TRAINING & REFERENCE

murach's MySQL

3RD EDITION

(Chapter 3)

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • www.murach.com

What developers have said about the previous editions

"If you ever wanted to learn to use MySQL, write SQL queries, create database elements, then this is the book to pick up. Rating: 10 Horseshoes."

Review by Mohamed Sanaulla, JavaRanch.com

"A great first book into SQL: From all the SQL books I looked over, this has by far the best division of chapters and the best order for learning."

Posted at an online bookseller

"As a developer with almost 10 years of MySQL experience, I still picked up a lot of new detail on things I thought I knew. Every development shop that works with MySQL should have a copy of this book."

David Bolton, C/C++/C# Guide, About.com

"This book was the text in my Database Concepts class, and I am so thankful that it was! It provided excellent explanations and examples of database syntax, queries, subqueries, design, etc. I aced the class and decided to take more database classes because of this book."

Posted at an online bookseller

"This book is not just up-to-date with the latest information on MySQL, but it is extremely easy to read and learn from. It only took me a couple weeks to rip through it!"

Posted at an online bookseller

"I was amazed at how much information was packed into this book. I learned a lot of new MySQL ideas, and I will be using it frequently as a reference."

Paul Turpin, Southeastern Inter-Relational Database Users Group

3

How to retrieve data from a single table

In this chapter, you'll learn how to code SELECT statements that retrieve data from a single table. The skills covered here are the essential ones that apply to any SELECT statement you code…no matter how many tables it operates on, no matter how complex the retrieval. So you'll want to be sure you have a good understanding of the material in this chapter before you go on to the chapters that follow.

An introduction to the SELECT statement

To get you started quickly, this chapter begins by presenting the basic syntax of the SELECT statement. Then, it presents several examples that should give you an overview of how this statement works.

The basic syntax of the SELECT statement

Figure 3-1 presents the basic syntax of the SELECT statement. The syntax summary at the top of this figure uses conventions that are similar to those used in other programming manuals. Capitalized words are *keywords* that you have to type exactly as shown. In contrast, you have to provide replacements for the lowercase words. For example, you can enter a list of columns in place of *select_list*, and you can enter a table name in place of *table_source*.

Beyond that, you can omit the clauses enclosed in brackets ([ ]). If you compare the syntax in this figure with the coding examples in the next figure, you should easily see how the two are related.

This syntax summary has been simplified so you can focus on the five main clauses of the SELECT statement: SELECT, FROM, WHERE, ORDER BY, and LIMIT. Most SELECT statements contain the first four of these clauses. However, only the SELECT clause is required.

The SELECT clause is always the first clause in a SELECT statement. It identifies the columns in the result set. These columns are retrieved from the *base tables* named in the FROM clause. Since this chapter focuses on retrieving data from a single table, the examples in this chapter use FROM clauses that name a single base table. In the next chapter, though, you'll learn how to retrieve data from two or more tables.

The WHERE, ORDER BY, and LIMIT clauses are optional. The ORDER BY clause determines how the rows in the result set are sorted, and the WHERE clause determines which rows in the base table are included in the result set. The WHERE clause specifies a *search condition* that's used to *filter* the rows in the base table. When this condition is true, the row is included in the result set.

The LIMIT clause limits the number of rows in the result set. In contrast to the WHERE clause, which uses a search condition, the LIMIT clause simply returns a specified number of rows, regardless of the size of the full result set. Of course, if the result set has fewer rows than are specified by the LIMIT clause, all the rows in the result set are returned.

The basic syntax of the SELECT statement

```
SELECT select_list
[FROM table_source]
[WHERE search_condition]
[ORDER BY order_by_list]
[LIMIT row_limit]
```

The five clauses of the SELECT statement

Clause	Description
`SELECT`	Describes the columns in the result set.
`FROM`	Names the base table from which the query retrieves the data.
`WHERE`	Specifies the conditions that must be met for a row to be included in the result set.
`ORDER BY`	Specifies how to sort the rows in the result set.
`LIMIT`	Specifies the number of rows to return.

Description

- You use the basic SELECT statement shown above to retrieve the columns specified in the SELECT clause from the *base table* specified in the FROM clause and store them in a result set.
- The WHERE clause is used to *filter* the rows in the base table so that only those rows that match the search condition are included in the result set. If you omit the WHERE clause, all of the rows in the base table are included.
- The search condition of a WHERE clause consists of one or more *Boolean expressions* that result in a true, false, or null value. If the combination of all the expressions is a true value, the row being tested is included in the result set. Otherwise, it's not.
- If you include the ORDER BY clause, the rows in the result set are sorted in the specified sequence. Otherwise, the sequence of the rows is not guaranteed by MySQL.
- If you include the LIMIT clause, the result set that's retrieved is limited to a specified number of rows. If you omit this clause, all rows that match are returned.
- You must code the clauses in the order shown or you'll get a syntax error.

Note

- The syntax shown above does not include all of the clauses of the SELECT statement. You'll learn about the other clauses later in this book.

Figure 3-1 The basic syntax of the SELECT statement

SELECT statement examples

Figure 3-2 presents five SELECT statement examples. All of these statements retrieve data from the Invoices table that you experimented with in the last chapter. After each statement, you can see its result set as displayed by MySQL Workbench. In these examples, a horizontal or vertical scroll bar indicates that the result set contains more rows or columns than can be displayed at one time.

The first statement in this figure retrieves all of the rows and columns from the Invoices table. Here, an asterisk (*) is used as a shorthand to indicate that all of the columns should be retrieved, and the WHERE and LIMIT clauses are omitted so all of the rows in the table are retrieved. In addition, this statement doesn't include an ORDER BY clause, so the rows are in primary key sequence.

The second statement retrieves selected columns from the Invoices table. These columns are listed in the SELECT clause. Like the first statement, this statement doesn't include a WHERE or a LIMIT clause, so all the rows are retrieved. Then, the ORDER BY clause causes the rows to be sorted by the invoice_total column in descending order, from largest to smallest.

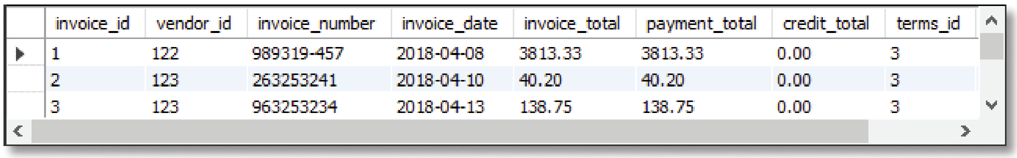
The third statement also lists the columns to be retrieved. In this case, though, the last column is calculated from two columns in the base table, credit_total and payment_total, and the resulting column is given the name total_credits. In addition, the WHERE clause specifies that only the invoice whose invoice_id column has a value of 17 should be retrieved.

The fourth SELECT statement includes a WHERE clause whose condition specifies a range of values. In this case, only invoices with invoice dates between 06/01/2018 and 06/30/2018 are retrieved. In addition, the rows in the result set are sorted by invoice date.

The last statement in this figure shows another example of the WHERE clause. In this case, only those rows with invoice totals greater than 50,000 are retrieved. Since none of the rows in the Invoices table satisfy this condition, the result set is empty.

A SELECT statement that retrieves all the data from the Invoices table

```
SELECT * FROM invoices
```

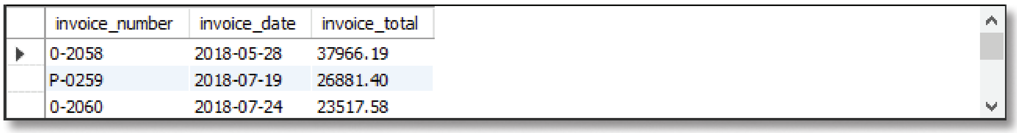


invoice_id	vendor_id	invoice_number	invoice_date	invoice_total	payment_total	credit_total	terms_id
1	122	989319-457	2018-04-08	3813.33	3813.33	0.00	3
2	123	263253241	2018-04-10	40.20	40.20	0.00	3
3	123	963253234	2018-04-13	138.75	138.75	0.00	3

```
(114 rows)
```

A SELECT statement that retrieves three columns from each row, sorted in descending sequence by invoice total

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
ORDER BY invoice_total DESC
```

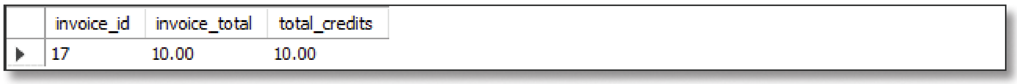


invoice_number	invoice_date	invoice_total
0-2058	2018-05-28	37966.19
P-0259	2018-07-19	26881.40
0-2060	2018-07-24	23517.58

```
(114 rows)
```

A SELECT statement that retrieves two columns and a calculated value for a specific invoice

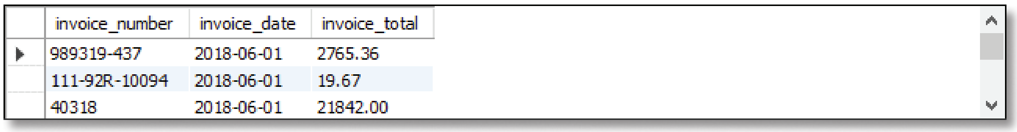
```
SELECT invoice_id, invoice_total,
       credit_total + payment_total AS total_credits
FROM invoices
WHERE invoice_id = 17
```



invoice_id	invoice_total	total_credits
17	10.00	10.00

A SELECT statement that retrieves all invoices between given dates

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE invoice_date BETWEEN '2018-06-01' AND '2018-06-30'
ORDER BY invoice_date
```

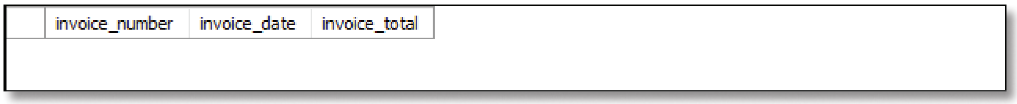


invoice_number	invoice_date	invoice_total
989319-437	2018-06-01	2765.36
111-92R-10094	2018-06-01	19.67
40318	2018-06-01	21842.00

```
(37 rows)
```

A SELECT statement that returns an empty result set

```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE invoice_total > 50000
```



invoice_number	invoice_date	invoice_total

Figure 3-2 SELECT statement examples

How to code the SELECT clause

Now that you have a general idea of how the main clauses of a SELECT statement work, you're ready to learn the details for coding the first clause, the SELECT clause. You can use this clause to specify the columns for a result set.

How to code column specifications

Figure 3-3 begins by presenting a more detailed syntax for the SELECT clause. In this syntax, you can choose between the items in a syntax summary that are separated by pipes (|), and you can omit items enclosed in brackets ([ ]). If you have a choice between two or more optional items, the default item is underlined. And if an element can be coded multiple times in a statement, it's followed by an ellipsis (…).

This figure continues by summarizing four techniques you can use to specify the columns for a result set. First, you can code an asterisk in the SELECT clause to retrieve all of the columns in the base table. When you use this technique, MySQL returns the columns in the order that they are defined in the base table.

Second, you can code a list of column names from the base table separated by commas. In this figure, for instance, the second example specifies three columns that are in the Vendors table.

Third, you can code an *expression* that uses arithmetic operators. The result of an expression is a single value. In this figure, for instance, the third example uses an expression to subtract the payment_total and credit_total columns from the invoice_total column and return the balance due.

Fourth, you can code an expression that uses functions. In this figure, for instance, the fourth example uses the CONCAT function to join a column named first_name, a space, and a column named last_name. Here, two single quotes are used to identify the literal value for the space.

When you code the SELECT clause, you should include only the columns you need. For example, you shouldn't code an asterisk to retrieve all the columns unless you need all the columns. That's because the amount of data that's retrieved can affect system performance. This is particularly important if you're developing SQL statements that will be used by application programs.

For now, don't worry if you don't completely understand all four techniques. In the next four figures, you'll learn more about how they work.

The expanded syntax of the SELECT clause

```
SELECT [ALL|DISTINCT]
       column_specification [[AS] result_column]
    [, column_specification [[AS] result_column]]...
```

Four ways to code column specifications

Source	Option	Syntax
Base table value	All columns	*
	Column name	column_name
Calculated value	Result of a calculation	Arithmetic expressions (see figure 3-5)
	Result of a function	Functions (see figures 3-6 and 3-7)

Column specifications that use base table values

The * is used to retrieve all columns

```
SELECT *
```

Column names are used to retrieve specific columns

```
SELECT vendor_name, vendor_city, vendor_state
```

Column specifications that use calculated values

An arithmetic expression that calculates the balance due

```
SELECT invoice_total - payment_total - credit_total AS balance_due
```

A function that returns the full name

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name
```

Description

- Use SELECT * only when you need to retrieve all of the columns from a table. Otherwise, list the names of the columns you need.
- An *expression* is a combination of column names and operators that evaluate to a single value. In the SELECT clause, you can code expressions that include one or more arithmetic operators and expressions that include one or more functions.
- After each column specification, you can code an AS clause to specify the name for the column in the result set. See figure 3-4 for details.

Note

- The ALL and DISTINCT keywords specify whether or not duplicate rows are returned. See figure 3-9 for details.

Figure 3-3 How to code column specifications

How to name the columns in a result set using aliases

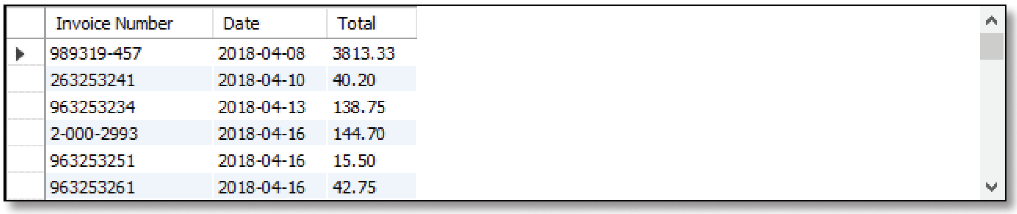
By default, MySQL gives a column in a result set the same name as the column in the base table. If the column is based on a calculated value, it's assigned a name based on the expression for the value. However, whenever you want, you can specify a different name known as a *column alias* as shown in figure 3-4.

To assign a column alias, you code the column specification followed by the AS keyword and the new name as shown by the first example in this figure. Here, the statement creates an alias of "Invoice Number" for the invoice_number column, "Date" for the invoice_date column, and "Total" for the invoice_total column. To include a space in the alias for the first column, this statement encloses that alias in double quotes (").

The second example in this figure shows what happens when you don't assign an alias to a calculated column. In that case, MySQL automatically assigns the column an alias that's the same as the column's expression. Since the expressions for many calculated values are cumbersome, you typically assign a shorter alias for calculated values as shown throughout the rest of this chapter.

A SELECT statement that renames the columns in the result set

```
SELECT invoice_number AS "Invoice Number", invoice_date AS Date,
       invoice_total AS Total
FROM invoices
```

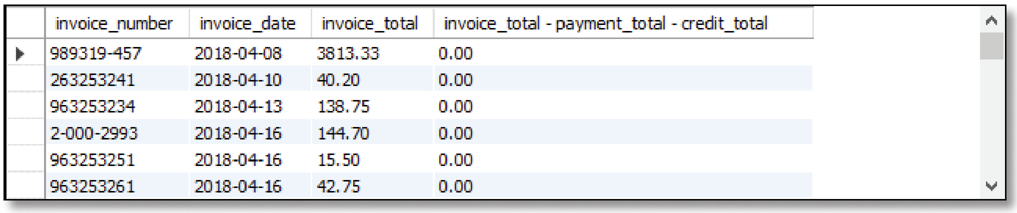


Invoice Number	Date	Total
989319-457	2018-04-08	3813.33
263253241	2018-04-10	40.20
963253234	2018-04-13	138.75
2-000-2993	2018-04-16	144.70
963253251	2018-04-16	15.50
963253261	2018-04-16	42.75

```
(114 rows)
```

A SELECT statement that doesn't name a calculated column

```
SELECT invoice_number, invoice_date, invoice_total,
       invoice_total - payment_total - credit_total
FROM invoices
```



invoice_number	invoice_date	invoice_total	invoice_total - payment_total - credit_total
989319-457	2018-04-08	3813.33	0.00
263253241	2018-04-10	40.20	0.00
963253234	2018-04-13	138.75	0.00
2-000-2993	2018-04-16	144.70	0.00
963253251	2018-04-16	15.50	0.00
963253261	2018-04-16	42.75	0.00

```
(114 rows)
```

Description

- By default, a column in the result set is given the same name as the column in the base table. If that's not what you want, you can specify a substitute name, or *column alias*, for the column.
- To specify an alias for a column, use the AS phrase. Although the AS keyword is optional, I recommend you code it for readability.
- If you don't specify an alias for a column that's based on a calculated value, MySQL uses the expression for the calculated value as the column name.
- To include spaces or special characters in an alias, enclose the alias in double quotes (") or single quotes (').

Figure 3-4 How to name the columns in a result set using aliases

How to code arithmetic expressions

Figure 3-5 shows how to code *arithmetic expressions*. To start, it summarizes the *arithmetic operators* you can use in this type of expression. Then, it presents three examples that show how you use these operators.

The SELECT statement in the first example includes an arithmetic expression that calculates the balance due for an invoice. This expression subtracts the payment_total and credit_total columns from the invoice_total column. The resulting column is given an alias of balance_due.

When MySQL evaluates an arithmetic expression, it performs the operations from left to right based on the *order of precedence*. To start, MySQL performs multiplication, division, and modulo operations. Then, it performs addition and subtraction operations.

If that's not what you want, you can use parentheses to specify how an expression is evaluated. Then, MySQL evaluates the expressions in the innermost sets of parentheses first, followed by the expressions in outer sets of parentheses. Within each set of parentheses, MySQL evaluates the expression from left to right in the order of precedence.

If you want, you can also use parentheses to clarify an expression even if they're not needed for the expression to be evaluated properly. However, you should avoid cluttering your SQL statements with unnecessary parentheses.

To show how parentheses and the order of precedence affect the evaluation of an expression, consider the second example in this figure. Here, the expressions in the second and third columns both perform the same operations. These expressions use one column name (invoice_id) that returns a number and two *literal values* for numbers (7 and 3). When you code a literal value for a number, you don't need to enclose it in quotes.

When MySQL evaluates the expression in the second column, it performs the multiplication operation before the addition operation because multiplication comes before addition in the order of precedence. When MySQL evaluates the expression in the third column, though, it performs the addition operation first because it's enclosed in parentheses. Because of this, these two expressions return different values as shown in the result set.

Although you're probably familiar with the addition, subtraction, multiplication, and division operators, you may not be familiar with the MOD (%) or DIV operators. MOD returns the remainder of a division of two integers, and DIV returns the integer quotient of two numbers. These are shown in the third example in this figure. Here, the second column contains the quotient of the two numbers, which MySQL automatically converts from an integer value to a decimal value. Then, the third column uses the DIV operator to return the integer quotient of the same division operation. The fourth column uses the modulo operator to return the remainder of the division operation.

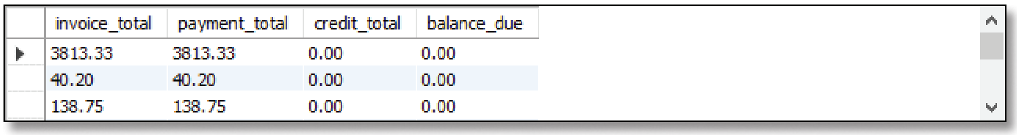
Before going on, you should notice that the second and third SELECT statements include an ORDER BY clause that sorts the result set in ascending sequence by the invoice_id column. Although you might think that this would be the default, that's not the case with MySQL. Instead, the rows in a result set are returned in the most efficient way. If you want the rows returned in a specific sequence, then, you need to include the ORDER BY clause.

The arithmetic operators in order of precedence

Operator	Name	Order of precedence
`*`	Multiplication	1
`/`	Division	1
`DIV`	Integer division	1
`% (MOD)`	Modulo (remainder)	1
`+`	Addition	2
`-`	Subtraction	2

A SELECT statement that calculates the balance due

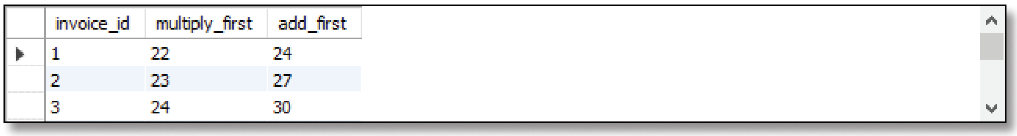
```
SELECT invoice_total, payment_total, credit_total,
       invoice_total - payment_total - credit_total AS balance_due
FROM invoices
```



invoice_total	payment_total	credit_total	balance_due
3813.33	3813.33	0.00	0.00
40.20	40.20	0.00	0.00
138.75	138.75	0.00	0.00

Use parentheses to control the sequence of operations

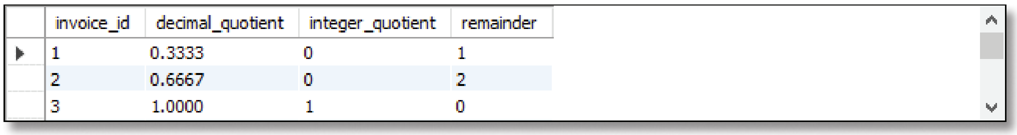
```
SELECT invoice_id,
       invoice_id + 7 * 3 AS multiply_first,
       (invoice_id + 7) * 3 AS add_first
FROM invoices
ORDER BY invoice_id
```



invoice_id	multiply_first	add_first
1	22	24
2	23	27
3	24	30

Use the DIV and modulo operators

```
SELECT invoice_id,
       invoice_id / 3 AS decimal_quotient,
       invoice_id DIV 3 AS integer_quotient,
       invoice_id % 3 AS remainder
FROM invoices
ORDER BY invoice_id
```



invoice_id	decimal_quotient	integer_quotient	remainder
1	0.3333	0	1
2	0.6667	0	2
3	1.0000	1	0

Description

- Unless parentheses are used, the operations in an expression take place from left to right in the *order of precedence*. For arithmetic expressions, MySQL performs multiplication, division, and modulo operations first. Then, it performs addition and subtraction operations.
- When necessary, you can use parentheses to override or clarify the sequence of operations.

Figure 3-5 How to code arithmetic expressions

How to use the CONCAT function to join strings

Figure 3-6 presents the CONCAT function and shows you how to use it to join, or *concatenate*, strings. In MySQL, a *string* can contain any combination of characters, and a *function* performs an operation and returns a value. To code a function, you begin by entering its name followed by a set of parentheses. If the function requires an *argument*, or *parameter*, you enter it within the parentheses. If the function takes more than one argument, you separate them with commas.

In this figure, the first example shows how to use the CONCAT function to join the vendor_city and vendor_state columns in the Vendors table. Since this example doesn't assign an alias to this column, MySQL automatically assigns the expression formula as the column name. In addition, there isn't a space between the vendor_state and the vendor_city in the result set. Since this makes the data difficult to read, this string should be formatted as shown in the second or third example.

The second example shows how to format a string expression by adding spaces and punctuation. Here, the vendor_city column is concatenated with a literal value for a string that contains a comma and a space. Then, the vendor_state column is concatenated with that result, followed by a literal value for a string that contains a single space and the vendor_zip_code column.

To code a string literal, you can enclose the value in either single quotes (') or double quotes ("). Occasionally, you may need to include a single quote as an apostrophe within a literal value for a string. If you're using single quotes around the literal, however, MySQL will misinterpret the apostrophe as the end of the string. To solve this, you can code two single quotation marks in a row as shown by the third example. Or, you can use double quotes like this:

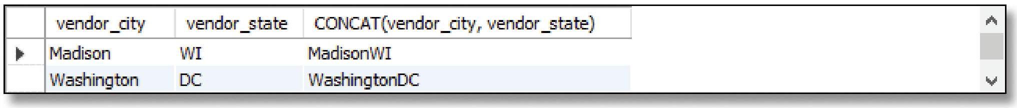
```
CONCAT(vendor_name, "'s Address: ") AS vendor
```

The syntax of the CONCAT function

```
CONCAT(string1[, string2]...)
```

How to concatenate string data

```
SELECT vendor_city, vendor_state, CONCAT(vendor_city, vendor_state)
FROM vendors
```

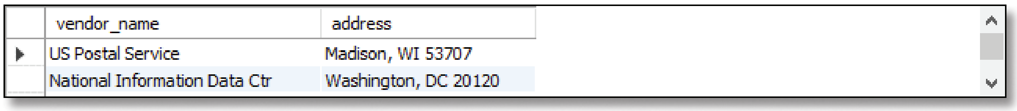


vendor_city	vendor_state	CONCAT(vendor_city, vendor_state)
Madison	WI	MadisonWI
Washington	DC	WashingtonDC

```
(122 rows)
```

How to format string data using literal values

```
SELECT vendor_name,
       CONCAT(vendor_city, ', ', vendor_state, ' ', vendor_zip_code)
           AS address
FROM vendors
```

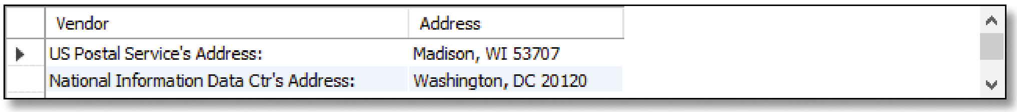


vendor_name	address
US Postal Service	Madison, WI 53707
National Information Data Ctr	Washington, DC 20120

```
(122 rows)
```

How to include apostrophes in literal values

```
SELECT CONCAT(vendor_name, '''s Address: ') AS Vendor,
       CONCAT(vendor_city, ', ', vendor_state, ' ', vendor_zip_code)
           AS Address
FROM vendors
```



Vendor	Address
US Postal Service's Address:	Madison, WI 53707
National Information Data Ctr's Address:	Washington, DC 20120

```
(122 rows)
```

Description

- An expression can include any of the *functions* that are supported by MySQL. A function performs an operation and returns a value.
- To code a function, code the function name followed by a set of parentheses. Within the parentheses, code any *parameters*, or *arguments*, required by the function. If a function requires two or more arguments, separate them with commas.
- To code a literal value for a string, enclose one or more characters within single quotes (') or double quotes (").
- To include a single quote within a literal value for a string, code two single quotes. Or, use double quotes instead of single quotes to start and end the literal value.
- To join, or *concatenate*, two or more string columns or literal values, use the CONCAT function.

Figure 3-6 How to use the CONCAT function to join strings

How to use functions with strings, dates, and numbers

Figure 3-7 shows how to work with three more functions. The LEFT function operates on strings, the DATE_FORMAT function operates on dates, and the ROUND function operates on numbers. For now, don't worry about the details of how the functions shown here work, because you'll learn more about all of these functions in chapter 9. Instead, just focus on how they're used in column specifications.

The first example in this figure shows how to use the LEFT function to extract the first character of the vendor_contact_first_name and vendor_contact_last_name columns. The first parameter of this function specifies the string value, and the second parameter specifies the number of characters to return. Then, this statement concatenates the results of the two LEFT functions to form initials as shown in the result set.

The second example shows how to use the DATE_FORMAT function to change the format used to display date values. This function requires two parameters. The first parameter is the date value to be formatted and the second is a format string that uses specific values as placeholders for the various parts of the date. The first column in this example returns the invoice_date column in the default MySQL date format, "yyyy-mm-dd". Since this format isn't used as often in the USA, the second column is formatted in the more typical "mm/dd/yy" format. In the third column, the invoice date is in another format that's commonly used. In chapter 9, you'll learn more about specifying the format string for the DATE_FORMAT function.

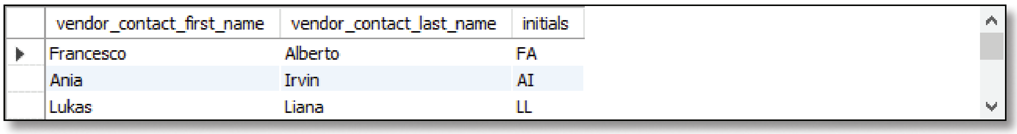
The third example uses the ROUND function to round the value of the invoice_total column to the nearest dollar and nearest dime. This function can accept either one or two parameters. The first parameter specifies the number to be rounded and the optional second parameter specifies the number of decimal places to keep. If the second parameter is omitted, the function rounds to the nearest integer.

The syntax of the LEFT, DATE_FORMAT, and ROUND functions

```
LEFT(string, number_of_characters)
DATE_FORMAT(date, format_string)
ROUND(number[, number_of_decimal_places])
```

A SELECT statement that uses the LEFT function

```
SELECT vendor_contact_first_name, vendor_contact_last_name,
       CONCAT(LEFT(vendor_contact_first_name, 1),
              LEFT(vendor_contact_last_name, 1)) AS initials
FROM vendors
```

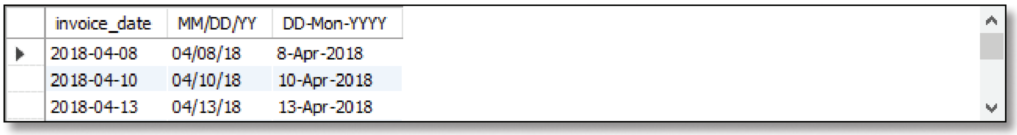


vendor_contact_first_name	vendor_contact_last_name	initials
Francesco	Alberto	FA
Ania	Irvin	AI
Lukas	Liana	LL

```
(122 rows)
```

A SELECT statement that uses the DATE_FORMAT function

```
SELECT invoice_date,
       DATE_FORMAT(invoice_date, '%m/%d/%y') AS 'MM/DD/YY',
       DATE_FORMAT(invoice_date, '%e-%b-%Y') AS 'DD-Mon-YYYY'
FROM invoices
ORDER BY invoice_date
```

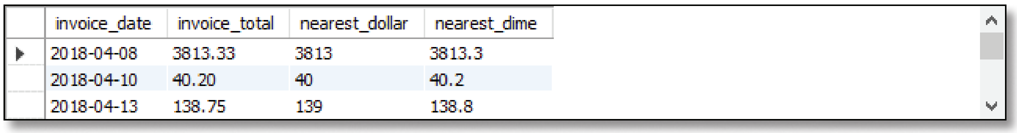


invoice_date	MM/DD/YY	DD-Mon-YYYY
2018-04-08	04/08/18	8-Apr-2018
2018-04-10	04/10/18	10-Apr-2018
2018-04-13	04/13/18	13-Apr-2018

```
(114 rows)
```

A SELECT statement that uses the ROUND function

```
SELECT invoice_date, invoice_total,
       ROUND(invoice_total) AS nearest_dollar,
       ROUND(invoice_total, 1) AS nearest_dime
FROM invoices
ORDER BY invoice_date
```



invoice_date	invoice_total	nearest_dollar	nearest_dime
2018-04-08	3813.33	3813	3813.3
2018-04-10	40.20	40	40.2
2018-04-13	138.75	139	138.8

```
(114 rows)
```

Description

- When using the DATE_FORMAT function to specify the format of a date, you use the percent sign (%) to identify a format code. For example, a format code of *m* returns the month number with a leading zero if necessary. For more information about these codes, see chapter 9.
- For more information about using functions, see chapter 9.

Figure 3-7 How to use functions with strings, dates, and numbers

How to test expressions by coding statements without FROM clauses

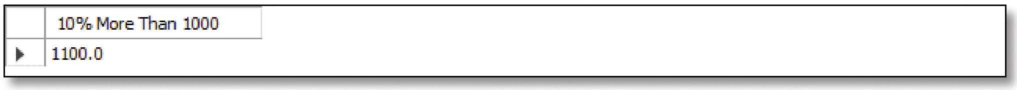
When you use MySQL, you don't have to code FROM clauses in SELECT statements. This makes it easy for you to code SELECT statements that test expressions and functions like those that you've seen in this chapter. Instead of coding column specifications in the SELECT clause, you use literals or functions to supply the test values you need. And you code column aliases to display the results. Then, once you're sure that the code works as you intend it to, you can add the FROM clause and replace the literals or functions with the correct column specifications.

Figure 3-8 shows how to test expressions. Here, the first example tests an arithmetic expression using numeric literals that make it easy to verify the results. The remaining examples test the functions that you saw in figure 3-7. If you compare these statements, you'll see that the second and fourth examples simply replace the column specifications in figure 3-7 with literal values. The third example uses another function, CURRENT_DATE, to supply a date value in place of the invoice_date column that's coded in figure 3-7.

Four SELECT statements without FROM clauses

Example 1: Testing a calculation

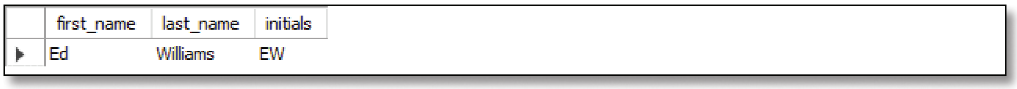
```
SELECT 1000 * (1 + .1) AS "10% More Than 1000"
```



10% More Than 1000
1100.0

Example 2: Testing the CONCAT function

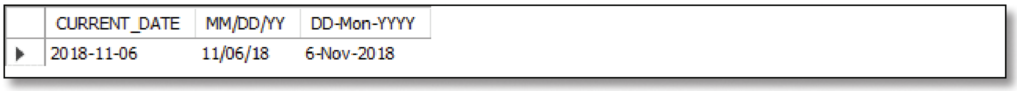
```
SELECT "Ed" AS first_name, "Williams" AS last_name,
    CONCAT(LEFT("Ed", 1), LEFT("Williams", 1)) AS initials
```



first_name	last_name	initials
Ed	Williams	EW

Example 3: Testing the DATE_FORMAT function

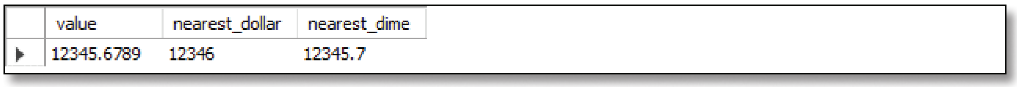
```
SELECT CURRENT_DATE,
       DATE_FORMAT(CURRENT_DATE, '%m/%d/%y') AS 'MM/DD/YY',
       DATE_FORMAT(CURRENT_DATE, '%e-%b-%Y') AS 'DD-Mon-YYYY'
```



CURRENT_DATE	MM/DD/YY	DD-Mon-YYYY
2018-11-06	11/06/18	6-Nov-2018

Example 4: Testing the ROUND function

```
SELECT 12345.6789 AS value,
       ROUND(12345.6789) AS nearest_dollar,
       ROUND(12345.6789, 1) AS nearest_dime
```



value	nearest_dollar	nearest_dime
12345.6789	12346	12345.7

Description

- With MySQL, you don't have to code a FROM clause. This makes it easy to test expressions that include arithmetic operators and functions.
- The CURRENT_DATE function returns the current date. The parentheses are optional for this function.

Figure 3-8 How to test expressions

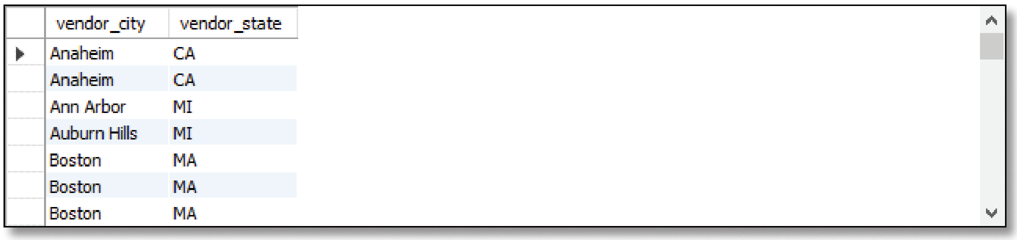
How to eliminate duplicate rows

By default, all of the rows in the base table that satisfy the search condition in the WHERE clause are included in the result set. In some cases, though, that means that the result set will contain duplicate rows, or rows whose column values are identical. If that's not what you want, you can include the DISTINCT keyword in the SELECT clause to eliminate the duplicate rows.

Figure 3-9 shows how this works. Here, both SELECT statements retrieve the vendor_city and vendor_state columns from the Vendors table. The first statement doesn't include the DISTINCT keyword. Because of that, the same city and state can appear in the result set more than once. In the results shown in this figure, for example, you can see that Anaheim CA occurs twice and Boston MA occurs three times. In contrast, the second statement includes the DISTINCT keyword, so each city and state combination is included only once.

A SELECT statement that returns all rows

```
SELECT vendor_city, vendor_state
FROM vendors
ORDER BY vendor_city
```

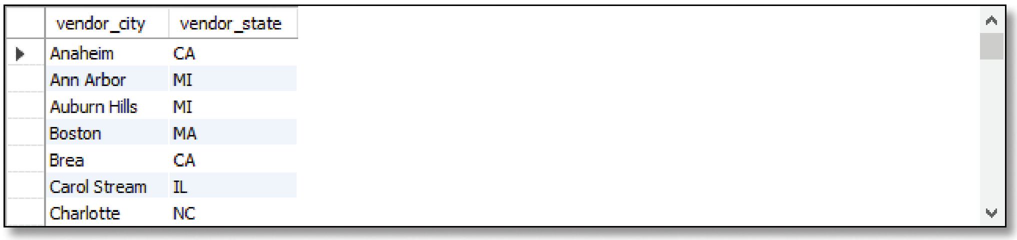


vendor_city	vendor_state
Anaheim	CA
Anaheim	CA
Ann Arbor	MI
Auburn Hills	MI
Boston	MA
Boston	MA
Boston	MA

```
(122 rows)
```

A SELECT statement that eliminates duplicate rows

```
SELECT DISTINCT vendor_city, vendor_state
FROM vendors
ORDER BY vendor_city
```



vendor_city	vendor_state
Anaheim	CA
Ann Arbor	MI
Auburn Hills	MI
Boston	MA
Brea	CA
Carol Stream	IL
Charlotte	NC

```
(53 rows)
```

Description

- The DISTINCT keyword prevents duplicate (identical) rows from being included in the result set. DISTINCTROW is a synonym for DISTINCT.
- The ALL keyword causes all rows matching the search condition to be included in the result set, regardless of whether rows are duplicated. Since this is the default, you'll usually omit the ALL keyword.
- To use the DISTINCT or ALL keyword, code it immediately after the SELECT keyword as shown above.

Figure 3-9   How to eliminate duplicate rows

How to code the WHERE clause

Earlier in this chapter, I mentioned that to improve performance, you should code your SELECT statements so they retrieve only the columns you need. That goes for retrieving rows too: The fewer rows you retrieve, the more efficient the statement will be. Because of that, you typically include a WHERE clause on your SELECT statements with a search condition that filters the rows in the base table so only the rows you need are retrieved. In the topics that follow, you'll learn a variety of ways to code this clause.

How to use the comparison operators

Figure 3-10 shows how to use the *comparison operators* in the search condition of a WHERE clause to compare two expressions. If the result of the comparison is true, the row being tested is included in the query results.

The examples in this figure show how to use the comparison operators. The first WHERE clause, for example, uses the equal operator (=) to retrieve only those rows whose vendor_state column has a value of 'IA'. Here, the state code is a string literal so it must be enclosed in single or double quotes. In contrast, the second WHERE clause uses the greater than (>) operator to retrieve only those rows that have a balance greater than zero. In this case, zero (0) is a numeric literal so it isn't enclosed in quotes.

The third WHERE clause shows another way to retrieve all the invoices with a balance due by rearranging the comparison expression. Like the second clause, it uses the greater than operator. Instead of comparing the balance due to a value of zero, however, it compares the invoice total to the total of the payments and credits that have been applied to the invoice.

The fourth WHERE clause shows how you can use comparison operators other than equal with string data. In this example, the less than operator (<) is used to compare the value of the vendor_name column to a literal string that contains the letter M. That causes the query to return all vendors with names that begin with the letters A through L.

You can also use the comparison operators with date literals, as shown by the fifth and sixth WHERE clauses. The fifth clause retrieves rows with invoice dates on or before July 31, 2018, and the sixth clause retrieves rows with invoice dates on or after July 1, 2018. Like literal values for strings, literal values for dates must be enclosed in single or double quotes. Also, literal values for dates must use this format: YYYY-MM-DD. This is the default date format used by MySQL.

The last two WHERE clauses show how you can test for a not-equal condition. In both cases, only rows with a credit total that isn't equal to zero are retrieved.

Whenever possible, you should compare expressions that have similar data types. If you compare expressions that have different data types, MySQL implicitly converts the data type for you. Generally, this implicit conversion is acceptable. However, implicit conversions can occasionally yield unexpected results. To prevent this, you can explicitly convert the data type by using the CAST or CONVERT functions, which you'll learn about in chapter 8.

The syntax of the WHERE clause with comparison operators

```
WHERE expression_1 operator expression_2
```

The comparison operators

`=`	Equal
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`<>`	Not equal
`!=`	Not equal

Examples of WHERE clauses that retrieve…

Vendors located in Iowa

```
WHERE vendor_state = 'IA'
```

Invoices with a balance due (two variations)

```
WHERE invoice_total - payment_total - credit_total > 0
WHERE invoice_total > payment_total + credit_total
```

Vendors with names from A to L

```
WHERE vendor_name < 'M'
```

Invoices on or before a specified date

```
WHERE invoice_date <= '2018-07-31'
```

Invoices on or after a specified date

```
WHERE invoice_date >= '2018-07-01'
```

Invoices with credits that don't equal zero (two variations)

```
WHERE credit_total <> 0
WHERE credit_total != 0
```

Description

- You can use a *comparison operator* to compare any two expressions. Since MySQL automatically converts the data for comparison, the expressions may be of unlike data types. However, the comparison may sometimes produce unexpected results.
- If the result of a comparison is a true value, the row being tested is included in the result set. If it's a false or null value, the row isn't included.
- To use a string literal or a date literal in a comparison, enclose it in quotes. To use a numeric literal, enter the number without quotes.
- Character comparisons performed on MySQL databases are not case-sensitive. So, for example, 'CA' and 'ca' are considered equivalent.
- If you compare a null value using one of these comparison operators, the result is always a null value. To test for null values, use the IS NULL clause presented in figure 3-15.

Figure 3-10 How to use the comparison operators

How to use the AND, OR, and NOT logical operators

Figure 3-11 shows how to use *logical operators* in a WHERE clause. You can use the AND and OR operators to combine two or more search conditions into a *compound condition*. And you can use the NOT operator to negate a search condition. The examples in this figure show how these operators work.

The first two examples show the AND and OR operators. When you use the AND operator, both conditions must be true. So, in the first example, only those vendors in the state of New Jersey and the city of Springfield are retrieved from the Vendors table. When you use the OR operator, though, only one of the conditions must be true. So, in the second example, all the vendors in the state of New Jersey and all the vendors in the city of Pittsburg (no matter what state) are retrieved.

The third example shows how to use the NOT operator to negate a condition. Here, vendors that are not in the state of California are returned. The fourth example shows a compound condition that uses two NOT operators. This condition is difficult to understand. To make it easier to understand, you can rewrite this condition to remove the NOT operators as shown in the fifth example.

The last two examples in this figure show how the order of precedence for the logical operators and the use of parentheses affect the result of a search condition. By default, the NOT operator is evaluated first, followed by AND, and then by OR. However, you can use parentheses to override the order of precedence or to clarify a logical expression, just as you can with arithmetic expressions.

In the next to last example, for instance, no parentheses are used, so the two conditions connected by the AND operator are evaluated first. In the last example, though, parentheses are used so the two conditions connected by the OR operator are evaluated first. If you take a minute to review the results in this figure, you should quickly see how these two conditions differ.

The syntax of the WHERE clause with logical operators

```
WHERE [NOT] search_condition_1 {AND|OR} [NOT] search_condition_2 ...
```

Examples of WHERE clauses that use logical operators

The AND operator

```
WHERE vendor_state = 'NJ' AND vendor_city = 'Springfield'
```

The OR operator

```
WHERE vendor_state = 'NJ' OR vendor_city = 'Pittsburg'
```

The NOT operator

```
WHERE NOT vendor_state = 'CA'
```

The NOT operator in a complex search condition

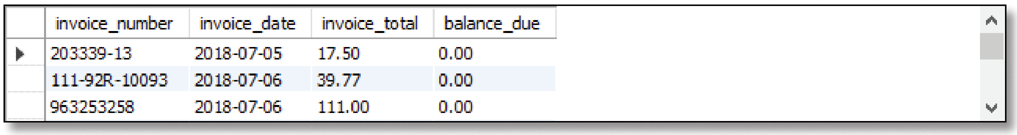
```
WHERE NOT (invoice_total >= 5000 OR NOT invoice_date <= '2018-08-01')
```

The same condition rephrased to eliminate the NOT operator

```
WHERE invoice_total < 5000 AND invoice_date <= '2018-08-01'
```

A compound condition without parentheses

```
WHERE invoice_date > '2018-07-03' OR invoice_total > 500
  AND invoice_total - payment_total - credit_total > 0
```

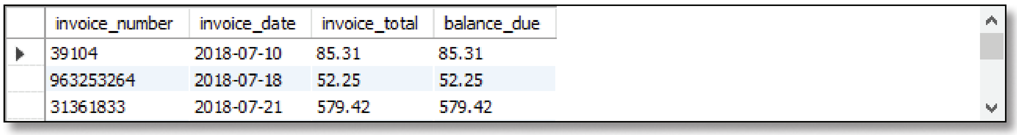


invoice_number	invoice_date	invoice_total	balance_due
203339-13	2018-07-05	17.50	0.00
111-92R-10093	2018-07-06	39.77	0.00
963253258	2018-07-06	111.00	0.00

```
(33 rows)
```

The same compound condition with parentheses

```
WHERE (invoice_date > '2018-07-03' OR invoice_total > 500)
  AND invoice_total - payment_total - credit_total > 0
```



invoice_number	invoice_date	invoice_total	balance_due
39104	2018-07-10	85.31	85.31
963253264	2018-07-18	52.25	52.25
31361833	2018-07-21	579.42	579.42

```
(11 rows)
```

Description

- You can use the AND and OR *logical operators* to create *compound conditions* that consist of two or more conditions. You use the AND operator to specify that the search must satisfy both of the conditions, and you use the OR operator to specify that the search must satisfy at least one of the conditions.
- You can use the NOT operator to negate a condition. Because this can make the search condition unclear, you should rephrase the condition if possible so it doesn't use NOT.
- When MySQL evaluates a compound condition, it evaluates the operators in this sequence: (1) NOT, (2) AND, and (3) OR. You can use parentheses to override this order of precedence or to clarify the sequence in which the operations are evaluated.

Figure 3-11 How to use the AND, OR, and NOT logical operators

How to use the IN operator

Figure 3-12 shows how to code a WHERE clause that uses the IN operator. When you use this operator, the value of the test expression is compared with the list of expressions in the IN phrase. If the test expression is equal to one of the expressions in the list, the row is included in the query results. This is shown by the first example in this figure, which returns all rows whose terms_id column is equal to 1, 3, or 4.

You can also use the NOT operator with the IN phrase to test for a value that's not in a list of expressions. This is shown by the second example. Here, only those vendors that aren't in California, Nevada, or Oregon are retrieved.

At the top of this figure, the syntax of the IN phrase shows that you can code a *subquery* in place of a list of expressions. As you'll learn in chapter 7, subqueries are a powerful feature. For now, though, you should know that a subquery is simply a SELECT statement within another statement.

In the third example, for instance, a subquery is used to return a list of vendor_id values for vendors who have invoices dated July 18, 2018. Then, the WHERE clause retrieves a row only if the vendor_id is in that list. Note that for this to work, the subquery must return a single column, in this case, vendor_id.

The syntax of the WHERE clause with an IN phrase

```
WHERE test_expression [NOT] IN
      ({subquery|expression_1 [, expression_2]...})
```

Examples of the IN phrase

An IN phrase with a list of numeric literals

```
WHERE terms_id IN (1, 3, 4)
```

An IN phrase preceded by NOT

```
WHERE vendor_state NOT IN ('CA', 'NV', 'OR')
```

An IN phrase with a subquery

```
WHERE vendor_id IN
    (SELECT vendor_id
     FROM invoices
     WHERE invoice_date = '2018-07-18')
```

Description

- You can use the IN phrase to test whether an expression is equal to a value in a list of expressions. Each of the expressions in the list is automatically converted to the same type of data as the test expression.
- The list of expressions can be coded in any order without affecting the order of the rows in the result set.
- You can use the NOT operator to test for an expression that's not in the list of expressions.
- You can also compare the test expression to the items in a list returned by a *subquery*. You'll learn more about coding subqueries in chapter 7.

Figure 3-12 How to use the IN operator

How to use the BETWEEN operator

Figure 3-13 shows how to use the BETWEEN operator in a WHERE clause. When you use this operator, the value of a test expression is compared to the range of values specified in the BETWEEN phrase. If the value falls within this range, the row is included in the query results.

The first example in this figure shows a simple WHERE clause that uses the BETWEEN operator. It retrieves invoices with invoice dates between June 1, 2018 and June 30, 2018. Note that the range is inclusive, so invoices with invoice dates of June 1 and June 30 are included in the results.

The second example shows how to use the NOT operator to select rows that aren't within a given range. In this case, vendors with zip codes that aren't between 93600 and 93799 are included in the results.

The third example shows how you can use a calculated value in the test expression. Here, the payment_total and credit_total columns are subtracted from the invoice_total column to give the balance due. Then, this value is compared to the range specified in the BETWEEN phrase.

The last example shows how you can use calculated values in the BETWEEN phrase. Here, the first value is the credit_total column and the second value is the credit_total column plus 500. So the results include all those invoices where the amount paid is between the credit amount and $500 more than the credit amount.

The syntax of the WHERE clause with a BETWEEN phrase

```
WHERE test_expression [NOT] BETWEEN begin_expression AND end_expression
```

Examples of the BETWEEN phrase

A BETWEEN phrase with literal values

```
WHERE invoice_date BETWEEN '2018-06-01' AND '2018-06-30'
```

A BETWEEN phrase preceded by NOT

```
WHERE vendor_zip_code NOT BETWEEN 93600 AND 93799
```

A BETWEEN phrase with a test expression coded as a calculated value

```
WHERE invoice_total - payment_total - credit_total BETWEEN 200 AND 500
```

A BETWEEN phrase with the upper and lower limits coded as calculated values

```
WHERE payment_total BETWEEN credit_total AND credit_total + 500
```

Description

- You can use the BETWEEN phrase to test whether an expression falls within a range of values. The lower limit must be coded as the first expression, and the upper limit must be coded as the second expression. Otherwise, MySQL returns an empty result set.
- The two expressions used in the BETWEEN phrase for the range of values are inclusive. That is, the result set includes values that are equal to the upper or lower limit.
- You can use the NOT operator to test for an expression that's not within the given range.

Figure 3-13 How to use the BETWEEN operator

How to use the LIKE and REGEXP operators

To retrieve rows that match a specific *string pattern*, or *mask*, you can use the LIKE or REGEXP operators as shown in figure 3-14. The LIKE operator is an older operator that lets you search for simple string patterns. When you use this operator, the mask can contain one or both of the *wildcard* symbols shown in the first table in this figure.

In contrast to the LIKE operator, the REGEXP operator allows you to create complex string patterns known as *regular expressions*. To do that, you can use the special characters and constructs shown in the second table in this figure. Although creating regular expressions can be tricky at first, they allow you to search for virtually any string pattern.

In the first example in this figure, the LIKE phrase specifies that all vendors in cities that start with the letters SAN should be included in the query results. Here, the percent sign (%) indicates that any character or characters can follow these three letters. So San Diego and Santa Ana are both included in the results.

The second example selects all vendors whose vendor name starts with the letters COMPU, followed by any one character, the letters ER, and any characters after that. The vendor names Compuserve and Computerworld both match that pattern.

In the third example, the REGEXP phrase searches for the letters SA within the vendor_city column. Since the letters can be in any position within the string, both Pasadena and Santa Ana are included in the results.

The next two examples demonstrate how to use REGEXP to match a pattern to the beginning or end of the string being tested. In the fourth example, the mask ^SA matches the letters SA at the beginning of vendor_city, as in Santa Ana and Sacramento. In contrast, the mask NA$ matches the letters NA at the end of vendor_city, as shown in the fifth example.

The sixth example uses the pipe (|) character to search for either of two string patterns: RS or SN. In this case, the first pattern would match Traverse City and the second would match Fresno, so both are included in the result set.

The last four examples use brackets to specify multiple values. In the seventh example, the vendor_state column is searched for values that contain the letter N followed by either C or V. That excludes NJ and NY. In contrast, the eighth example searches for states that contain the letter N followed by any letter from A to J. This excludes NV and NY.

The ninth example searches the values in the vendor_contact_last_name column for a name that can be spelled two different ways: Damien or Damion. To do that, the mask specifies the two possible characters in the fifth position, E and O, within brackets. In the final example, the REGEXP phrase searches for a vendor_city that ends with any letter, a vowel, and then the letter N.

Both the LIKE and REGEXP operators provide powerful functionality for finding information in a database. However, searches that use these operators sometimes run slowly since they can't use a table's indexes. As a result, you should only use these operators when necessary.

The syntax of the WHERE clause with a LIKE phrase

```
WHERE match_expression [NOT] LIKE pattern
```

The syntax of the WHERE clause with a REGEXP phrase

```
WHERE match_expression [NOT] REGEXP pattern
```

LIKE wildcards

Symbol	Description
`%`	Matches any string of zero or more characters.
`_`	Matches any single character.

REGEXP special characters and constructs

Character/Construct	Description
`^`	Matches the pattern to the beginning of the value being tested.
`$`	Matches the pattern to the end of the value being tested.
`.`	Matches any single character.
`[charlist]`	Matches any single character listed within the brackets.
`[char1-char2]`	Matches any single character within the given range.
|	Separates two string patterns and matches either one.

WHERE clauses that use the LIKE and REGEXP operators

Example	Results that match the mask
`WHERE vendor_city LIKE 'SAN%'`	"San Diego", "Santa Ana"
`WHERE vendor_name LIKE 'COMPU_ER%'`	"Compuserve", "Computerworld"
`WHERE vendor_city REGEXP 'SA'`	"Pasadena", "Santa Ana"
`WHERE vendor_city REGEXP '^SA'`	"Santa Ana", "Sacramento"
`WHERE vendor_city REGEXP 'NA$'`	"Gardena", "Pasadena", "Santa Ana"
`WHERE vendor_city REGEXP 'RS|SN'`	"Traverse City", "Fresno"
`WHERE vendor_state REGEXP 'N[CV]'`	"NC" and "NV" but not "NJ" or "NY"
`WHERE vendor_state REGEXP 'N[A-J]'`	"NC" and "NJ" but not "NV" or "NY"
`WHERE vendor_contact_last_name REGEXP 'DAMI[EO]N'`	"Damien" and "Damion"
`WHERE vendor_city REGEXP '[A-Z][AEIOU]N$'`	"Boston", "Mclean", "Oberlin"

Description

- You use the LIKE and REGEXP operators to retrieve rows that match a *string pattern*, called a *mask*. The mask determines which values in the column satisfy the condition.
- The mask for a LIKE phrase can contain special symbols, called *wildcards*. The mask for a REGEXP phrase can contain special characters and constructs. Masks aren't case-sensitive.
- If you use the NOT keyword, only those rows with values that don't match the string pattern are included in the result set.
- Most LIKE and REGEXP phrases significantly degrade performance compared to other types of searches, so use them only when necessary.

Figure 3-14 How to use the LIKE and REGEXP operators

For the sake of brevity, this chapter only presents the most common symbols that are used in regular expressions. However, MySQL supports most of the symbols that are standard for creating regular expressions. For more information about creating regular expressions, please consult the online MySQL Reference Manual. If you're familiar with using regular expressions in other programming languages such as PHP, you'll find that they work similarly in MySQL.

In addition to the REGEXP operator, MySQL 8.0 provides some new functions that work with regular expressions. You'll learn about these functions in chapter 9 when you learn about some other functions for working with strings.

How to use the IS NULL clause

In chapter 1, you learned that a column can contain a *null value*. A null value is typically used to indicate that a value is not known. A null value is not the same as an empty string (''). An empty string is typically used to indicate that the value is known, and it doesn't exist.

If you're working with a database that allows null values, you need to know how to test for them in search conditions. To do that, you use the IS NULL clause as shown in figure 3-15.

This figure uses a table named Null_Sample to show how to search for null values. This table contains two columns: invoice_id and invoice_total. The values in this table are displayed in the first example.

The second example shows what happens when you retrieve all the rows with invoice_total equal to zero. In this case, the row that has a null value isn't included in the result set. As the third example shows, this row isn't included in the result set when invoice_total isn't equal to zero either. Instead, you have to use the IS NULL clause to retrieve rows with null values, as shown by the fourth example.

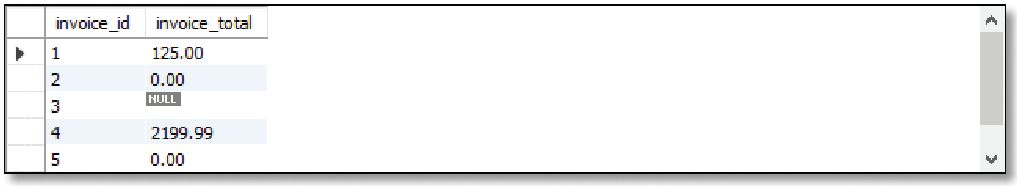
You can also use the NOT operator with the IS NULL clause, as shown by the last example. When you use this operator, all of the rows that don't contain null values are included in the query results.

The syntax of the WHERE clause with the IS NULL clause

```
WHERE expression IS [NOT] NULL
```

The contents of the Null_Sample table

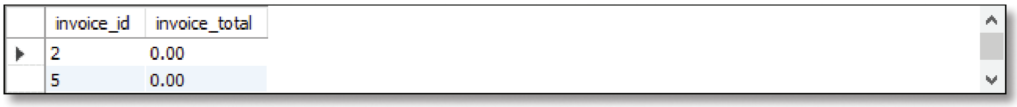
```
SELECT * FROM null_sample
```



invoice_id	invoice_total
1	125.00
2	0.00
3	NULL
4	2199.99
5	0.00

A SELECT statement that retrieves rows with zero values

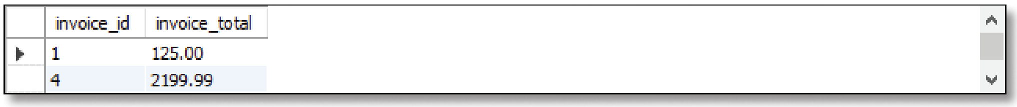
```
SELECT * FROM null_sample
WHERE invoice_total = 0
```



invoice_id	invoice_total
2	0.00
5	0.00

A SELECT statement that retrieves rows with non-zero values

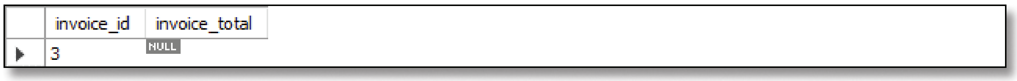
```
SELECT * FROM null_sample
WHERE invoice_total <> 0
```



invoice_id	invoice_total
1	125.00
4	2199.99

A SELECT statement that retrieves rows with null values

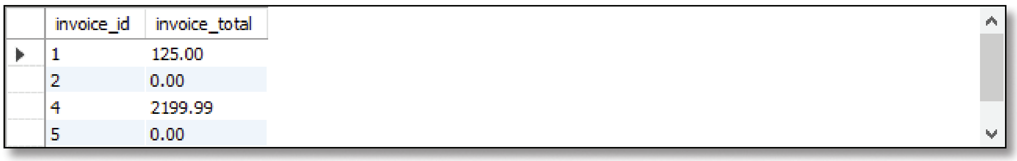
```
SELECT * FROM null_sample
WHERE invoice_total IS NULL
```



invoice_id	invoice_total
3	NULL

A SELECT statement that retrieves rows without null values

```
SELECT *
FROM null_sample
WHERE invoice_total IS NOT NULL
```



invoice_id	invoice_total
1	125.00
2	0.00
4	2199.99
5	0.00

Description

- A *null value* represents a value that's unknown, unavailable, or not applicable. It isn't the same as a zero or an empty string ('').

Figure 3-15 How to use the IS NULL clause

How to code the ORDER BY clause

The ORDER BY clause specifies the sort order for the rows in a result set. In most cases, you'll use column names from the base table to specify the sort order as you saw in some of the examples earlier in this chapter. However, you can also use other techniques to sort the rows in a result set, as described in the topics that follow.

How to sort by a column name

Figure 3-16 presents the expanded syntax of the ORDER BY clause. This syntax shows that you can sort by one or more expressions in either ascending or descending sequence. The three examples in this figure show how to code this clause for expressions that involve column names.

The first two examples show how to sort the rows in a result set by a single column. In the first example, the rows in the Vendors table are sorted in ascending sequence by the vendor_name column. Since ascending is the default sequence, the ASC keyword can be omitted. In the second example, the rows are sorted by the vendor_name column in descending sequence.

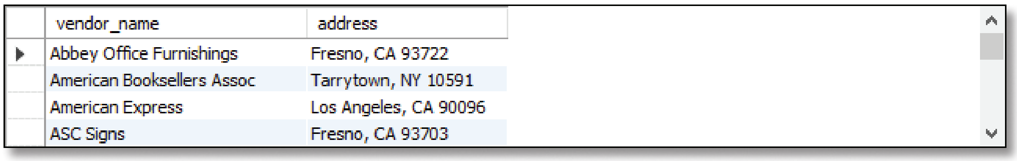
To sort by more than one column, you simply list the names in the ORDER BY clause separated by commas as shown in the third example. This can be referred to as a *nested sort* because one sort is nested within another. Here, the rows in the Vendors table are first sorted by the vendor_state column in ascending sequence. Then, within each state, the rows are sorted by the vendor_city column in ascending sequence. Finally, within each city, the rows are sorted by the vendor_name column in ascending sequence.

The expanded syntax of the ORDER BY clause

```
ORDER BY expression [ASC|DESC][, expression [ASC|DESC]]...
```

An ORDER BY clause that sorts by one column in ascending sequence

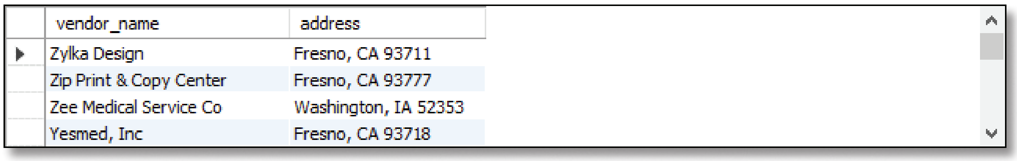
```
SELECT vendor_name,
  CONCAT(vendor_city, ', ', vendor_state, ' ', vendor_zip_code) AS address
FROM vendors
ORDER BY vendor_name
```



vendor_name	address
Abbey Office Furnishings	Fresno, CA 93722
American Booksellers Assoc	Tarrytown, NY 10591
American Express	Los Angeles, CA 90096
ASC Signs	Fresno, CA 93703

An ORDER BY clause that sorts by one column in descending sequence

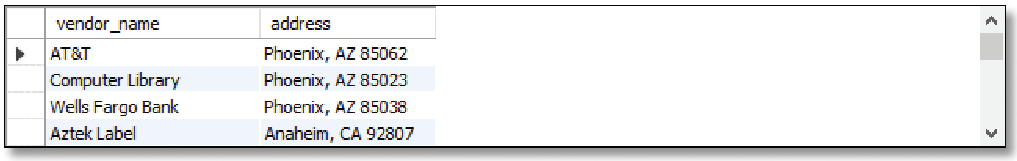
```
SELECT vendor_name,
  CONCAT(vendor_city, ', ', vendor_state, ' ', vendor_zip_code) AS address
FROM vendors
ORDER BY vendor_name DESC
```



vendor_name	address
Zylka Design	Fresno, CA 93711
Zip Print & Copy Center	Fresno, CA 93777
Zee Medical Service Co	Washington, IA 52353
Yesmed, Inc	Fresno, CA 93718

An ORDER BY clause that sorts by three columns

```
SELECT vendor_name,
  CONCAT(vendor_city, ', ', vendor_state, ' ', vendor_zip_code) AS address
FROM vendors
ORDER BY vendor_state, vendor_city, vendor_name
```



vendor_name	address
AT&T	Phoenix, AZ 85062
Computer Library	Phoenix, AZ 85023
Wells Fargo Bank	Phoenix, AZ 85038
Aztek Label	Anaheim, CA 92807

Description

- The ORDER BY clause specifies how you want the rows in the result set sorted. You can sort by one or more columns, and you can sort each column in either ascending (ASC) or descending (DESC) sequence. ASC is the default.
- By default, in an ascending sort, special characters appear first in the sort sequence, followed by numbers, then letters. This sort order is determined by the character set used by the server, which you can change when you start the server.
- Null values appear first in the sort sequence, even if you're using DESC.
- You can sort by any column in the base table regardless of whether it's included in the SELECT clause.

Figure 3-16 How to sort by a column name

How to sort by an alias, expression, or column number

Figure 3-17 presents three more techniques that you can use to specify sort columns. First, you can use a column alias that's defined in the SELECT clause. The first SELECT statement in this figure, for example, sorts by a column named Address, which is an alias for the concatenation of the vendor_city, vendor_state, and vendor_zip_code columns. Notice that within the Address column, the result set is also sorted by the vendor_name column.

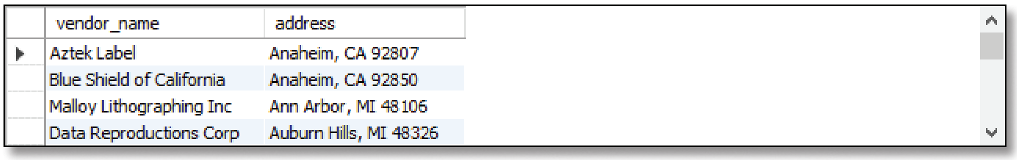
You can also use an arithmetic or string expression in the ORDER BY clause, as shown by the second example in this figure. Here, the expression consists of the vendor_contact_last_name column concatenated with the vendor_contact_first_name column. Notice that neither of these columns is included in the SELECT clause. Although MySQL allows this coding technique, many other SQL dialects don't.

The last example in this figure shows how you can use column numbers to specify a sort order. To use this technique, you code the number that corresponds to the column of the result set, where 1 is the first column, 2 is the second column, and so on. In this example, the ORDER BY clause sorts the result set by the second column, which contains the concatenated address, then by the first column, which contains the vendor name. As a result, this statement returns the same result set that's returned by the first statement.

However, the statement that uses column numbers is more difficult to read because you have to look at the SELECT clause to see what columns the numbers refer to. In addition, if you add or remove columns from the SELECT clause, you may also have to change the ORDER BY clause to reflect the new column positions. As a result, you should avoid using this technique in most situations.

An ORDER BY clause that uses an alias

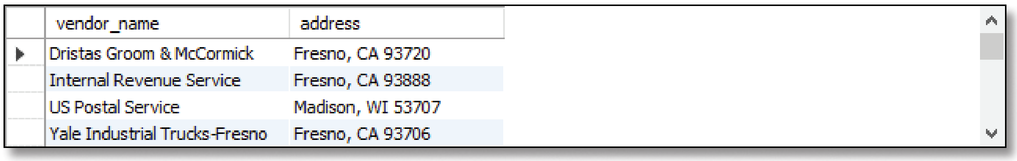
```
SELECT vendor_name,
  CONCAT(vendor_city, ', ', vendor_state, ' ', vendor_zip_code) AS address
FROM vendors
ORDER BY address, vendor_name
```



vendor_name	address
Aztek Label	Anaheim, CA 92807
Blue Shield of California	Anaheim, CA 92850
Malloy Lithographing Inc	Ann Arbor, MI 48106
Data Reproductions Corp	Auburn Hills, MI 48326

An ORDER BY clause that uses an expression

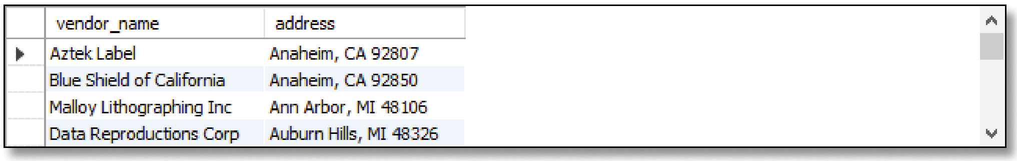
```
SELECT vendor_name,
  CONCAT(vendor_city, ', ', vendor_state, ' ', vendor_zip_code) AS address
FROM vendors
ORDER BY CONCAT(vendor_contact_last_name, vendor_contact_first_name)
```



vendor_name	address
Dristas Groom & McCormick	Fresno, CA 93720
Internal Revenue Service	Fresno, CA 93888
US Postal Service	Madison, WI 53707
Yale Industrial Trucks-Fresno	Fresno, CA 93706

An ORDER BY clause that uses column positions

```
SELECT vendor_name,
  CONCAT(vendor_city, ', ', vendor_state, ' ', vendor_zip_code) AS address
FROM vendors
ORDER BY 2, 1
```



vendor_name	address
Aztek Label	Anaheim, CA 92807
Blue Shield of California	Anaheim, CA 92850
Malloy Lithographing Inc	Ann Arbor, MI 48106
Data Reproductions Corp	Auburn Hills, MI 48326

Description

- The ORDER BY clause can include a column alias that's specified in the SELECT clause if the column alias does not include spaces.
- The ORDER BY clause can include any valid expression. The expression can refer to any column in the base table, even if it isn't included in the result set.
- The ORDER BY clause can use numbers to specify the columns to use for sorting. In that case, 1 represents the first column in the result set, 2 represents the second column, and so on.

Figure 3-17 How to sort by an alias, expression, or column number

How to code the LIMIT clause

The LIMIT clause specifies the maximum number of rows that are returned in the result set. For most queries, you want to see the entire result set so you won't use this clause. However, there may be times when you want to retrieve just a subset of a larger result set.

Figure 3-18 presents the expanded syntax of the LIMIT clause. This clause can take one or two arguments as shown by the three examples in this figure.

How to limit the number of rows

In its simplest form, you code the LIMIT clause with a single numeric argument. Then, the number of rows in the result set is, at most, the number you specify. But if the result set is smaller than the number you specify, the LIMIT clause has no effect.

In the first example, the SELECT statement includes the LIMIT 5 clause, so the entire result set is five rows. Without the LIMIT clause, this statement would return 114 rows. Because the result set is sorted by invoice_total in descending sequence, this result set represents the five largest invoices.

How to return a range of rows

If you code the optional offset argument of the LIMIT clause, it represents an *offset*, or starting point for the result set. This offset starts from a value of 0, which refers to the first row in the result set. In the second example, then, the offset is 2 so the result set starts with the third invoice. Then, since the row count is 3, the result set contains just 3 rows.

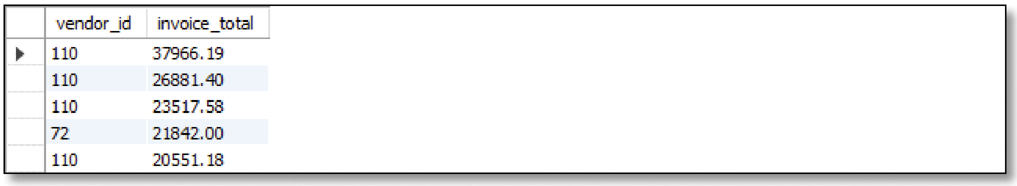
Similarly, the third example has an offset of 100, so the result set starts with row 101. Note that the row count for the LIMIT clause in this example is 1000. Since the table contains only 114 rows, though, the result set contains just the last 14 rows in the table.

The expanded syntax of the LIMIT clause

```
LIMIT [offset,] row_count
```

A SELECT statement with a LIMIT clause that starts with the first row

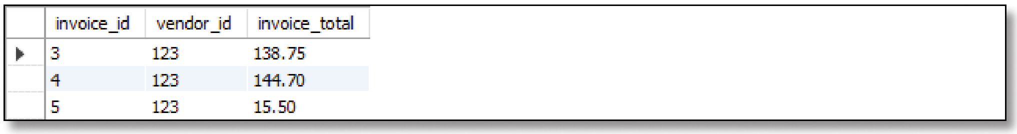
```
SELECT vendor_id, invoice_total
FROM invoices
ORDER BY invoice_total DESC
LIMIT 5
```



vendor_id	invoice_total
110	37966.19
110	26881.40
110	23517.58
72	21842.00
110	20551.18

A SELECT statement with a LIMIT clause that starts with the third row

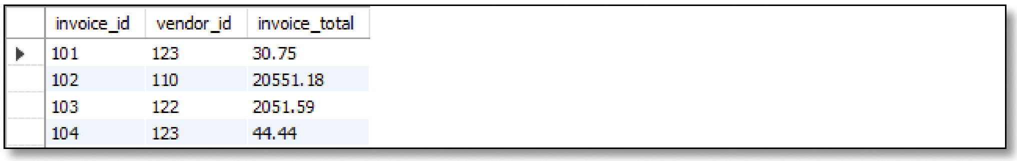
```
SELECT invoice_id, vendor_id, invoice_total
FROM invoices
ORDER BY invoice_id
LIMIT 2, 3
```



invoice_id	vendor_id	invoice_total
3	123	138.75
4	123	144.70
5	123	15.50

A SELECT statement with a LIMIT clause that starts with the 101st row

```
SELECT invoice_id, vendor_id, invoice_total
FROM invoices
ORDER BY invoice_id
LIMIT 100, 1000
```



invoice_id	vendor_id	invoice_total
101	123	30.75
102	110	20551.18
103	122	2051.59
104	123	44.44

```
(14 rows)
```

Description

- You can use the LIMIT clause to limit the number of rows returned by the SELECT statement. This clause takes one or two integer arguments.
- If you code a single argument, it specifies the maximum row count, beginning with the first row. If you code both arguments, the *offset* specifies the first row to return, where the offset of the first row is 0.
- If you want to retrieve all of the rows from a certain offset to the end of the result set, code -1 for the row count.
- Typically, you'll use an ORDER BY clause whenever you use the LIMIT clause.

Figure 3-18 How to code the LIMIT clause

Perspective

The goal of this chapter has been to teach you the basic skills for coding SELECT statements. As a result, you'll use these skills in almost every SELECT statement you code.

As you'll see in the next chapter and in chapters 6 and 7, though, there's a lot more to coding SELECT statements than what's presented here. In these chapters, then, you'll learn additional skills for coding SELECT statements. When you complete these chapters, you'll know everything you need to know about retrieving data from a MySQL database.

Terms

keyword
base table
search condition
filter
Boolean expression
expression
column alias
arithmetic expression
arithmetic operator
order of precedence
literal value
string
function
argument
parameter
concatenate
comparison operator
logical operator
compound condition
subquery
string pattern
mask
wildcard
regular expression
null value
nested sort
offset

Exercises

Run some of the examples in this chapter

In these exercises, you'll use MySQL Workbench to run some of the scripts for the examples in this chapter. This assumes that you already know how to use MySQL Workbench, as described in chapter 2.

1. Start MySQL Workbench.
2. Open the script named 3-02.sql that you should find in this directory: c:\murach\mysql\book_scripts\ch03. When it opens, you should see all of the queries for figure 3-2. Note that each of these queries has a semicolon at the end of it.
3. Move the insertion point into the first query and press Ctrl+Enter or click on the Execute Current Statement button to run the query. This shows you the data that's in the Invoices table that you'll be working with in this chapter.
4. Move the insertion point into the second query and run it.

5. Open the script named 3-05.sql in the ch03 directory. Then, run the second query. When you do, you'll see that the result set is in sequence by the invoice_id column.
6. Delete the ORDER BY clause from the SELECT statement and run the query again. Scroll through the result set to see that the rows are no longer in a particular sequence. When you're done, close the script without saving the changes.
7. Open and run the queries for any of the other examples in this chapter that you're interested in reviewing.

Enter and run your own SELECT statements

In these exercises, you'll enter and run your own SELECT statements. To do that, you can open the script for an example that is similar to the statement you need to write, copy the statement into a new SQL tab, and modify the statement. That can save you both time and syntax errors.

8. Write a SELECT statement that returns three columns from the Vendors table: vendor_name, vendor_contact_last_name, and vendor_contact_first_name. Then, run this statement to make sure it works correctly.

   Add an ORDER BY clause to this statement that sorts the result set by last name and then first name, both in ascending sequence. Then, run this statement again to make sure it works correctly. This is a good way to build and test a statement, one clause at a time.
9. Write a SELECT statement that returns one column from the Vendors table named full_name that joins the vendor_contact_last_name and vendor_contact_first_name columns.

   Format this column with the last name, a comma, a space, and the first name like this:

   ```
   Doe, John
   ```

   Sort the result set by last name and then first name in ascending sequence.

   Return only the contacts whose last name begins with the letter A, B, C, or E. This should retrieve 41 rows.
10. Write a SELECT statement that returns these column names and data from the Invoices table:

Due Date	The invoice_due_date column
Invoice Total	The invoice_total column
10%	10% of the value of invoice_total
Plus 10%	The value of invoice_total plus 10%

    Return only the rows with an invoice total that's greater than or equal to 500 and less than or equal to 1000. This should retrieve 12 rows.

    Sort the result set in descending sequence by invoice_due_date.

11. Write a SELECT statement that returns these columns from the Invoices table:

invoice_number	The invoice_number column
invoice_total	The invoice_total column
payment_credit_total	Sum of the payment_total and credit_total columns
balance_due	The invoice_total column minus the payment_total and credit_total columns

Return only invoices that have a balance due that's greater than $50.

Sort the result set by balance due in descending sequence.

Use the LIMIT clause so the result set contains only the rows with the 5 largest balances.

Work with nulls and test expressions

12. Write a SELECT statement that returns these columns from the Invoices table:

invoice_number	The invoice_number column
invoice_date	The invoice_date column
balance_due	The invoice_total column minus the payment_total and credit_total columns
payment_date	The payment_date column

Return only the rows where the payment_date column contains a null value. This should retrieve 11 rows.

13. Write a SELECT statement without a FROM clause that uses the CURRENT_DATE function to return the current date in its default format.

Use the DATE_FORMAT function to format the current date in this format:

```
mm-dd-yyyy
```

This displays the month, day, and four-digit year of the current date.

Give this column an alias of current_date. To do that, you must enclose the alias in quotes since that name is already used by the CURRENT_DATE function.

14. Write a SELECT statement without a FROM clause that creates a row with these columns:

starting_principal	Starting principal of $50,000
interest	6.5% of the principal
principal_plus_interest	The principal plus the interest

To calculate the third column, add the expressions you used for the first two columns.