Part 1

How to use MySQL Workbench  
and other development tools

Before you start the exercises…

Before you start these exercises, you need to install the MySQL server and MySQL Workbench. The procedures for doing that are provided in appendix A (Windows) and B (macOS).

In addition, you’ll need to get the mgs\_ex\_starts directory from your instructor. This directory contains some script files that you need to do the exercises.

Exercises

In these exercises, you’ll use MySQL Workbench to create the My Guitar Shop database, to review the tables in this database, and to enter SQL statements and run them against this database.

Make sure the MySQL server is running

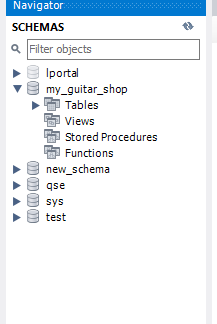
1. Start MySQL Workbench and open a connection for the root user.
2. Check whether the MySQL server is running. If it isn’t, start it.

Use MySQL Workbench to create the My Guitar Shop database.

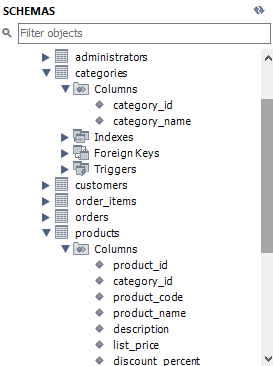
1. Open the script file named my\_guitar\_shop.sql that’s in the mgs\_ex\_starts directory by clicking the Open SQL Script File button in the SQL Editor toolbar. Then, use the resulting dialog box to locate and open the file.
2. Execute the entire script by clicking the Execute SQL Script button in the SQL editor toolbar or by pressing Ctrl+Shift+Enter. When you do, the Output window displays messages that indicate whether the script executed successfully.

Use MySQL Workbench to review the My Guitar Shop database

1. In the Schemas category of the Navigator window, expand the node for the database named my\_guitar\_shop so you can see all of the database objects it contains. If it isn’t displayed in the Schemas tab of the Navigator window, you may need to click on the Refresh button to display it.



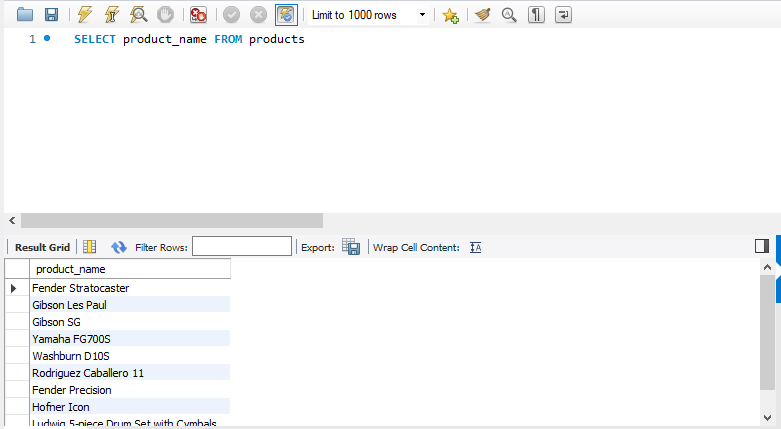
1. View the data for the Categories and Products tables.
2. Navigate through the database objects and view the column definitions for at least the Categories and Products tables.



Use MySQL Workbench to enter and run SQL statements

1. Double-click on the my\_guitar\_shop database to set it as the default database. When you do that, MySQL Workbench should display the database in bold.
2. Open a SQL editor tab. Then, enter and run this SQL statement:

SELECT product\_name FROM products



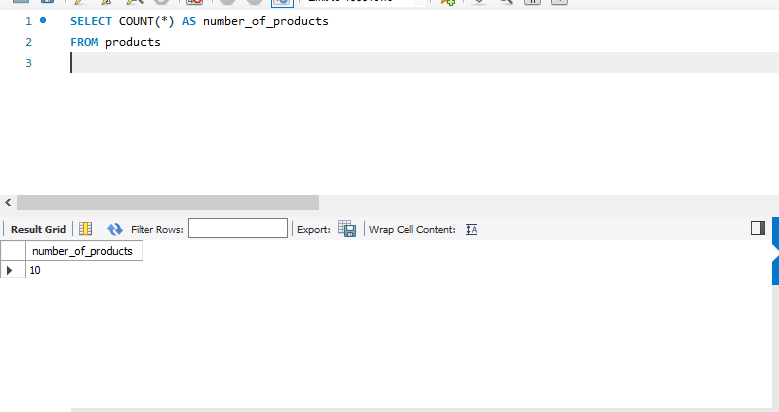
1. Delete the *e* at the end of product\_name and run the statement again. Note the error number and the description of the error.



1. Open another SQL editor tab. Then, enter and run this statement:

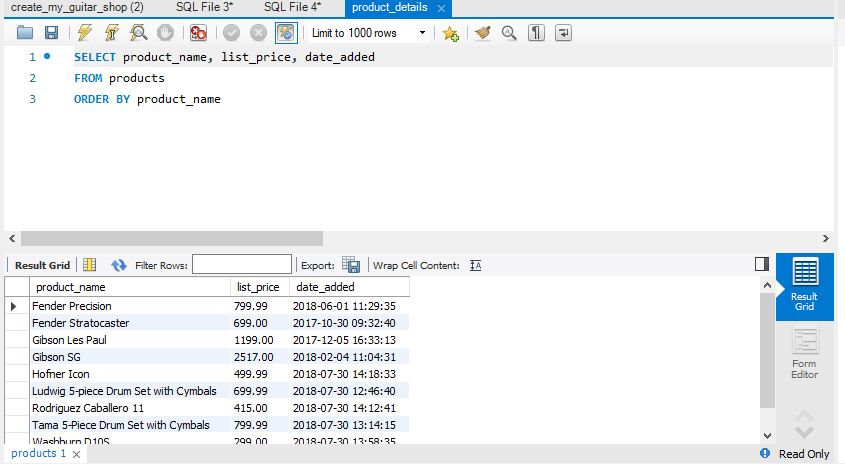
SELECT COUNT(\*) AS number\_of\_products

FROM products

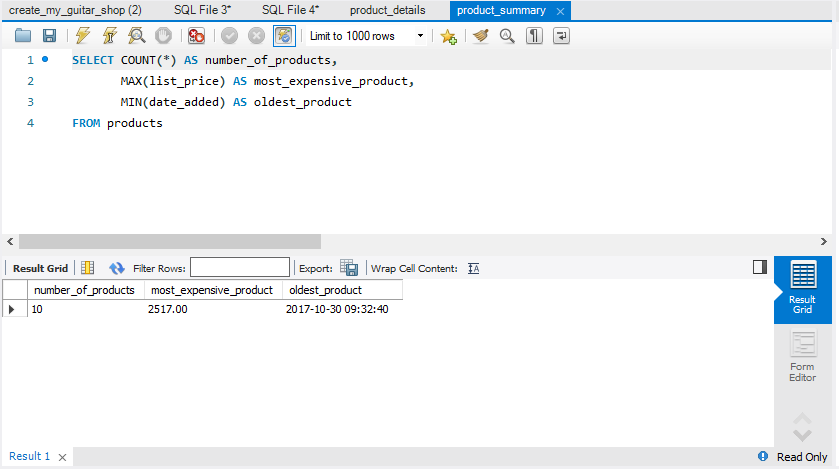


Use MySQL Workbench to open and run scripts

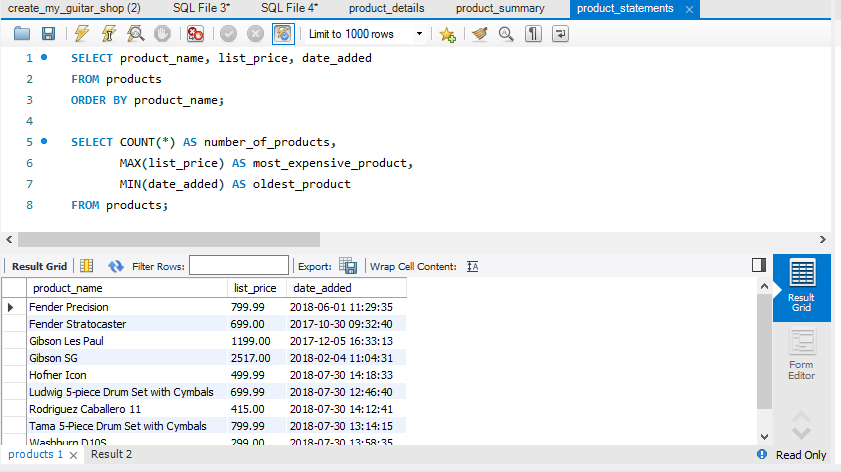
1. Open the script named product\_details.sql that’s in the mgs\_ex\_starts directory. Note that this script contains just one SQL statement. Then, run the statement.



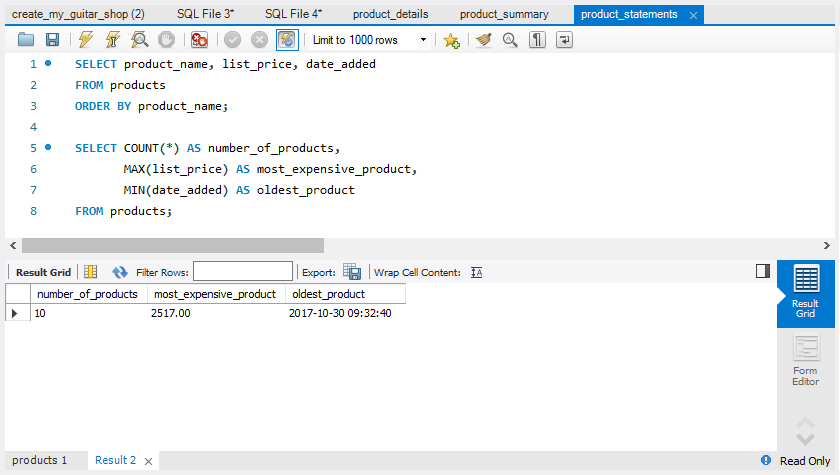
1. Open the script named product\_summary.sql that’s in the mgs\_ex\_starts directory. Note that this opens another SQL editor tab.



1. Open the script named product\_statements.sql that’s in the mgs\_ex\_starts directory. Notice that this script contains two SQL statements that end with semicolons.
2. Press the Ctrl+Shift+Enter keys or click the Execute SQL Script button to run both of the statements in this script. Note that this displays the results in two Result tabs. Make sure to view the results of both SELECT statements.  
   **Result 1:**



**Result 2:**



1. Move the insertion point into the first statement and press Ctrl+Enter to run just that statement.
2. Move the insertion point into the second statement and press Ctrl+Enter to run just that statement.
3. Exit from MySQL Workbench.

Part 2

How to retrieve data   
from a single table

Exercises

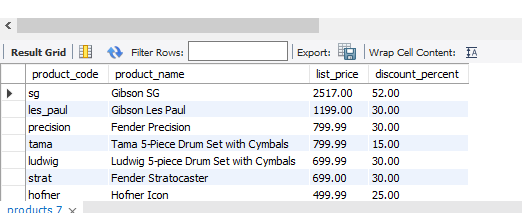
Enter and run your own SELECT statements

In these exercises, you’ll enter and run your own SELECT statements.

1. Write a SELECT statement that returns four columns from the Products table: product\_code, product\_name, list\_price, and discount\_percent. Then, run this statement to make sure it works correctly.

Add an ORDER BY clause to this statement that sorts the result set by list price in descending 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.

**SELECT product\_code, product\_name, list\_price, discount\_percent FROM products ORDER BY list\_price desc**



1. Write a SELECT statement that returns one column from the Customers table named full\_name that joins the last\_name and first\_name columns.

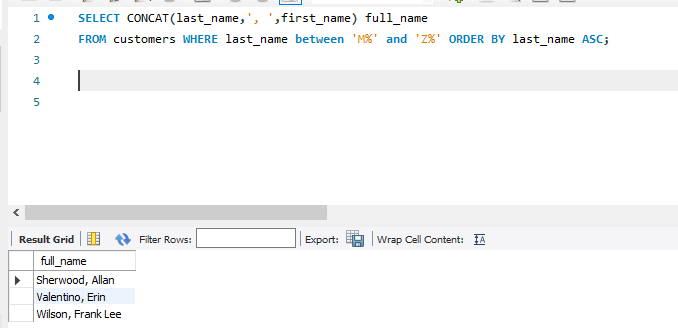
Format this column with the last name, a comma, a space, and the first name like this:

Doe, John

Sort the result set by the last\_name column in ascending sequence.

Return only the customers whose last name begins with letters from M to Z.

NOTE: When comparing strings of characters, ‘M’ comes before any string of characters that begins with ‘M’. For example, ‘M’ comes before ‘Murach’.



1. Write a SELECT statement that returns these columns from the Products table:

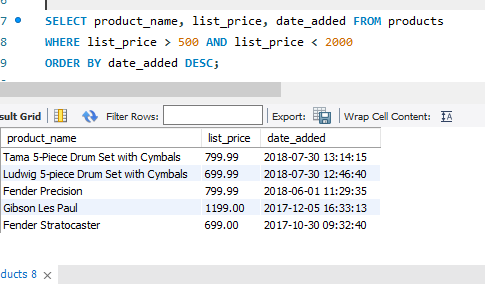
product\_name The product\_name column

list\_price The list\_price column

date\_added The date\_added column

Return only the rows with a list price that’s greater than 500 and less than 2000.

Sort the result set by the date\_added column in descending sequence.



1. Write a SELECT statement that returns these column names and data from the Products table:

product\_name The product\_name column

list\_price The list\_price column

discount\_percent The discount\_percent column

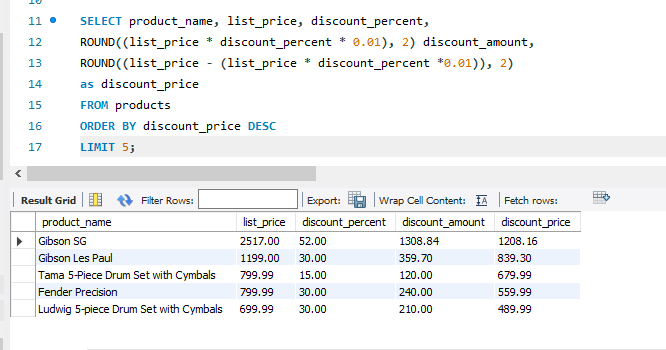
discount\_amount A column that’s calculated from the previous two columns

discount\_price A column that’s calculated from the previous three columns

Round the discount\_amount and discount\_price columns to 2 decimal places.

Sort the result set by the discount\_price column in descending sequence.

Use the LIMIT clause so the result set contains only the first 5 rows.



1. Write a SELECT statement that returns these column names and data from the Order\_Items table:

item\_id The item\_id column

item\_price The item\_price column

discount\_amount The discount\_amount column

quantity The quantity column

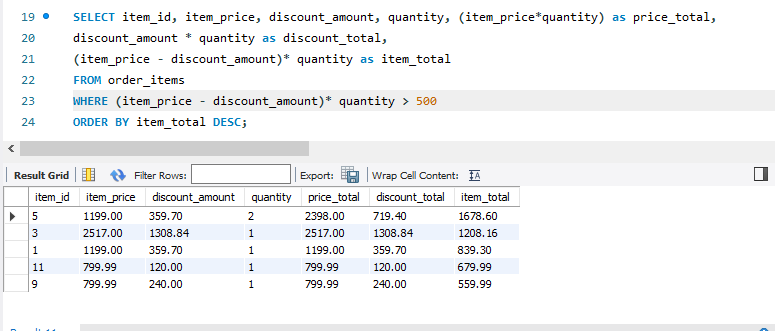
price\_total A column that’s calculated by multiplying the item price by the quantity

discount\_total A column that’s calculated by multiplying the discount amount by the quantity

item\_total A column that’s calculated by subtracting the discount amount from the item price and then multiplying by the quantity

Only return rows where the item\_total is greater than 500.

Sort the result set by the item\_total column in descending sequence.



Work with nulls and test expressions

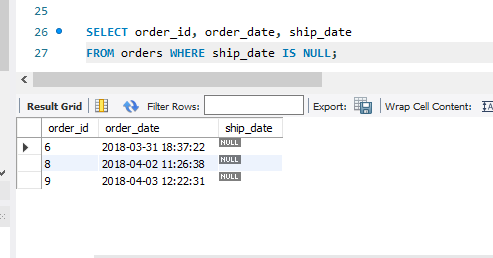
1. Write a SELECT statement that returns these columns from the Orders table:

order\_id The order\_id column

order\_date The order\_date column

ship\_date The ship\_date column

Return only the rows where the ship\_date column contains a null value.

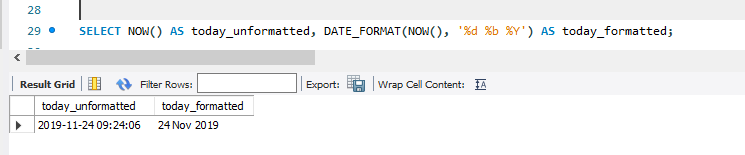


1. Write a SELECT statement without a FROM clause that uses the NOW function to create a row with these columns:

today\_unformatted The NOW function unformatted

today\_formatted The NOW function in this format:   
DD-Mon-YYYY

This displays a number for the day, an abbreviation for the month, and a four-digit year.



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

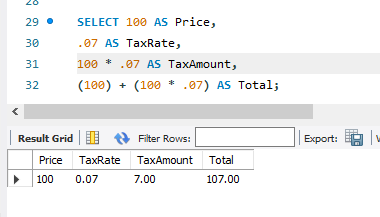
price 100 (dollars)

tax\_rate .07 (7 percent)

tax\_amount The price multiplied by the tax

total The price plus the tax

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



Part 3

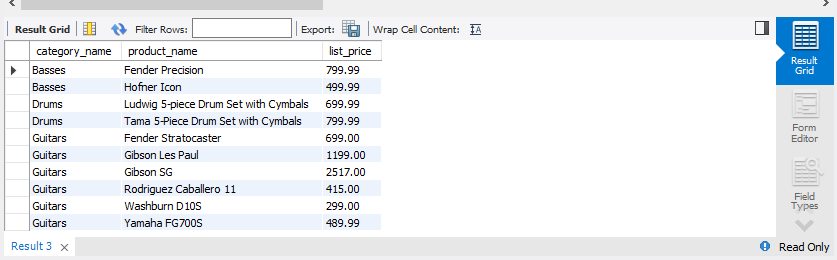
How to retrieve data   
from two or more tables

Exercises

1. Write a SELECT statement that joins the Categories table to the Products table and returns these columns: category\_name, product\_name, list\_price.

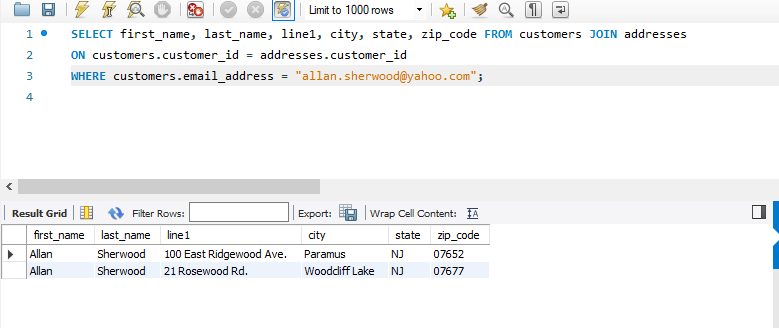
Sort the result set by the category\_name column and then by the product\_name column in ascending sequence.  
**SELECT category\_name, product\_name, list\_price FROM categories JOIN products ON categories.category\_id = products.category\_id**

**ORDER BY category\_name, product\_name asc**



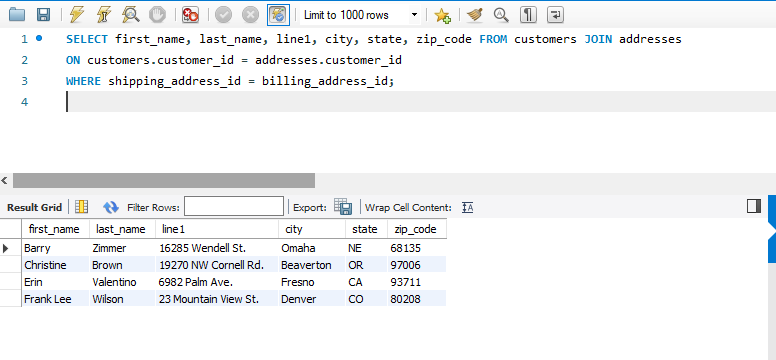
1. Write a SELECT statement that joins the Customers table to the Addresses table and returns these columns: first\_name, last\_name, line1, city, state, zip\_code.

Return one row for each address for the customer with an email address of [allan.sherwood@yahoo.com](mailto:allan.sherwood@yahoo.com).



1. Write a SELECT statement that joins the Customers table to the Addresses table and returns these columns: first\_name, last\_name, line1, city, state, zip\_code.

Return one row for each customer, but only return addresses that are the shipping address for a customer.



1. Write a SELECT statement that joins the Customers, Orders, Order\_Items, and Products tables. This statement should return these columns: last\_name, first\_name, order\_date, product\_name, item\_price, discount\_amount, and quantity.

Use aliases for the tables.

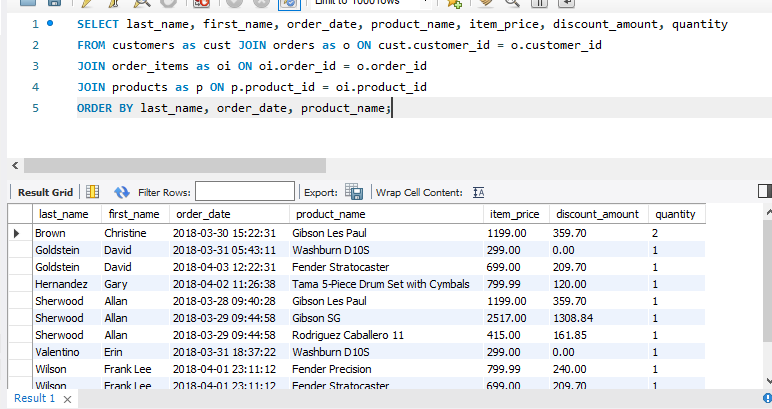
Sort the final result set by the last\_name, order\_date, and product\_name columns.  
**SELECT last\_name, first\_name, order\_date, product\_name, item\_price, discount\_amount, quantity**

**FROM customers as cust JOIN orders as o ON cust.customer\_id = o.customer\_id**

**JOIN order\_items as oi ON oi.order\_id = o.order\_id**

**JOIN products as p ON p.product\_id = oi.product\_id**

**ORDER BY last\_name, order\_date, product\_name;**



1. Write a SELECT statement that returns the product\_name and list\_price columns from the Products table.

Return one row for each product that has the same list price as another product.   
*Hint: Use a self-join to check that the product\_id columns aren’t equal but the list\_price columns are equal.*

Sort the result set by the product\_name column.

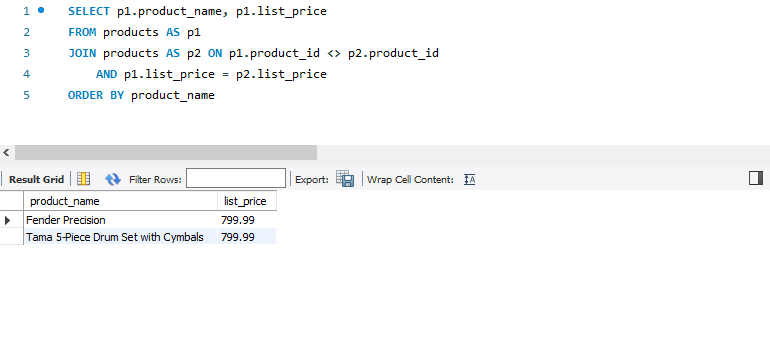
**SELECT p1.product\_name, p1.list\_price**

**FROM products AS p1**

**JOIN products AS p2 ON p1.product\_id <> p2.product\_id**

**AND p1.list\_price = p2.list\_price**

**ORDER BY product\_name**



1. Write a SELECT statement that returns these two columns:

category\_name The category\_name column from the Categories table

product\_id The product\_id column from the Products table

Return one row for each category that has never been used. *Hint: Use an outer join and only return rows where the product\_id column contains a null value.*

**SELECT category\_name**

**, product\_id**

**FROM categories LEFT**

**JOIN products ON categories.category\_id = products.product\_id**

**WHERE product\_id IS NULL**

1. Use the UNION operator to generate a result set consisting of three columns from the Orders table:

ship\_status A calculated column that contains a value of SHIPPED or NOT SHIPPED

order\_id The order\_id column

order\_date The order\_date column

If the order has a value in the ship\_date column, the ship\_status column should contain a value of SHIPPED. Otherwise, it should contain a value of NOT SHIPPED.

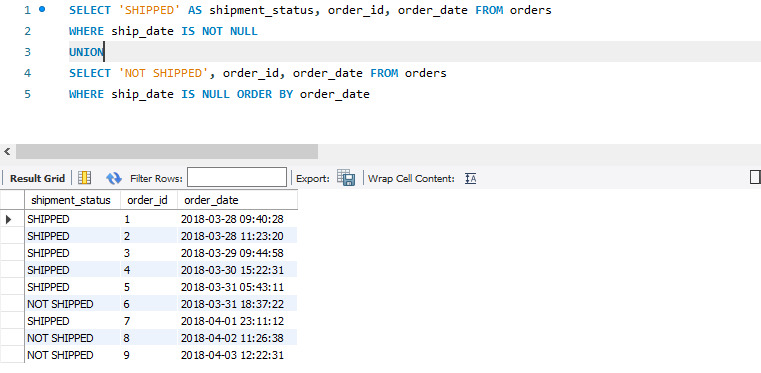
Sort the final result set by the order\_date column.  
**SELECT 'SHIPPED' AS shipment\_status, order\_id, order\_date FROM orders**

**WHERE ship\_date IS NOT NULL**

**UNION**

**SELECT 'NOT SHIPPED', order\_id, order\_date FROM orders**

**WHERE ship\_date IS NULL ORDER BY order\_date**



Part 4

How to insert, update, and delete data

Exercises

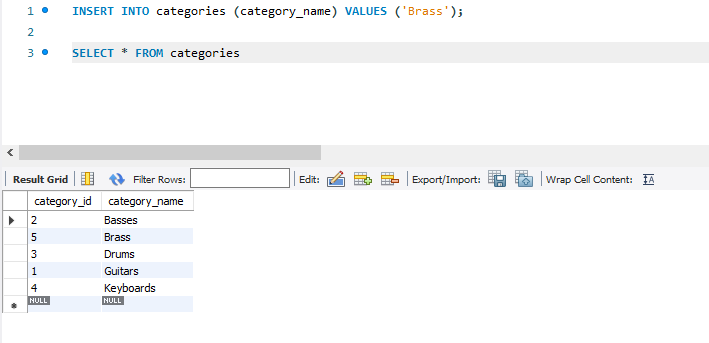
To test whether a table has been modified correctly as you do these exercises, you can write and run an appropriate SELECT statement.

1. Write an INSERT statement that adds this row to the Categories table:

category\_name: Brass

Code the INSERT statement so MySQL automatically generates the category\_id column.

**INSERT INTO categories (category\_name) VALUES (‘Brass’);**



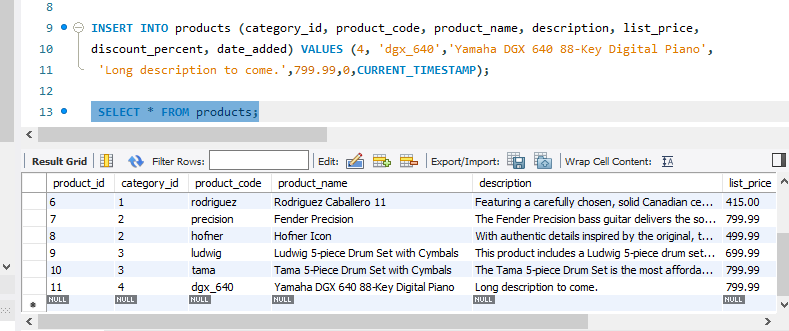
1. Write an UPDATE statement that modifies the row you just added to the Categories table. This statement should change the product\_name column to “Woodwinds”, and it should use the category\_id column to identify the row.  
   **UPDATE categories SET category\_name = 'Woodwinds' WHERE category\_id = 5;**
2. Write a DELETE statement that deletes the row you added to the Categories table in exercise 1. This statement should use the category\_id column to identify the row.  
   **DELETE FROM categories WHERE category\_id = 2;**
3. Write an INSERT statement that adds this row to the Products table:

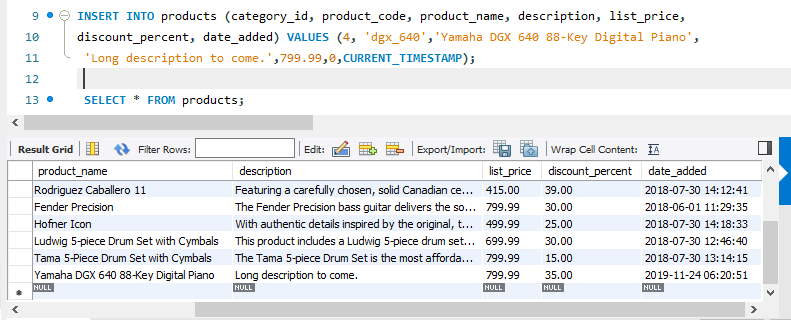
product\_id: The next automatically generated ID   
category\_id: 4  
product\_code: dgx\_640  
product\_name: Yamaha DGX 640 88-Key Digital Piano  
description: Long description to come.  
list\_price: 799.99  
discount\_percent: 0  
date\_added: Today’s date/time.

Use a column list for this statement.  
**INSERT INTO products (category\_id, product\_code, product\_name, description, list\_price,**

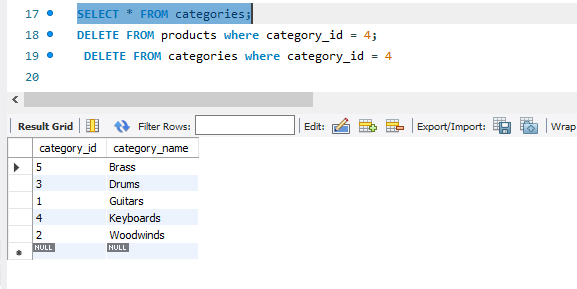
**discount\_percent, date\_added) VALUES (4, 'dgx\_640','Yamaha DGX 640 88-Key Digital Piano',**

**'Long description to come.',799.99,0,CURRENT\_TIMESTAMP);**



1. Write an UPDATE statement that modifies the product you added in exercise 4. This statement should change the discount\_percent column from 0% to 35%.
2. 
3. Write a DELETE statement that deletes the Keyboards category. When you execute this statement, it will produce an error since the category has related rows in the Products table. To fix that, precede the DELETE statement with another DELETE statement that deletes all products in this category. (Remember that to code two or more statements in a script, you must end each statement with a semicolon.)

**SELECT statement reveals the id for keyboards categories and then that id is used to execute delete statements.**



1. Write an INSERT statement that adds this row to the Customers table:

email\_address: rick@raven.com  
password: (empty string)  
first\_name: Rick  
last\_name: Raven

Use a column list for this statement.  
**INSERT INTO customers (email\_address, password, first\_name, last\_name) VALUES ('rick@raven.com', '', 'Rick', 'Raven');**

1. Write an UPDATE statement that modifies the Customers table. Change the password column to “secret” for the customer with an email address of rick@raven.com.  
   **UPDATE customers SET password = 'secret' WHERE email\_address = 'rick@raven.com';**
2. Write an UPDATE statement that modifies the Customers table. Change the password column to “reset” for every customer in the table. If you get an error due to safe-update mode, you can add a LIMIT clause to update the first 100 rows of the table. (This should update all rows in the table.)  
   **UPDATE customers SET password = 'reset';**
3. Open the script named create\_my\_guitar\_shop.sql that’s in the mgs\_ex\_starts directory. Then, run this script. That should restore the data that’s in the database.

Part 5

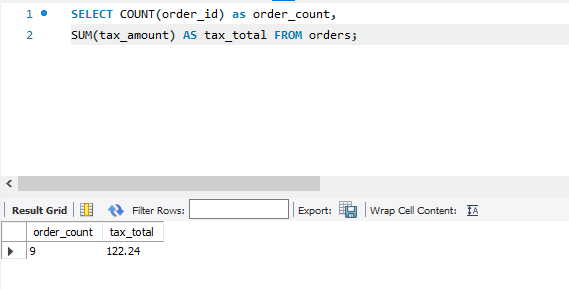
How to code summary queries

Exercises

1. Write a SELECT statement that returns these columns:

The count of the number of orders in the Orders table

The sum of the tax\_amount columns in the Orders table



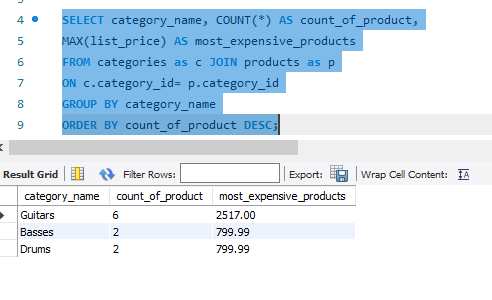
1. Write a SELECT statement that returns one row for each category that has products with these columns:

The category\_name column from the Categories table

The count of the products in the Products table

The list price of the most expensive product in the Products table

Sort the result set so the category with the most products appears first.



1. Write a SELECT statement that returns one row for each customer that has orders with these columns:

The email\_address column from the Customers table

The sum of the item price in the Order\_Items table multiplied by the quantity in the Order\_Items table

The sum of the discount amount column in the Order\_Items table multiplied by the quantity in the Order\_Items table

Sort the result set in descending sequence by the item price total for each customer.  
***SELECT SUM(item\_price \* quantity) AS total\_prices, email\_address,***

***SUM(discount\_amount \* quantity) AS total\_discounted***

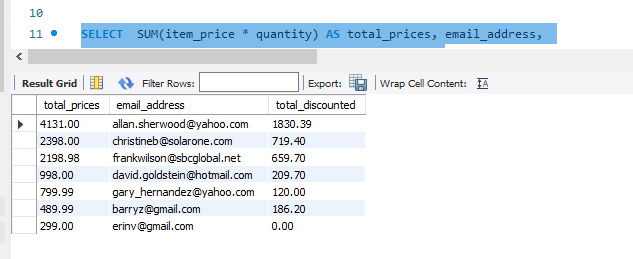
***FROM Customers c***

***JOIN orders as o ON c.customer\_id = o.customer\_id***

***JOIN order\_items as oi ON o.order\_id = oi.order\_id***

***GROUP BY email\_address***

***ORDER BY total\_prices DESC;***



1. Write a SELECT statement that returns one row for each customer that has orders with these columns:

The email\_address column from the Customers table

A count of the number of orders

The total amount for each order (*Hint: First, subtract the discount amount from the price. Then, multiply by the quantity.*)

Return only those rows where the customer has more than 1 order.

Sort the result set in descending sequence by the sum of the line item amounts.  
***SELECT COUNT(o.order\_id) AS order\_count,***

***SUM((item\_price - discount\_amount) \* quantity) AS order\_total, email\_address***

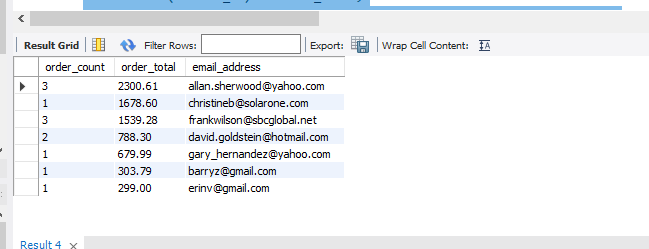
***FROM customers as c***

***JOIN orders as o ON c.customer\_id = o.customer\_id***

***JOIN order\_items as oi ON o.order\_id = oi.order\_id***

***GROUP BY email\_address***

***ORDER BY order\_total DESC;***



1. Modify the solution to exercise 4 so it only counts and totals line items that have an item\_price value that’s greater than 400.

***SELECT COUNT(o.order\_id) AS order\_count,***

***SUM((item\_price - discount\_amount) \* quantity) AS order\_total, email\_address***

***FROM customers as c***

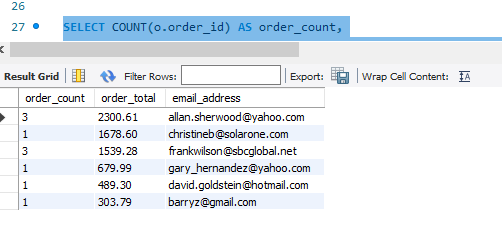
***JOIN orders as o ON c.customer\_id = o.customer\_id***

***JOIN order\_items as oi ON o.order\_id = oi.order\_id***

***WHERE item\_price > 400***

***GROUP BY email\_address***

***ORDER BY order\_total DESC;***

1. 
2. Write a SELECT statement that answers this question: What is the total amount ordered for each product? Return these columns:

The product\_name column from the Products table

The total amount for each product in the Order\_Items table (*Hint: You can calculate the total amount by subtracting the discount amount from the item price and then multiplying it by the quantity*)

Use the WITH ROLLUP operator to include a row that gives the grand total.

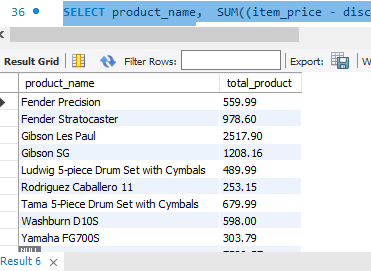
*Note: Once you add the WITH ROLLUP operator, you may need to use MySQL Workbench’s Execute SQL Script button instead of its Execute Current Statement button to execute this statement.*

***SELECT product\_name, SUM((item\_price - discount\_amount) \* quantity) AS total\_product***

***FROM products as p***

***JOIN order\_items as oi ON p.product\_id = oi.product\_id***

***GROUP BY product\_name WITH ROLLUP;***



1. Write a SELECT statement that answers this question: Which customers have ordered more than one product? Return these columns:

The email\_address column from the Customers table

The count of distinct products from the customer’s orders

Sort the result set in ascending sequence by the email\_address column.  
***SELECT email\_address, COUNT(DISTINCT oi.product\_id) AS count\_product***

***FROM customers as c***

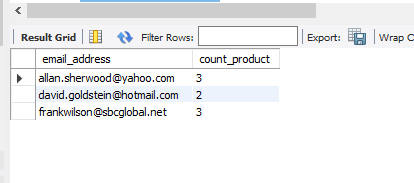
***JOIN orders as o ON c.customer\_id = o.customer\_id***

***JOIN order\_items as oi ON o.order\_id = oi.order\_id***

***GROUP BY email\_address***

***HAVING COUNT(DISTINCT oi.product\_id) > 1***

***ORDER BY email\_address ASC;***



1. Write a SELECT statement that answers this question: What is the total quantity purchased for each product within each category? Return these columns:

The category\_name column from the category table

The product\_name column from the products table

The total quantity purchased for each product with orders in the Order\_Items table

Use the WITH ROLLUP operator to include rows that give a summary for each category name as well as a row that gives the grand total.

Use the IF and GROUPING functions to replace null values in the category\_name and product\_name columns with literal values if they’re for summary rows.

***SELECT category\_name, product\_name, COUNT(quantity) as total\_count***

***FROM products as p***

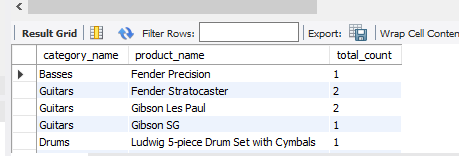
***JOIN order\_items as oi ON p.product\_id = oi.product\_id***

***JOIN categories as c ON c.category\_id = p.category\_id***

***WHERE category\_name IS NOT NULL***

***OR product\_name IS NOT NULL***

***GROUP BY product\_name WITH ROLLUP;***



1. Write a SELECT statement that uses an aggregate window function to get the total amount of each order. Return these columns:

The order\_id column from the Order\_Items table

The total amount for each order item in the Order\_Items table (*Hint: You can calculate the total amount by subtracting the discount amount from the item price and then multiplying it by the quantity*)

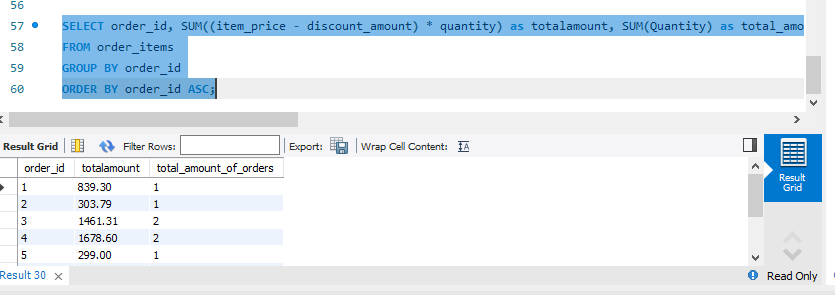
The total amount for each order

Sort the result set in ascending sequence by the order\_id column.

***SELECT order\_id, SUM((item\_price - discount\_amount) \* quantity) as totalamount, SUM(Quantity) OVER (ORDER BY order\_id ASEC)as total\_amount\_of\_orders***

***FROM order\_items***

***GROUP BY order\_id;***



1. Modify the solution to exercise 9 so the column that contains the total amount for each order contains a cumulative total by item amount.

Add another column to the SELECT statement that uses an aggregate window function to get the average item amount for each order.

Modify the SELECT statement so it uses a named window for the two aggregate functions.

***SELECT category\_name, product\_name, COUNT(quantity) OVER() as total\_count***

***FROM products as p***

***JOIN order\_items as oi ON p.product\_id = oi.product\_id***

***JOIN categories as c ON c.category\_id = p.category\_id***

***WHERE category\_name IS NOT NULL***

***OR product\_name IS NOT NULL;***

1. Write a SELECT statement that uses aggregate window functions to calculate the order total for each customer and the order total for each customer by date. Return these columns:

The customer\_id column from the Orders table

The order\_date column from the Orders table

The total amount for each order item in the Order\_Items table

The sum of the order totals for each customer

The sum of the order totals for each customer by date *(Hint: You can create a peer group to get these values)*

***SELECT o.customer\_id, o.order\_date, SUM(oi.item\_price), SUM(oi.quantity)***

***OVER (partition by c.customer\_id), SUM(oi.quantity) OVER (order by o.order\_date)***

***FROM order\_items as o JOIN orders as o ON o.order\_id = oi.order\_id***

***JOIN products as p ON p.customer\_id = o.customer\_id;***

Part 6

How to code subqueries

Exercises

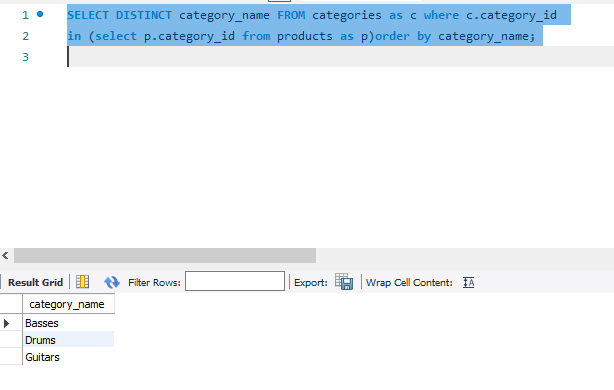
1. Write a SELECT statement that returns the same result set as this SELECT statement, but don’t use a join. Instead, use a subquery in a WHERE clause that uses the IN keyword.

SELECT DISTINCT category\_name

FROM categories c JOIN products p

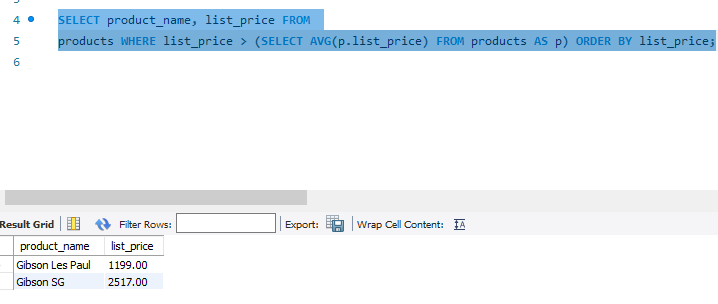
ON c.category\_id = p.category\_id

ORDER BY category\_name



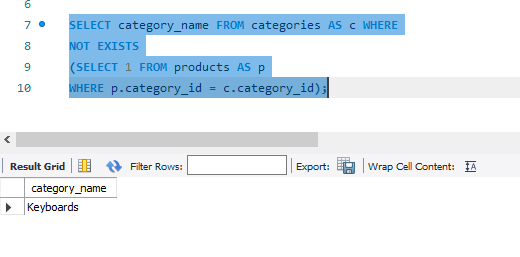
1. Write a SELECT statement that answers this question: Which products have a list price that’s greater than the average list price for all products?

Return the product\_name and list\_price columns for each product.

Sort the result set by the list\_price column in descending sequence.  


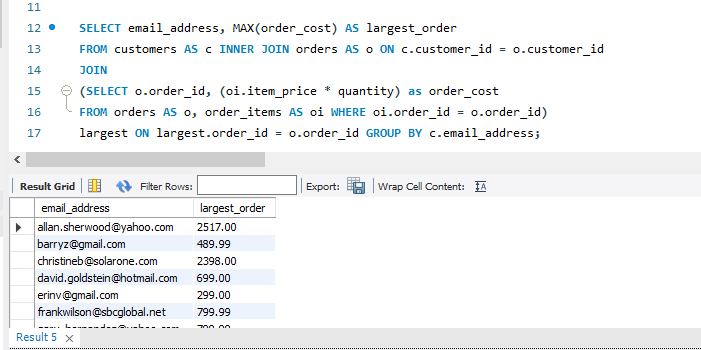
1. Write a SELECT statement that returns the category\_name column from the Categories table.

Return one row for each category that has never been assigned to any product in the Products table. To do that, use a subquery introduced with the NOT EXISTS operator.



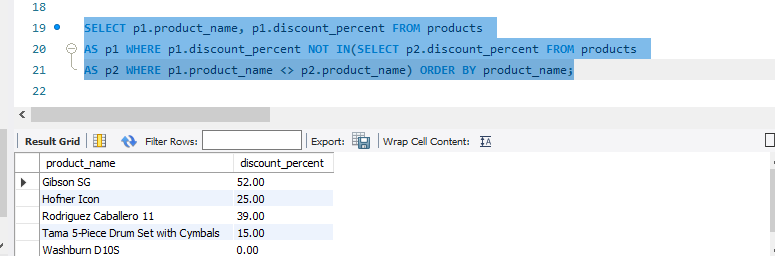
1. Write a SELECT statement that returns three columns: email\_address, order\_id, and the order total for each customer. To do this, you can group the result set by the email\_address and order\_id columns. In addition, you must calculate the order total from the columns in the Order\_Items table.

Write a second SELECT statement that uses the first SELECT statement in its FROM clause. The main query should return two columns: the customer’s email address and the largest order for that customer. To do this, you can group the result set by the email\_address. Sort the result set by the largest order in descending sequence.



1. Write a SELECT statement that returns the name and discount percent of each product that has a unique discount percent. In other words, don’t include products that have the same discount percent as another product.

Sort the result set by the product\_name column.



1. Use a correlated subquery to return one row per customer, representing the customer’s oldest order (the one with the earliest date). Each row should include these three columns: email\_address, order\_id, and order\_date.

Sort the result set by the order\_date and order\_id columns.  
