ▼

```
SQL> SELECT * FROM PRICE;
ITEM            WHOLESALE
--------------- ---------
TOMATOES              .34
POTATOES              .51
BANANAS               .67
TURNIPS               .45
CHEESE                .89
APPLES                .23
6 rows selected.
```

Now type

Input ▼

```
SQL> SELECT ITEM, WHOLESALE, WHOLESALE + 0.15
   2 FROM PRICE;
```

Here the + adds 15 cents to each price to produce the following:

Output ▼

```
ITEM            WHOLESALE WHOLESALE+0.15
--------------- --------- ---------------
TOMATOES              .34             .49
POTATOES              .51             .66
BANANAS               .67             .82
TURNIPS               .45             .60
CHEESE                .89            1.04
APPLES                .23             .38
6 rows selected.
```

Analysis ▼

What is this last column with the unattractive column heading `WHOLESALE+0.15`? It's not in the original table. SQL allows you to create a virtual or derived column by combining or modifying existing columns.

Retype the original entry:

Input ▼

```
SQL> SELECT * FROM PRICE;
```

The following table results:

Output ▼

```
ITEM            WHOLESALE
--------------- ---------
TOMATOES              .34
POTATOES              .51
BANANAS               .67
TURNIPS               .45
CHEESE                .89
APPLES                .23
6 rows selected.
```

Analysis ▼

The output confirms that the original data has not been changed and that the column heading `WHOLESALE+0.15` is not a permanent part of it. In fact, the column heading is so unattractive that you should do something about it.

Type the following:

Input ▼

```
SQL> SELECT ITEM, WHOLESALE, (WHOLESALE + 0.15) RETAIL
   2 FROM PRICE;
```

Here's the result:

Output ▼

```
ITEM            WHOLESALE    RETAIL
--------------- ---------    ------
TOMATOES              .34       .49
POTATOES              .51       .66
BANANAS               .67       .82
TURNIPS               .45       .60
CHEESE                .89      1.04
APPLES                .23       .38
6 rows selected.
```


Analysis ▼

This is wonderful! Not only can you create new output columns, but you can also rename them on the fly. You can rename any of the columns using the syntax `<column_name> <alias>`. (Note the space between the `column_name` and `alias`.)

For example, the query

Input ▼

```
SQL> SELECT ITEM PRODUCE, WHOLESALE, WHOLESALE + 0.25 RETAIL
   2 FROM PRICE;
```

renames the columns as follows:

Output ▼

```
PRODUCE         WHOLESALE    RETAIL
--------------- ---------    -------
TOMATOES              .34       .59
POTATOES              .51       .76
BANANAS               .67       .92
TURNIPS               .45       .70
CHEESE                .89      1.14
APPLES                .23       .48
6 rows in set (0.00 sec)
```

NOTE

Some implementations of SQL use the syntax <column name = alias>. The preceding example would be written as follows:

```
SQL> SELECT ITEM = PRODUCE,
   2 WHOLESALE,
   3 WHOLESALE + 0.25 = RETAIL,
   4 FROM PRICE;
```

Alternatively, the SQL standard allows you to use the AS keyword, which is implemented in many database systems and looks like the following:

```
SQL> SELECT ITEM AS PRODUCE,
   2 WHOLESALE,
    3 WHOLESALE + 0.25 = RETAIL,
    4 FROM PRICE;
```

Check your implementation for the exact syntax.

NOTE

MySQL allows you to present your column alias in mixed case.

You might be wondering what use aliasing is if you are not using command-line SQL. Fair enough. Have you ever wondered how report builders work? Some day, when you are asked to write a report generator, you'll remember this and not spend weeks reinventing what Dr. Codd and IBM have wrought.

In some implementations of SQL, the plus sign does double duty as a character operator. You'll see that side of the plus sign a little later in this lesson.

Minus (-)

Minus also has two uses. First, it can change the sign of a number. You can use the table HILOW to demonstrate this function.

Input/Output ▼

```
SQL> SELECT * FROM HILOW;
STATE           LOWS     HIGHS
-----           ----     -----
CA               -50       120
FL                20       110
LA                15        99
ND               -70       101
NE               -60       100
```

For example, here's a way to manipulate the data:

Input/Output ▼

```
SQL> SELECT STATE, - LOWS, - HIGHS
   2 FROM HILOW;
STATE           LOWS     HIGHS
-----           ----     -----
CA                50      -120
FL               -20      -110
LA               -15       -99
ND                70      -101
NE                60      -100
```

NOTE Notice that the minus sign was reversed on the temperatures.

The second (and obvious) use of the minus sign is to subtract one column from another—for example,

Input/Output ▼

```
SQL> SELECT STATE,
  2  LOWS,
  3  HIGHS,
  4  (-HIGHS - LOWS) DIFFERENCE
  5  FROM HILOW;

STATE           LOWS     HIGHS  DIFFERENCE
-----           ----     -----  ----------
CA               -50       120         170
FL                20       110          90
LA                15        99          84
ND               -70       101         171
NE               -60       100         160
```

If you accidentally use the minus sign on a character field, you get something like this:

Input/Output ▼

```
SQL> SELECT -STATE FROM HILOW;

ERROR:
ORA-01722: invalid number
no rows selected
```

The exact error message varies with implementation. Here is an example using MySQL:

Input/Output ▼

```
mysql> select -state
    -> from hilow;
+--------+
| -state |
+--------+
|      0 |
|      0 |
|      0 |
|      0 |
+--------+
4 rows in set (0.00 sec)
```

MySQL evaluated the `SELECT` statement, but as you can see, the results are rather meaningless.

Divide (/)

The division operator has only the one obvious meaning. Using the table `PRICE`, type the following:

Input/Output ▼

```
SQL> SELECT * FROM PRICE;
ITEM            WHOLESALE
--------        ---------
TOMATOES              .34
POTATOES              .51
BANANAS               .67
TURNIPS               .45
CHEESE                .89
APPLES                .23
6 rows selected.

mysql> select * from price;
+----------+-----------+
| item     | wholesale |
+----------+-----------+
| TOMATOES |      0.34 |
| POTATOES |      0.51 |
| BANANAS  |      0.67 |
| TURNIPS  |      0.45 |
| CHEESE   |      0.89 |
| APPLES   |      0.23 |
+----------+-----------+

6 rows in set (0.26 sec)
```

You can show the effects of a two-for-one sale by typing the next statement:

Input/Output ▼

```
SQL> SELECT ITEM, WHOLESALE, (WHOLESALE/2) SALEPRICE
  2  FROM PRICE;
ITEM            WHOLESALE SALEPRICE
--------------- --------- ---------
TOMATOES              .34       .17
POTATOES              .51      .255
BANANAS               .67      .335
TURNIPS               .45      .225
CHEESE                .89      .445
APPLES                .23      .115
6 rows selected.
```

The same example in MySQL would be:

Input/Output ▼

```
mysql> select ITEM, WHOLESALE, (WHOLESALE/2) Saleprice
    -> from price;
+----------+-----------+-----------+
¦ ITEM     ¦ WHOLESALE ¦ Saleprice ¦
+----------+-----------+-----------+
¦ TOMATOES ¦      0.34 ¦    0.1700 ¦
¦ POTATOES ¦      0.51 ¦    0.2550 ¦
¦ BANANAS  ¦      0.67 ¦    0.3350 ¦
¦ TURNIPS  ¦      0.45 ¦    0.2250 ¦
¦ CHEESE   ¦      0.89 ¦    0.4450 ¦
¦ APPLES   ¦      0.23 ¦    0.1150 ¦
+----------+-----------+-----------+
6 rows in set (0.26 sec)
```

The use of division in the preceding `SELECT` statement is straightforward (except that coming up with half pennies can be tough).

Multiply (*)

The multiplication operator is also straightforward. Again, using the `PRICE` table, type the following:

Input/Output ▼

```
SQL> SELECT * FROM PRICE;
ITEM            WHOLESALE
--------------- ---------
TOMATOES              .34
POTATOES              .51
```

```
BANANAS               .67
TURNIPS               .45
CHEESE                .89
APPLES                .23
6 rows selected.
```

The output from this query reflects an across-the-board 10% discount. The actual data in the table has not changed.

Input/Output ▼

```
SQL> SQL> SELECT ITEM, WHOLESALE, WHOLESALE * 0.9 NEWPRICE
   2 FROM PRICE;

ITEM            WHOLESALE  NEWPRICE
--------------- ---------  --------
TOMATOES              .34      .306
POTATOES              .51      .459
BANANAS               .67      .603
TURNIPS               .45      .405
CHEESE                .89      .801
APPLES                .23      .207
6 rows selected.
```

The same example in MySQL would be:

Input/Output ▼

```
mysql> select Item,
    -> Wholesale, Wholesale * 0.9 "New Price"
    -> from price;
+----------+-----------+-----------+
¦ Item     ¦ Wholesale ¦ New Price ¦
+----------+-----------+-----------+
¦ TOMATOES ¦      0.34 ¦      0.31 ¦
¦ POTATOES ¦      0.51 ¦      0.46 ¦
¦ BANANAS  ¦      0.67 ¦      0.60 ¦
¦ TURNIPS  ¦      0.45 ¦      0.41 ¦
¦ CHEESE   ¦      0.89 ¦      0.80 ¦
¦ APPLES   ¦      0.23 ¦      0.21 ¦
+----------+-----------+-----------+
6 rows in set (0.00 sec)
```

NOTE

One last thing about aliases: You can give your column a two-word heading by using quotes to surround your aliases. Sometimes this will be single quotes and sometimes it will be double quotes. Please check your specific implementation's documentation to see what it allows.

These operators enable you to perform powerful calculations in a SELECT statement.

Modulo (%)

The modulo operator returns the integer remainder of the division operation. Using the table REMAINS, type the following:

Input/Output ▼

```
SQL> SELECT * FROM REMAINS;
NUMERATOR  DENOMINATOR
---------  -----------
       10           5
        8           3
       23           9
       40          17
     1024          16
       85          34
6 rows selected.
```


The same example in MySQL would be:

Input/Output ▼

```
mysql> select * from remains;
+-----------+-------------+
| numerator | denominator |
+-----------+-------------+
|        10 |           5 |
|         8 |           3 |
|        23 |           9 |
|        40 |          17 |
|      1024 |          16 |
|        85 |          34 |
+-----------+-------------+
6 rows in set (0.43 sec)
```

You can also create a new output column, REMAINDER, to hold the values of NUMERATOR % DENOMINATOR:

Input/Output ▼

```
SQL> SELECT NUMERATOR,
   2 DENOMINATOR,
   3 NUMERATOR%DENOMINATOR REMAINDER
   4 FROM REMAINS;

NUMERATOR DENOMINATOR REMAINDER
```

```
--------- ---------- ---------
       10          5         0
        8          3         2
       23          9         5
       40         17         6
     1024         16         0
       85         34        17
6 rows selected.
```

The same example in MySQL would be:

Input/Output ▼

```
mysql> select numerator, denominator, numerator%denominator remainder
    -> from remains;
+-----------+-------------+----------------------+
| numerator | denominator | remainder            |
+-----------+-------------+----------------------+
|        10 |           5 |                    0 |
|         8 |           3 |                    2 |
|        23 |           9 |                    5 |
|        40 |          17 |                    6 |
|      1024 |          16 |                    0 |
|        85 |          34 |                   17 |
+-----------+-------------+----------------------+
6 rows in set (0.01 sec)
```

Analysis ▼

Some implementations of SQL implement modulo as a function called `MOD` (see Lesson 7, "Molding Data with Built-in Functions"). The following statement produces results that are identical to the results in the preceding statement:

Input/Output ▼

```
SQL> SELECT NUMERATOR,
   2 DENOMINATOR,
   3 MOD(NUMERATOR,DENOMINATOR) REMAINDER
   4 FROM REMAINS;

NUMERATOR DENOMINATOR REMAINDER
--------- ---------- ---------
       10          5         0
        8          3         2
       23          9         5
       40         17         6
     1024         16         0
       85         34        17
6 rows selected.
```

The same example in MySQL would be:

Input/Output ▼

```
mysql> select numerator, denominator,
    -> mod(numerator,denominator) remainder
    -> from remains;
+-----------+-------------+---------------------------+
| numerator | denominator | remainder                 |
+-----------+-------------+---------------------------+
|        10 |           5 |                         0 |
|         8 |           3 |                         2 |
|        23 |           9 |                         5 |
|        40 |          17 |                         6 |
|      1024 |          16 |                         0 |
|        85 |          34 |                        17 |
+-----------+-------------+---------------------------+
6 rows in set (0.00 sec)
```


Precedence

Precedence is the order in which an implementation will evaluate different operators in the same expression. This section examines the use of precedence in a `SELECT` statement. Using the table `PRECEDENCE`, type the following:

Input/Output ▼

```
SQL> SELECT * FROM PRECEDENCE;
       N1         N2         N3         N4
  -------  ---------  ---------  ---------
        1          2          3          4
       13         24         35         46
        9          3         23          5
       63          2         45          3
        7          2          1          4
5 rows selected.

mysql> select * from precedence;
+----+----+----+----+
| n1 | n2 | n3 | n4 |
+----+----+----+----+
|  1 |  2 |  3 |  4 |
| 13 | 24 | 35 | 46 |
|  9 |  3 | 23 |  5 |
| 63 |  2 | 45 |  3 |
|  7 |  2 |  1 |  4 |
+----+----+----+----+
5 rows in set (0.00 sec)
```

Use the following code segment to test precedence:

Input/Output ▼

```
SQL> SELECT
   2 N1+N2*N3/N4,
   3 (N1+N2)*N3/N4,
   4 N1+(N2*N3)/N4
   5 FROM PRECEDENCE;
N1+N2*N3/N4 (N1+N2)*N3/N4 N1+(N2*N3)/N4
----------- ------------ -------------
        2.5          2.25           2.5
   31.26087     28.152174      31.26087
       22.8          55.2          22.8
         93           975            93
        7.5          2.25           7.5

5 rows selected.
mysql> select n1+n2*n3/n4,
    -> (n1+n2)*n3/n4,
    -> n1+(n2*n3)/n4
    -> from precedence;
+-------------+-------------+---------------+
| n1+n2*n3/n4 |(n1+n2)*n3/n4| n1+(n2*n3)/n4 |
+-------------+-------------+---------------+
|        2.50 |           9 |          2.50 |
|       31.26 |        1295 |         31.26 |
|       22.80 |         276 |         22.80 |
|       93.00 |        2925 |         93.00 |
|        7.50 |           9 |          7.50 |
+-------------+-------------+---------------+
5 rows in set (0.00 sec)
```

Notice that the first and last columns are identical. If you added a fourth column `N1+N2*(N3/N4)`, its values would also be identical to those of the current first and last columns. The rules for precedence follow the usual algebraic set in that values are normally executed in the following order moving left to right.

1. Parentheses
2. Multiplication/division
3. Addition/subtraction

Analysis ▼

Quite simply, values inside parentheses are computed first, then multiplication or division operations are performed, and lastly addition and subtraction operations are

performed. These rules are important to remember as you start to write more complicated calculations to analyze data.

Comparison Operators

True to their name, comparison operators compare expressions and return one of three values: `TRUE`, `FALSE`, or `UNKNOWN`. Wait a minute! Unknown? `TRUE` and `FALSE` are self-explanatory, but what is `UNKNOWN`?

To understand how you could get an `UNKNOWN`, you need to know a little about the concept of `NULL`. In database terms, `NULL` is the absence of data in a field. It does not mean that a column has a zero or a blank in it. A zero or a blank is a value. `NULL` means nothing is in that field.

If you make a comparison such as `Field = 9` and the only acceptable value for `Field` is `NULL`, the comparison will come back `UNKNOWN`. Because `UNKNOWN` is an uncomfortable condition, most flavors of SQL change `UNKNOWN` to `FALSE` and provide a special operator, `IS NULL`, to test for a `NULL` condition.

Here's an example of `NULL`: Suppose an entry in the `PRICE` table does not contain a value for `WHOLESALE`. The results of a query might look like this:

Input/Output ▼

```
SQL> SELECT * FROM PRICE;

ITEM            WHOLESALE
---------       ---------
TOMATOES              .34
POTATOES              .51
BANANAS               .67
TURNIPS               .45
CHEESE                .89
APPLES                .23
ORANGES

7 rows selected.
```

Analysis ▼

Notice that no value appears in the `WHOLESALE` field position for `ORANGES`. The value of the field `WHOLESALE` for `ORANGES` is `NULL`. The `NULL` is noticeable in this case because it is in a numeric column. However, if the `NULL` appeared in the `ITEM` column, it would be impossible to tell the difference between `NULL` and a blank.

Try to find the NULL:

Input/Output ▼

```
SQL> SELECT *
   2 FROM PRICE
   3 WHERE WHOLESALE IS NULL;
ITEM            WHOLESALE
--------        ---------
ORANGES

1 rows selected.
```

As you can see by the output, ORANGES is the only item whose value for WHOLESALE is NULL, or does not contain a value. What if you use the equal sign (=) instead?

Input/Output ▼

```
SQL>SELECT *
   2 FROM PRICE
   3 WHERE WHOLESALE = NULL;

no rows selected
```

Analysis ▼

You wouldn't find anything because the comparison WHOLESALE = NULL returned a FALSE—the result was unknown. It would be more appropriate to use an IS NULL instead of =, changing the WHERE statement to WHERE WHOLESALE IS NULL. In this case, you would get all the rows where a NULL existed.

This example also illustrates both the use of the most common comparison operator (=) and the playground of all comparison operators, the WHERE clause. You already know about the WHERE clause, so here's a brief look at the equal sign.

Equal Sign (=)

Earlier today you saw how some implementations of SQL use the equal sign in the SELECT clause to assign an alias. In the WHERE clause, the equal sign is the most commonly used comparison operator. Used alone, the equal sign is a very convenient way of selecting one value out of many. Try this:

Input/Output ▼

```
SQL> SELECT * FROM FRIENDS;
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- --------
BUNDY           AL                    100 555-1111 IL 22333
```

```
MEZA            AL                    200 555-2222 UK
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
BULHER          FERRIS                345 555-3223 IL 23332
5 rows selected.
```

Let's find JD's row. (On a short list this task appears trivial, but you might have more friends than we do—or you might have a list with thousands of records.)

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE FIRSTNAME = 'JD';

LASTNAME        FIRSTNAME        AREACODE     PHONE         ST        ZIP
--------        ---------          ------   --------      ----       ----
MAST            JD                   381   555-6767        LA       23456
1 rows selected.
mysql> select * from friends
    -> where firstname = 'JD';
+----------+-----------+----------+----------+----+-------+
¦ lastname ¦ firstname ¦ areacode ¦ phone    ¦ st ¦ zip   ¦
+----------+-----------+----------+----------+----+-------+
¦ MAST     ¦ JD        ¦      381 ¦ 555-6767 ¦ LA ¦ 23456 ¦
+----------+-----------+----------+----------+----+-------+
1 row in set (0.37 sec)
```

We got the result that we expected. Try this:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE FIRSTNAME = 'AL';
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
BUNDY           AL                    100 555-1111 IL 22333
MEZA            AL                    200 555-2222 UK
2 rows selected.
```

NOTE

Here you see that = can pull in multiple records. Notice that ZIP is blank on the second record. ZIP is a character field (you learn how to create and populate tables in Lesson 9, "Creating and Maintaining Tables"), and in this particular record, the NULL demonstrates that a NULL in a character field is impossible to differentiate from a blank field.

Here's another very important lesson concerning case sensitivity:

Input/Output ▼

```
SQL>  SELECT FIRSTNAME FROM FRIENDS
   2 WHERE FIRSTNAME = 'BUD';

FIRSTNAME
---------------
BUD
1 row selected.
mysql> select firstname from friends where firstname = 'BUD';
+-----------+
¦ firstname ¦
+-----------+
¦ BUD       ¦
+-----------+
1 row in set (0.00 sec)
```

Now try this:

Input/Output ▼

```
SQL> select FIRSTNAME from friends
   2 where firstname = 'Bud';

no rows selected.
mysql> select firstname
    -> from friends
    -> where firstname = 'bud';
+-----------+
¦ firstname ¦
+-----------+
¦ BUD       ¦
+-----------+
1 row in set (0.01 sec)
```

Analysis ▼

Even though SQL syntax is not case sensitive, data within it is, at least in some implementations. As you can see in the preceding examples, data stored in an Oracle database (SQL*Plus) is case sensitive, whereas the MySQL example demonstrates the opposite.

Most companies prefer to store data in uppercase to provide data consistency. I recommend that you always store data either in all uppercase or in all lowercase, regardless of what type of database you are working in. Mixing case might create difficulties when you try to retrieve accurate data through comparisons in the `WHERE` clause.

Greater Than (>) and Greater Than or Equal To (>=)

The greater than operator (>) works like this:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE AREACODE > 300;
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- ------
MAST            JD                    381 555-6767 LA 23456
BULHER          FERRIS                345 555-3223 IL 23332

2 rows selected.
```

This example found all the area codes greater than (but not including) 300. To include 300, type this:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE AREACODE >= 300;
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
BULHER          FERRIS                345 555-3223 IL 23332
3 rows selected.

mysql> select * from friends
    -> where areacode >= 300;
+----------+-----------+----------+----------+----+-------+
| lastname | firstname | areacode | phone    | st | zip   |
+----------+-----------+----------+----------+----+-------+
| MERRICK  | BUD       |      300 | 555-6666 | CO | 80212 |
| MAST     | JD        |      381 | 555-6767 | LA | 23456 |
| BULHER   | FERRIS    |      345 | 555-3223 | IL | 23332 |
+----------+-----------+----------+----------+----+-------+
3 rows in set (0.34 sec)
```

With this change you get area codes starting at 300 and going up. You could achieve the same results with the statement `AREACODE > 299`.

NOTE

Notice that no quotes surround `300` in either of the two prior SQL statements. Number-defined fields do not require quotes.

Less Than (<) and Less Than or Equal To (<=)

As you might expect, these comparison operators work the same way as > and >= work, only in reverse:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE ST < 'LA';
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
BUNDY           AL                    100 555-1111 IL 22333
MERRICK         BUD                   300 555-6666 CO 80212
BULHER          FERRIS                345 555-3223 IL 23332
3 rows selected.

mysql> select * from friends where st < 'LA';
+----------+-----------+----------+----------+----+-------+
¦ lastname ¦ firstname ¦ areacode ¦ phone    ¦ st ¦ zip   ¦
+----------+-----------+----------+----------+----+-------+
¦ BUNDY    ¦ AL        ¦      100 ¦ 555-1111 ¦ IL ¦ 22333 ¦
¦ MERRICK  ¦ BUD       ¦      300 ¦ 555-6666 ¦ CO ¦ 80212 ¦
¦ BULHER   ¦ FERRIS    ¦      345 ¦ 555-3223 ¦ IL ¦ 23332 ¦
+----------+-----------+----------+----------+----+-------+
3 rows in set (0.00 sec)
```

NOTE

In an Oracle database, if the column has only two characters, the column name is shortened to two characters in the returned rows. If the column name had been `COWS`, it would come out `CO`. The widths of `AREACODE` and `PHONE` are wider than their column names, so they are not truncated.

Analysis ▼

Wait a minute. Did you just use < on a character field? Of course you did. You can use any of these operators on any data type. The result varies by data type. For example, use lowercase in the following state search:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE STATE < 'la';
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
BUNDY           AL                    100 555-1111 IL 22333
MEZA            AL                    200 555-2222 UK
```

```
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
BULHER          FERRIS                345 555-3223 IL 23332
5 rows selected.

mysql> select * from friends where st < 'la';
+----------+-----------+----------+----------+----+-------+
¦ lastname ¦ firstname ¦ areacode ¦ phone    ¦ st ¦ zip   ¦
+----------+-----------+----------+----------+----+-------+
¦ BUNDY    ¦ AL        ¦      100 ¦ 555-1111 ¦ IL ¦ 22333 ¦
¦ MERRICK  ¦ BUD       ¦      300 ¦ 555-6666 ¦ CO ¦ 80212 ¦
¦ BULHER   ¦ FERRIS    ¦      345 ¦ 555-3223 ¦ IL ¦ 23332 ¦
+----------+-----------+----------+----------+----+-------+
3 rows in set (0.00 sec)
```

Uppercase is usually sorted before lowercase; therefore, the uppercase codes returned are less than `la`. Again, to be safe, check your implementation.

TIP

To be sure of how these operators will behave, check your language tables. Most PC implementations use the ASCII tables.

To include the state of Louisiana in the original search, type

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE STATE <= 'LA';
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
BUNDY           AL                    100 555-1111 IL 22333
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
BULHER          FERRIS                345 555-3223 IL 23332
4 rows selected.

mysql> select * from friends where st <= 'LA';
+----------+-----------+----------+----------+----+-------+
¦ lastname ¦ firstname ¦ areacode ¦ phone    ¦ st ¦ zip   ¦
+----------+-----------+----------+----------+----+-------+
¦ BUNDY    ¦ AL        ¦      100 ¦ 555-1111 ¦ IL ¦ 22333 ¦
¦ MERRICK  ¦ BUD       ¦      300 ¦ 555-6666 ¦ CO ¦ 80212 ¦
¦ MAST     ¦ JD        ¦      381 ¦ 555-6767 ¦ LA ¦ 23456 ¦
¦ BULHER   ¦ FERRIS    ¦      345 ¦ 555-3223 ¦ IL ¦ 23332 ¦
+----------+-----------+----------+----------+----+-------+
4 rows in set (0.00 sec)
```

Inequalities (< > or !=)

When you need to find everything except for certain data, use the inequality symbol, which can be either < > or !=, depending on your SQL implementation. For example, to find everyone who is not AL, type this:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE FIRSTNAME <> 'AL';
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
BULHER          FERRIS                345 555-3223 IL 23332
3 rows selected.

mysql> select * from friends where firstname <> 'AL';
+----------+-----------+----------+----------+----+-------+
¦ lastname ¦ firstname ¦ areacode ¦ phone    ¦ st ¦ zip   ¦
+----------+-----------+----------+----------+----+-------+
¦ MERRICK  ¦ BUD       ¦      300 ¦ 555-6666 ¦ CO ¦ 80212 ¦
¦ MAST     ¦ JD        ¦      381 ¦ 555-6767 ¦ LA ¦ 23456 ¦
¦ BULHER   ¦ FERRIS    ¦      345 ¦ 555-3223 ¦ IL ¦ 23332 ¦
+----------+-----------+----------+----------+----+-------+
4 rows in set (0.00 sec)
```

To find everyone not living in California, type this:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE STATE != 'CA';
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
BUNDY           AL                    100 555-1111 IL 22333
MEZA            AL                    200 555-2222 UK
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
BULHER          FERRIS                345 555-3223 IL 23332
5 rows selected.

mysql> select * from friends where st != 'CA';
+----------+-----------+----------+----------+----+-------+
¦ lastname ¦ firstname ¦ areacode ¦ phone    ¦ st ¦ zip   ¦
+----------+-----------+----------+----------+----+-------+
¦ BUNDY    ¦ AL        ¦      100 ¦ 555-1111 ¦ IL ¦ 22333 ¦
¦ MEZA     ¦ AL        ¦      200 ¦ 555-2222 ¦ UK¦        ¦
```

```
¦ MERRICK  ¦ BUD       ¦      300 ¦ 555-6666 ¦ CO ¦ 80212 ¦
¦ MAST     ¦ JD        ¦      381 ¦ 555-6767 ¦ LA ¦ 23456 ¦
¦ BULHER   ¦ FERRIS    ¦      345 ¦ 555-3223 ¦ IL ¦ 23332 ¦
+----------+-----------+----------+----------+----+-------+
5 rows in set (0.00 sec)
```

NOTE

Notice that both symbols, <> and !=, can express "not equal" in the two implementations we have shown you.

Character Operators

You can use character operators to manipulate the way character strings are represented, both in the output of data and in the process of placing conditions on data to be retrieved. This section describes two character operators: the LIKE operator and the ¦¦ operator, the latter of which conveys the concept of character concatenation.

LIKE

What if you wanted to select parts of a database that fit a pattern but weren't quite exact matches? You could use the equal sign and run through all the possible cases, but that process would be boring and time-consuming. Instead, you can use LIKE. Consider the following:

Input/Output ▼

```
SQL> SELECT * FROM PARTS;
NAME            LOCATION        PARTNUMBER
-----------     -----------     ----------
APPENDIX        MID-STOMACH              1
ADAMS APPLE     THROAT                   2
HEART           CHEST                    3
SPINE           BACK                     4
ANVIL           EAR                      5
KIDNEY          MID-BACK                 6
6 rows selected.
```

How can you find all the parts located in the back? A quick visual inspection of this simple table shows that it has two parts, but unfortunately the locations have slightly different names. Try this:

Input/Output ▼

```
SQL> SELECT *
   2 FROM PARTS
   3 WHERE LOCATION LIKE '%BACK%';
```

```
NAME            LOCATION        PARTNUMBER
-------         --------        ----------
SPINE           BACK                     4
KIDNEY          MID-BACK                 6
2 rows selected.
```

Analysis ▼

You can see the use of the percent sign (%) in the statement after LIKE. When used inside a LIKE expression, % is a wildcard. What you asked for was any occurrence of BACK in the column location. If you queried

Input ▼

```
SQL> SELECT *
   2 FROM PARTS
   3 WHERE LOCATION LIKE 'BACK%';
```

you would get any occurrence that started with BACK:

Input/Output ▼

```
NAME            LOCATION        PARTNUMBER
-----           --------        ----------
SPINE           BACK                     4

1 rows selected.
mysql> select * from parts where location like 'BACK%';
+-------+----------+------------+
¦ name  ¦ location ¦ partnumber ¦
+-------+----------+------------+
¦ SPINE ¦ BACK     ¦          4 ¦
+-------+----------+------------+
1 row in set (0.00 sec)
```

If you queried

Input ▼

```
SQL> SELECT *
   2 FROM PARTS
    3 WHERE NAME LIKE 'A%';
```

you would get any name that starts with A:

Output ▼

```
NAME            LOCATION        PARTNUMBER
-----------     -----------     ----------
APPENDIX        MID-STOMACH              1
ADAMS APPLE     THROAT                   2
ANVIL           EAR                      5
3 rows selected.
```

Is `LIKE` case sensitive in *both* Oracle and MySQL? Try the next query to find out.

Input/Output ▼

```
SQL> SELECT *
   2 FROM PARTS
   3 WHERE NAME LIKE 'a%';

no rows selected

mysql> select * from parts where name like 'a%';
+-------------+-------------+------------+
¦ name        ¦ location    ¦ partnumber ¦
+-------------+-------------+------------+
¦ APPENDIX    ¦ MID-STOMACH ¦          1 ¦
¦ ADAMS APPLE ¦ THROAT      ¦          2 ¦
¦ ANVIL       ¦ EAR         ¦          5 ¦
+-------------+-------------+------------+
3 rows in set (0.00 sec)
```

The answer is yes in Oracle and no in MySQL. References to data are dependent upon the implementation you are working with.

What if you want to find data that matches all but one character in a certain pattern? In this case you could use a different type of wildcard: the underscore.

Underscore (_)

The underscore is the single-character wildcard. Using a modified version of the table `FRIENDS`, type this:

Input/Output ▼

```
SQL> SELECT * FROM FRIENDS;
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
BUNDY           AL                    100 555-1111 IL 22333
MEZA            AL                    200 555-2222 UK
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
```

```
BULHER          FERRIS                345 555-3223 IL 23332
PERKINS         ALTON                 911 555-3116 CA 95633
BOSS            SIR                   204 555-2345 CT 95633
7 rows selected.
```

To find all the records where `ST` starts with `C`, type the following:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE ST LIKE 'C_';
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
MERRICK         BUD                   300 555-6666 CO 80212
PERKINS         ALTON                 911 555-3116 CA 95633
BOSS            SIR                   204 555-2345 CT 95633
3 rows selected.

mysql> select * from friends where st like 'C_';
+----------+-----------+----------+----------+----+-------+
¦ lastname ¦ firstname ¦ areacode ¦ phone    ¦ st ¦ zip   ¦
+----------+-----------+----------+----------+----+-------+
¦ MERRICK  ¦ BUD       ¦      300 ¦ 555-6666 ¦ CO ¦ 80212 ¦
¦ PERKINS  ¦ ALTON     ¦      911 ¦ 555-3116 ¦ CA ¦ 95633 ¦
¦ BOSS     ¦ SIR       ¦      204 ¦ 555-2345 ¦ CT ¦ 95633 ¦
+----------+-----------+----------+----------+----+-------+
3 row in set (0.00 sec)
```

You can use several underscores in a statement:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE PHONE LIKE'555-6_6_';
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
2 rows selected.
```

The previous statement could also be written as follows:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE PHONE LIKE '555-6%';
```

```
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
2 rows selected.
```

Notice that the results are identical. These two wildcards can be combined. The next example finds all records with `L` as the second character:

Input/Output ▼

```
SQL> SELECT *
  2 FROM FRIENDS
   3 WHERE FIRSTNAME LIKE '_L%';
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
BUNDY           AL                    100 555-1111 IL 22333
MEZA            AL                    200 555-2222 UK
PERKINS         ALTON                 911 555-3116 CA 95633
3 rows selected.
```


Concatenation (||)

The || (double pipe) symbol concatenates two strings. Try this:

Input/Output ▼

```
SQL> SELECT FIRSTNAME || LASTNAME ENTIRENAME
   2 FROM FRIENDS;
ENTIRENAME
----------------------
AL             BUNDY
AL             MEZA
BUD            MERRICK
JD             MAST
FERRIS         BULHER
ALTON          PERKINS
SIR            BOSS
7 rows selected.
```

Analysis ▼

Notice that || is used instead of +. If you use + to try to concatenate the strings, the SQL interpreter used for this example (Oracle) returns the following error:

Input/Output ▼

```
SQL> SELECT FIRSTNAME + LASTNAME ENTIRENAME
   2  FROM FRIENDS;

ERROR:
ORA-01722: invalid number
```

It is looking for two numbers to add and throws the error invalid number when it doesn't find any.

NOTE

Some implementations of SQL, such as Microsoft SQL Server, use the plus sign to concatenate strings. Check your implementation.

NOTE

MySQL can be set up to allow the `||` for concatenation; however, this is not the default when MySQL is installed. `concat()` is the default. Any number of variables may be passed to the function `concat()`, and it is quite easy to use. Should you desire to change the parameters in MySQL to allow the use of the `||` for concatenation, first please research the subject in the documentation provided with MySQL.

Input/Output ▼

```
mysql> select concat(firstname," ",lastname)Entirename from friends;
+---------------+
¦ Entirename    ¦
+---------------+
¦ AL BUNDY      ¦
¦ BUD MERRICK   ¦
¦ JD MAST       ¦
¦ FERRIS BULHER ¦
¦ AL MEZA       ¦
¦ ALTON PERKINS ¦
¦ SIR BOSS      ¦
+---------------+
7 rows in set (0.00 sec)
```

Here's a more practical example using concatenation:

Input/Output ▼

```
SQL> SELECT LASTNAME || ',' || FIRSTNAME NAME
   2  FROM FRIENDS;

NAME
----------------------------------------------
BUNDY    , AL
MEZA     , AL
MERRICK  , BUD
MAST     , JD
BULHER   , FERRIS
PERKINS  , ALTON
BOSS     , SIR
7 rows selected.

mysql> select concat(lastname,","," ",firstname)Name from friends;
+----------------+
¦ Name           ¦
+----------------+
¦ BUNDY, AL      ¦
¦ MEZA, AL       ¦
¦ MERRICK, BUD   ¦
¦ MAST, JD       ¦
¦ BULHER, FERRIS ¦
¦ PERKINS, ALTON ¦
¦ BOSS, SIR      ¦

+----------------+
7 rows in set (0.00 sec)
```

The Oracle statement inserted a comma between the last name and the first name. This was done because Oracle (and other implementations) accounts for the entire length that a column may be when it concatenates to the other string. This creates a natural spacing between the values of the columns/strings. The MySQL statement inserted a comma and a space between the two columns. MySQL automatically runs the values of the columns/strings into one; thus, any "natural" spacing between the values is lost.

NOTE

More on this space issue: Notice the extra spaces between the first name and the last name in the Oracle examples. These spaces are actually part of the data. With certain data types, spaces are right-padded to values less than the total length allocated for a field. See your implementation. Data types will be discussed in Lesson 9. Additionally, if you try to concatenate a NULL value to a string, the result will be a NULL value for the entire expression. In these instances, you would possibly want to use a built-in function to remove the NULL values. This will be discussed in Lesson 7.

So far you have performed the comparisons one at a time. This method is fine for some problems, but what if you need to find all the people at work with last names starting with P who have less than three days of vacation time? Logical operators can help in this case.

Logical Operators

Logical operators separate two or more conditions in the `WHERE` clause of a SQL statement.

Vacation time is always a hot topic around the workplace. Say you designed a table called `VACATION` for the accounting department:

Input/Output ▼

```
SQL> SELECT * FROM VACATION;

LASTNAME        EMPLOYEENUM     YEARS LEAVETAKEN
--------------- -----------     ----- ----------
ABLE                    101         2          4
BAKER                   104         5         23
BLEDSOE                 107         8         45
BOLIVAR                 233         4         80
BOLD                    210        15        100
COSTALES                211        10         78
6 rows selected.
```

Suppose your company gives each employee 12 days of leave each year. Using what you have learned and a logical operator, find all the employees whose name starts with B and who have more than 50 days of leave coming.

Input/Output ▼

```
SQL> SELECT LASTNAME,
   2 YEARS * 12 - LEAVETAKEN REMAINING
   3 FROM VACATION
   4 WHERE LASTNAME LIKE 'B%'
   5 AND
   6 YEARS * 12 - LEAVETAKEN > 50;
LASTNAME        REMAINING
--------        ---------
BLEDSOE                51
BOLD                   80
2 rows selected.


mysql> select lastname,
```

```
    -> years*12 - leavetaken remaining
    -> from vacation
    -> where lastname like 'B%'
    -> and years*12 - leavetaken > 50;
+----------+-----------+
¦ lastname ¦ remaining ¦
+----------+-----------+
¦ BLEDSOE  ¦        51 ¦
¦ BOLD     ¦        80 ¦
+----------+-----------+
2 rows in set (0.00 sec)
```

Analysis ▼

This query is the most complicated you have done to date. The `SELECT` clause (lines 1 and 2) uses arithmetic operators to determine how many days of leave each employee has remaining. The normal precedence computes `YEARS * 12 - LEAVETAKEN`. (A clearer approach would be to write `(YEARS * 12) - LEAVETAKEN.)`

`LIKE` is used in line 4 with the wildcard `%` to find all the `B` names. Line 5 uses the `>` to find all occurrences greater than `50`.

The new element is on line 5. You used the logical operator `AND` to ensure that you found records that met the criteria in lines 4 *and* 5.

AND

`AND` requires that both expressions on either side be true to return `TRUE`. If either expression is false, `AND` returns `FALSE`. For example, to find out which employees have been with the company for 5 or fewer years and have taken more than 20 days leave, try this:

Input/Output ▼

```
SQL> SELECT LASTNAME
   2 FROM VACATION
   3 WHERE YEARS <= 5
   4 AND
   5 LEAVETAKEN > 20 ;
LASTNAME
--------
BAKER
BOLIVAR
2 rows selected.

mysql> select lastname from vacation
    -> where years <= 5
    -> and leavetaken > 20;
```

```
+----------+
¦ lastname ¦
+----------+
¦ BAKER    ¦
¦ BOLIVAR  ¦
+----------+
2 rows in set (0.00 sec)
```

If you want to know which employees have been with the company for 5 years or more and have taken less than 50 percent of their leave, you could write:

Input/Output ▼

```
SQL> SELECT LASTNAME WORKAHOLICS
   2 FROM VACATION
   3 WHERE YEARS >= 5
   4 AND
   5 ((YEARS *12)-LEAVETAKEN)/(YEARS * 12) < 0.50;
WORKAHOLICS
-----------
BOLD
COSTALES

2 rows selected.
```

```
mysql> select lastname Workaholics
    -> from vacation
    -> where years >= 5
    -> and ((years * 12) - leavetaken) / (years * 12) < 0.50;
+-------------+
¦ Workaholics ¦
+-------------+
¦ BOLD        ¦
¦ COSTALES    ¦
+-------------+
2 rows in set (0.00 sec)
```

Check these people for burnout. Also check out how we used the `AND` to combine these two conditions.

OR

You can also use `OR` to sum up a series of conditions. If any of the comparisons are true, `OR` returns `TRUE`. To illustrate the difference, run the last query with `OR` instead of with `AND`:

Input/Output ▼

```
SQL> SELECT LASTNAME WORKAHOLICS
   2 FROM VACATION
   3 WHERE YEARS >= 5
   4 OR
   5 ((YEARS *12)-LEAVETAKEN)/(YEARS * 12) < 0.50;

WORKAHOLICS
-----------
BAKER
BLEDSOE
BOLD
COSTALES

4 rows selected.
```



```
mysql> select lastname
    -> from vacation
    -> where years >= 5
    -> OR ((years*12)-leavetaken)/(years*12) < 0.50;
+----------+
¦ lastname ¦
+----------+
¦ BAKER    ¦
¦ BLEDSOE  ¦
¦ BOLD     ¦
¦ COSTALES ¦
+----------+
4 rows in set (0.00 sec)
```

The original names are still in the list, but you have three new entries (who would probably resent being called workaholics). These three new names made the list because they satisfied one of the conditions. `OR` requires only that one of the conditions be true for data to be returned.

NOT

`NOT` means just that. If the condition it applies to evaluates to `TRUE`, `NOT` makes it `FALSE`. If the condition after the `NOT` is `FALSE`, it becomes `TRUE`. For example, the following `SELECT` returns the only two names not beginning with `B` in the table:

Input/Output ▼

```
SQL> SELECT *
   2 FROM VACATION
   3 WHERE LASTNAME NOT LIKE 'B%';
```

```
LASTNAME        EMPLOYEENUM     YEARS LEAVETAKEN
--------        -----------     ----- ----------
ABLE                    101         2          4
COSTALES                211        10         78
2 rows selected.


mysql> select * from vacation
    -> where lastname not like 'B%';
+----------+-------------+-------+------------+
¦ lastname ¦ employeenum ¦ years ¦ leavetaken ¦
+----------+-------------+-------+------------+
¦ ABLE     ¦         101 ¦     2 ¦          4 ¦
¦ COSTALES ¦         211 ¦    10 ¦         78 ¦
+----------+-------------+-------+------------+
2 rows in set (0.00 sec)
```

`NOT` can also be used with the operator `IS` when applied to `NULL`. Recall the `PRICE` table where we put a `NULL` value in the `WHOLESALE` column opposite the item `ORANGES`.

Input/Output ▼

```
SQL> SELECT * FROM PRICE;

ITEM            WHOLESALE
--------------- ---------
TOMATOES              .34
POTATOES              .51
BANANAS               .67
TURNIPS               .45
CHEESE                .89
APPLES                .23
ORANGES
7 rows selected.
```

To find the non-`NULL` items, type this:

Input/Output ▼

```
SQL> SELECT *
   2 FROM PRICE
   3 WHERE WHOLESALE IS NOT NULL;
ITEM            WHOLESALE
--------------- ---------
TOMATOES              .34
POTATOES              .51
BANANAS               .67
TURNIPS               .45
```

```
CHEESE                .89
APPLES                .23
6 rows selected.
```

Set Operators

In Lesson 1, "Getting Started with SQL," you learned that SQL is based on the theory of sets. The following sections examine set operators. *Set operators* are used to combine different sets of data returned by different queries into one query, and ultimately, one data set. There are various set operators available in SQL that allow you to combine different data sets to meet your data processing needs.

UNION and UNION ALL

UNION returns the results of two queries minus the duplicate rows. The following two tables represent the rosters of teams:

Input/Output ▼

```
SQL> SELECT * FROM FOOTBALL;

NAME
---------------
ABLE
BRAVO
CHARLIE
DECON
EXITOR
FUBAR
GOOBER
7 rows selected.
SQL> SELECT * FROM SOFTBALL;

NAME
-----------------
ABLE
BAKER
CHARLIE
DEAN
EXITOR
FALCONER
GOOBER
7 rows selected.
```

How many different people play on one team or another?

Input/Output ▼

```
SQL> SELECT NAME FROM SOFTBALL
   2 UNION
   3 SELECT NAME FROM FOOTBALL;
NAME
---------------
ABLE
BAKER
BRAVO
CHARLIE
DEAN
DECON
EXITOR
FALCONER
FUBAR
GOOBER
10 rows selected.
```

`UNION` returns 10 distinct names from the two lists. How many names are on both lists (including duplicates)?

Input/Output ▼

```
SQL> SELECT NAME FROM SOFTBALL
   2 UNION ALL
   3 SELECT NAME FROM FOOTBALL;
NAME
---------------
ABLE
BAKER
CHARLIE
DEAN
EXITOR
FALCONER
GOOBER
ABLE
BRAVO
CHARLIE
DECON
EXITOR
FUBAR
GOOBER
14 rows selected.
```

Analysis ▼

The combined list—courtesy of the `UNION ALL` statement—has 14 names. `UNION ALL` works just like `UNION` except that it does not eliminate duplicates. You need to remember

that the UNION and UNION ALL statements will only work if all SELECT statements have the same columns. Otherwise, an error message will be returned. Now show me a list of players who are on both teams. You can't do that with UNION—you need to learn INTERSECT.

INTERSECT

INTERSECT returns only the rows found by both queries. The next SELECT statement shows the list of players who play on both teams:

Input/Output ▼

```
SQL> SELECT * FROM FOOTBALL
   2 INTERSECT
   3 SELECT * FROM SOFTBALL;
NAME
---------------
ABLE
CHARLIE
EXITOR
GOOBER

4 rows selected.
```

In this example, INTERSECT finds the short list of players who are on both teams by combining the results of the two SELECT statements. INTERSECT has the same limitations as the UNION and UNION ALL statement, in as much as the SELECT statements that it is binding must contain the same columns.

MINUS (Difference)

MINUS returns the rows from the first query that were not present in the second. For example:

Input/Output ▼

```
SQL> SELECT * FROM FOOTBALL
   2 MINUS
   3 SELECT * FROM SOFTBALL;
NAME
----------------
BRAVO
DECON
FUBAR

3 rows selected.
```

The preceding query shows the three football players who are not on the softball team. If you reverse the order, you get the three softball players who aren't on the football team:

Input/Output ▼

```
SQL> SELECT * FROM SOFTBALL
   2 MINUS
   3 SELECT * FROM FOOTBALL;
NAME
------------------
BAKER
DEAN
FALCONER

3 rows selected.
```

Miscellaneous Operators: IN and BETWEEN

The two operators IN and BETWEEN provide a shorthand for functions you already know how to do. If you wanted to find friends in Colorado, California, and Louisiana, you could type the following:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE ST= 'CA'
   4 OR
   5 ST ='CO'
   6 OR
   7 ST = 'LA';

LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
PERKINS         ALTON                 911 555-3116 CA 95633

3 rows selected.
```

Or you could type this:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE ST IN('CA','CO','LA');
```

```
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
PERKINS         ALTON                 911 555-3116 CA 95633
3 rows selected.
```

```
mysql> select * from friends
    -> where st in ('CA','CO','LA');
+----------+-----------+----------+----------+----+-------+
| lastname | firstname | areacode | phone    | st | zip   |
+----------+-----------+----------+----------+----+-------+
| MERRICK  | BUD       |      300 | 555-6666 | CO | 80212 |
| MAST     | JD        |      381 | 555-6767 | LA | 23456 |
| PERKINS  | ALTON     |      911 | 555-3116 | CA | 95633 |
+----------+-----------+----------+----------+----+-------+
2 rows in set (0.20 sec)
```


The second example is shorter and more readable than the first. You never know when you might have to go back and work on something you wrote months ago. `IN` also works with numbers. Consider the following, where the column `AREACODE` is a number:

Input/Output ▼

```
SQL> SELECT *
   2 FROM FRIENDS
   3 WHERE AREACODE IN(100,381,204);
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
BUNDY           AL                    100 555-1111 IL 22333
MAST            JD                    381 555-6767 LA 23456
BOSS            SIR                   204 555-2345 CT 95633
3 rows selected.
```

If you needed a range of data from the `PRICE` table, you could write the following:

Input/Output ▼

```
SQL>  SELECT *
  2   FROM PRICE
  3   WHERE WHOLESALE > 0.25
  4   AND
  5   WHOLESALE < 0.75;

ITEM            WHOLESALE
--------        ---------
TOMATOES              .34
```

```
POTATOES              .51
BANANAS               .67
TURNIPS               .45
4 rows selected.
```

Or using `BETWEEN`, you would write this:

Input/Output ▼

```
SQL> SELECT *
   2 FROM PRICE
   3 WHERE WHOLESALE BETWEEN 0.25 AND 0.75;
ITEM            WHOLESALE
--------        ---------
TOMATOES              .34
POTATOES              .51
BANANAS               .67
TURNIPS               .45
4 rows selected.


mysql> select * from price
    -> where wholesale between .25 and .75;
+----------+-----------+
¦ item     ¦ wholesale ¦
+----------+-----------+
¦ TOMATOES ¦      0.34 ¦
¦ POTATOES ¦      0.51 ¦
¦ BANANAS  ¦      0.67 ¦
¦ TURNIPS  ¦      0.45 ¦
+----------+-----------+
4 rows in set (0.08 sec)
```

Again, the second example is a cleaner, more readable solution than the first.

NOTE

If a `WHOLESALE` value of `0.25` existed in the `PRICE` table, that record would have been retrieved also. Parameters used with `BETWEEN` are inclusive.

Summary

At the beginning of this lesson, you knew how to use the basic `SELECT` and `FROM` clauses. Now you know how to use a host of operators that enable you to fine-tune your requests to the database. You learned how to use arithmetic, comparison, character, logical, and

set operators. This powerful set of tools provides the cornerstone of your SQL knowledge. In Lesson 4, you learn to increase the data-mining power of the SQL query by integrating other clauses such as the `WHERE` clause into your queries to perform operations involving grouping and ordering.

Q&A

Q **How does all this information apply to me if I am not using SQL from the command line as depicted in the examples?**

A Whether you use SQL in COBOL as Embedded SQL or in Microsoft's Open Database Connectivity (ODBC), you use the same basic constructions. You will use what you learned in these first lessons repeatedly as you work with SQL.

Q **Why are you constantly telling me to check my implementation? I thought there was a standard!**

A There is an ANSI standard (the most recent version was released in late 2008); however, most vendors modify it somewhat to suit their databases. The basics are similar if not identical, and each instance has extensions that other vendors copy and improve. We have chosen to use ANSI as a starting point and to point out the differences as we go along.

Workshop

The Workshop provides quiz questions to help solidify your understanding of the material covered, as well as exercises to provide you with experience in using what you have learned. Try to answer the quiz and exercise questions before checking the answers in Appendix A, "Answers."

Here are the `CREATE TABLE` statements and `INSERT` statements for the `FRIENDS` and `PRICE` tables. Type the following code into MySQL if you have not already done so.

```
create table friends
(lastname      varchar(15)     not null,
 firstname     varchar(15)     not null,
 areacode      numeric(3)      null,
 phone         varchar(9)      null,
 st            char(2)         not null,
 zip           varchar(5)      not null);

insert into friends values
('BUNDY', 'AL', '100', '555-1111', 'IL', '22333');

insert into friends values
('MEZA', 'AL', '200', '555-2222', 'UK', NULL);
```

```
insert into friends values
('MERRICK', 'BUD', '300', '555-6666', 'CO', '80212');

insert into friends values
('MAST', 'JD', '381', '555-6767', 'LA', '23456');

insert into friends values
('BULHER', 'FERRIS', '345', '555-3223', 'IL', '23332');

insert into friends values
('PERKINS', 'ALTON', '911', '555-3116', 'CA', '95633');

insert into friends values
('BOSS', 'SIR', '204', '555-2345', 'CT', '95633');

create table price
(item          varchar(15)     not null,
 wholesale     decimal(4,2)    not null);

insert into price values
('TOMATOES', '.34');

insert into price values
('POTATOES', '.51');

insert into price values
('BANANAS', '.67');

insert into price values
('TURNIPS', '.45');

insert into price values
('CHEESE', '.89');

insert into price values
('APPLES', '.23');
```

Quiz

Use the FRIENDS table to answer the following questions.

```
LASTNAME        FIRSTNAME        AREACODE PHONE    ST ZIP
--------        ---------        -------- -------- -- -----
BUNDY           AL                    100 555-1111 IL 22333
MEZA            AL                    200 555-2222 UK
MERRICK         BUD                   300 555-6666 CO 80212
MAST            JD                    381 555-6767 LA 23456
BULHER          FERRIS                345 555-3223 IL 23332
PERKINS         ALTON                 911 555-3116 CA 95633
BOSS            SIR                   204 555-2345 CT 95633
```

1. Write a query that returns everyone in the database whose last name begins with M.
2. Write a query that returns everyone who lives in Illinois with a first name of AL.
3. Given two tables (`PART1` and `PART2`) containing columns named `PARTNO`, how would you find out which part numbers are in both tables? Write the query.
4. What shorthand could you use instead of `WHERE a >= 10 AND a <=30`?
5. What will this query return?

```
SELECT FIRSTNAME
FROM FRIENDS
WHERE FIRSTNAME = 'AL'
  AND LASTNAME = 'BULHER';
```

6. What is the main difference in the result set when using `UNION` versus `UNION ALL`?
7. What is the primary difference between using `INTERSECT` and `MINUS`?

Exercises

1. Using the `FRIENDS` table, write a query that returns the following:

```
NAME                ST
-----           -------
AL              FROM IL
```

2. Using the `FRIENDS` table, write a query that returns the following:

```
NAME                        PHONE
--------------              ------------
MERRICK, BUD                300-555-6666
MAST, JD                    381-555-6767
BULHER, FERRIS              345-555-3223
```

3. Select all columns from the `PRICE` table where the column `WHOLESALE` is greater than .50.
4. What results do you get from the following query?

```
mysql> select *
    -> from price
    -> where item like '%ATO%';
```

5. Does MySQL support set operators such as `UNION`, `UNION ALL`, `INTERSECT`, and `MINUS`?
6. What is wrong with the following query?

```
SELECT FIRSTNAME,LASTNAME FROM FRIENDS_1
UNION
SELECT FIRSTNAME  FROM FRIENDS_2;
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

Q

R

S

T

U

V

W

X-Y-Z