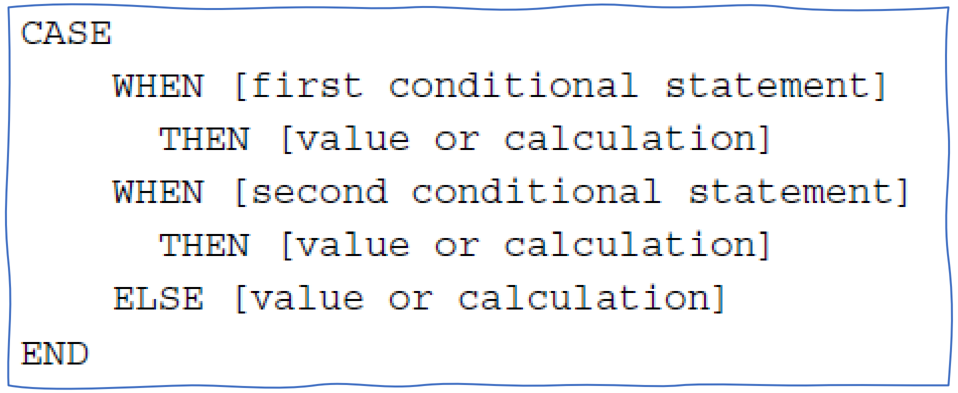
Databases and SQL for Data Science with Python

04/08/2024

CASE Statements

A CASE statement in SQL is used to create conditional logic within queries. It allows you to perform different actions based on different conditions. Here is the general syntax for a CASE statement in SQL:

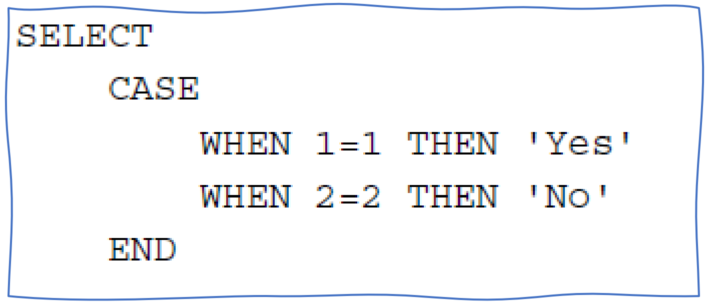


```
CASE
    WHEN [first conditional statement]
      THEN [value or calculation]
    WHEN [second conditional statement]
      THEN [value or calculation]
    ELSE [value or calculation]
END
```

This statement indicates that you want a column to contain different values under different conditions.

CASE Statements

the WHENs are evaluated in order, from top to bottom, and the first time a condition evaluates to TRUE, the corresponding THEN part of the statement is executed, and no other WHEN conditions are evaluated. To illustrate, consider this nonsense query:



```
SELECT
    CASE
        WHEN 1=1 THEN 'Yes'
        WHEN 2=2 THEN 'No'
    END
```

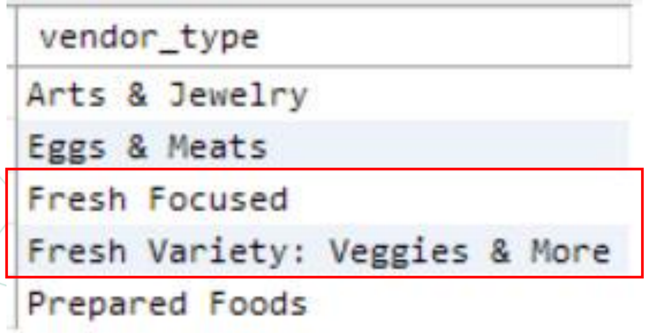
This query will always evaluate to “Yes,” because 1=1 is always TRUE, and therefore the 2=2 conditional statement is never evaluated, even though it is also true.

The ELSE part of the statement is optional, and that value or calculation result is returned if none of the conditional statements above it evaluate to TRUE.

If the ELSE is not included and none of the WHEN conditionals evaluate to TRUE, the resulting value will be NULL.

CASE Statements

Let’s say that we want to know which vendors primarily sell fresh produce and which don’t.



vendor_type
Arts & Jewelry
Eggs & Meats
Fresh Focused
Fresh Variety: Veggies & More
Prepared Foods

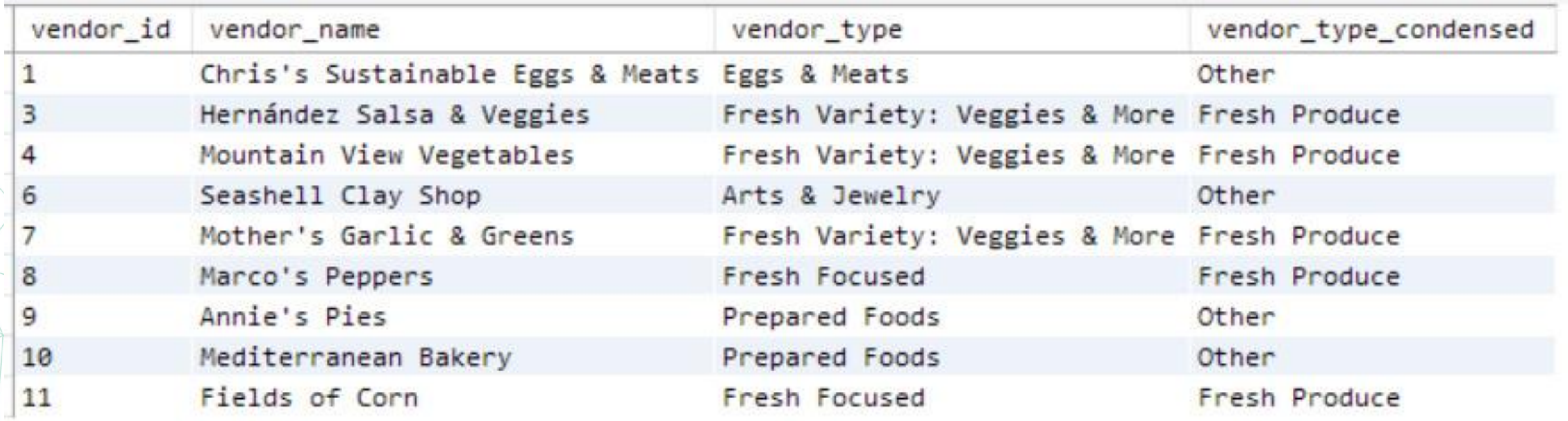
The vendors we want to label as “Fresh Produce” have the word “Fresh” in the vendor_type column.

We can use a CASE statement and the LIKE operator to create a new column:

```
SELECT
    vendor_id,
    vendor_name,
    vendor_type,
    CASE
        WHEN LOWER(vendor_type) LIKE '%fresh%'
          THEN 'Fresh Produce'
        ELSE 'Other'
    END AS vendor_type_condensed
FROM farmers_market.vendor
```

CASE Statements

This gives us:



vendor_id	vendor_name	vendor_type	vendor_type_condensed
1	Chris's Sustainable Eggs & Meats	Eggs & Meats	Other
3	Hernández Salsa & Veggies	Fresh Variety: Veggies & More	Fresh Produce
4	Mountain View Vegetables	Fresh Variety: Veggies & More	Fresh Produce
6	Seashell Clay Shop	Arts & Jewelry	Other
7	Mother's Garlic & Greens	Fresh Variety: Veggies & More	Fresh Produce
8	Marco's Peppers	Fresh Focused	Fresh Produce
9	Annie's Pies	Prepared Foods	Other
10	Mediterranean Bakery	Prepared Foods	Other
11	Fields of Corn	Fresh Focused	Fresh Produce

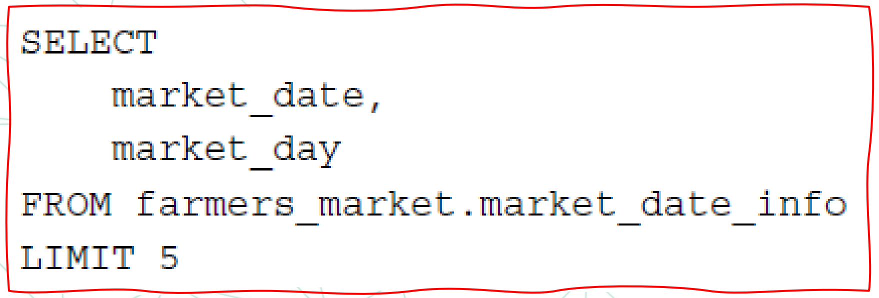
We’re using the LOWER() function to lowercase the vendor type string, because we don’t want the comparison to fail because of capitalization.

CASE Statements: Creating Binary Flags

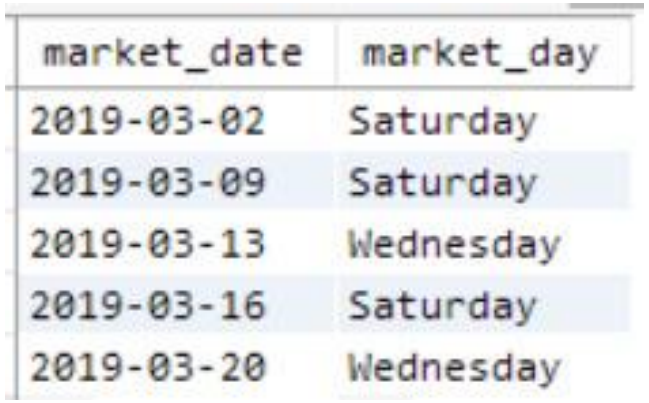
A CASE statement can create a binary flag field, commonly found in machine learning datasets.

This field contains only 1s or 0s, indicating values like "Yes/No" or "Exists/Doesn't exist."

For example, in our database, Farmer's Markets occur on Wednesday evenings or Saturday mornings. Machine learning algorithms may not understand the "Wednesday" and "Saturday" in the market_day column.



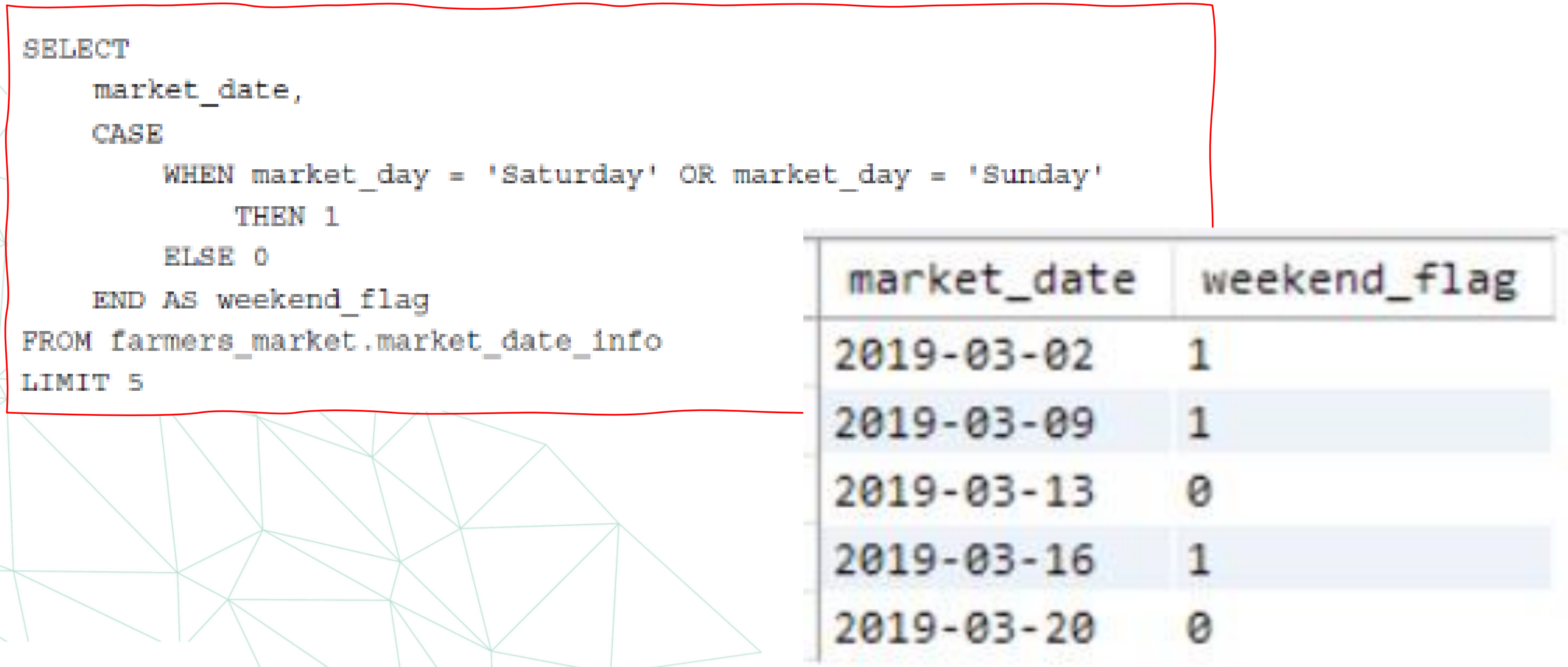
```
SELECT
    market_date,
    market_day
FROM farmers_market.market_date_info
LIMIT 5
```



market_date	market_day
2019-03-02	Saturday
2019-03-09	Saturday
2019-03-13	Wednesday
2019-03-16	Saturday
2019-03-20	Wednesday

CASE Statements: Creating Binary Flags

We can do this with a CASE statement, making a new column that contains a 1 if the market occurs on a Saturday or Sunday, and a 0 if it doesn't:



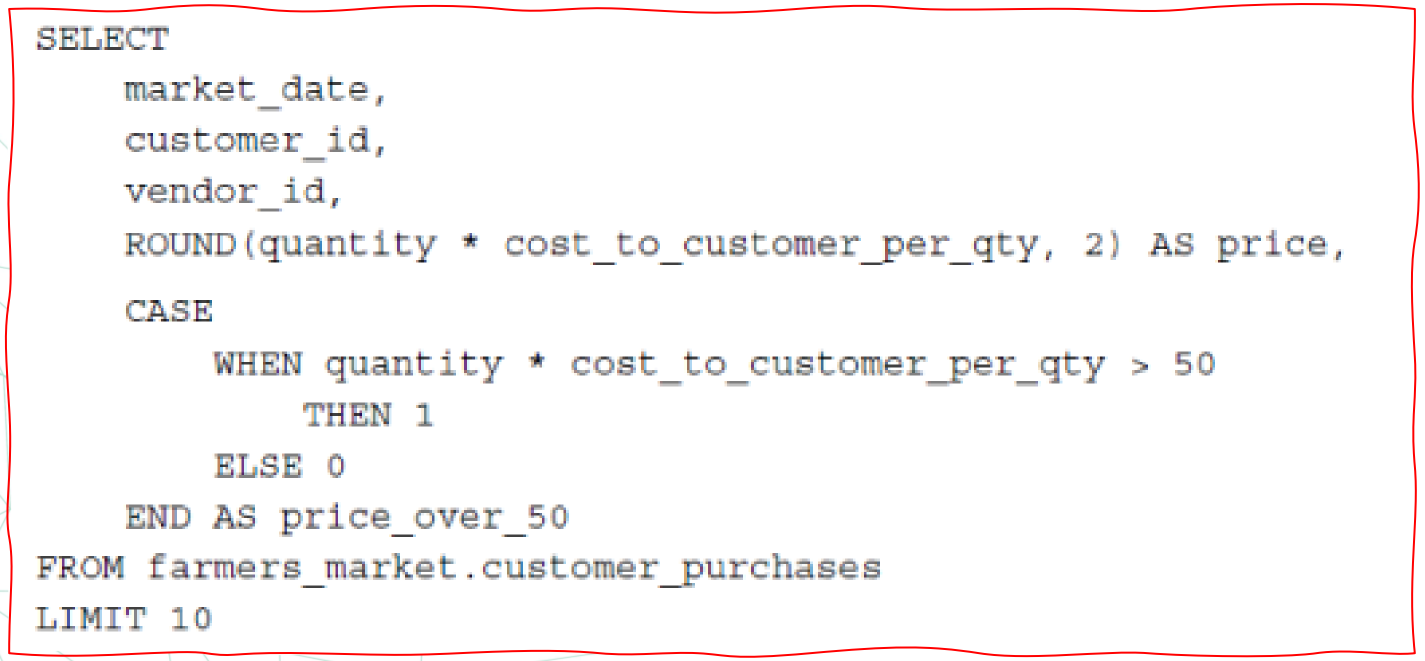
```
SELECT
    market_date,
    CASE
        WHEN market_day = 'Saturday' OR market_day = 'Sunday'
            THEN 1
        ELSE 0
    END AS weekend_flag
FROM farmers_market.market_date_info
LIMIT 5
```

market_date	weekend_flag
2019-03-02	1
2019-03-09	1
2019-03-13	0
2019-03-16	1
2019-03-20	0

CASE Statements: Binning Continuous Values

We initially filtered customer purchases with items costing over $50 using a conditional statement in the WHERE clause.

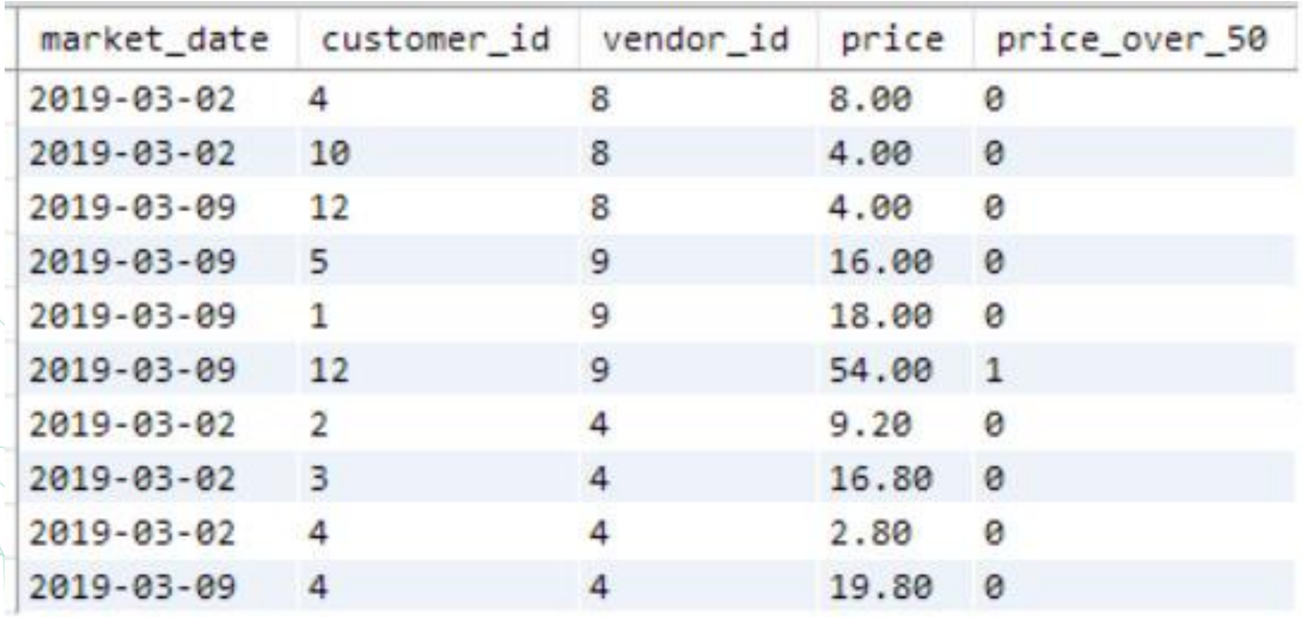
Instead, if we want to return all rows and indicate whether the cost was over $50, we can write the query as follows:



```
SELECT
    market_date,
    customer_id,
    vendor_id,
    ROUND(quantity * cost_to_customer_per_qty, 2) AS price,
    CASE
        WHEN quantity * cost_to_customer_per_qty > 50
            THEN 1
        ELSE 0
    END AS price_over_50
FROM farmers_market.customer_purchases
LIMIT 10
```

CASE Statements: Binning Continuous Values

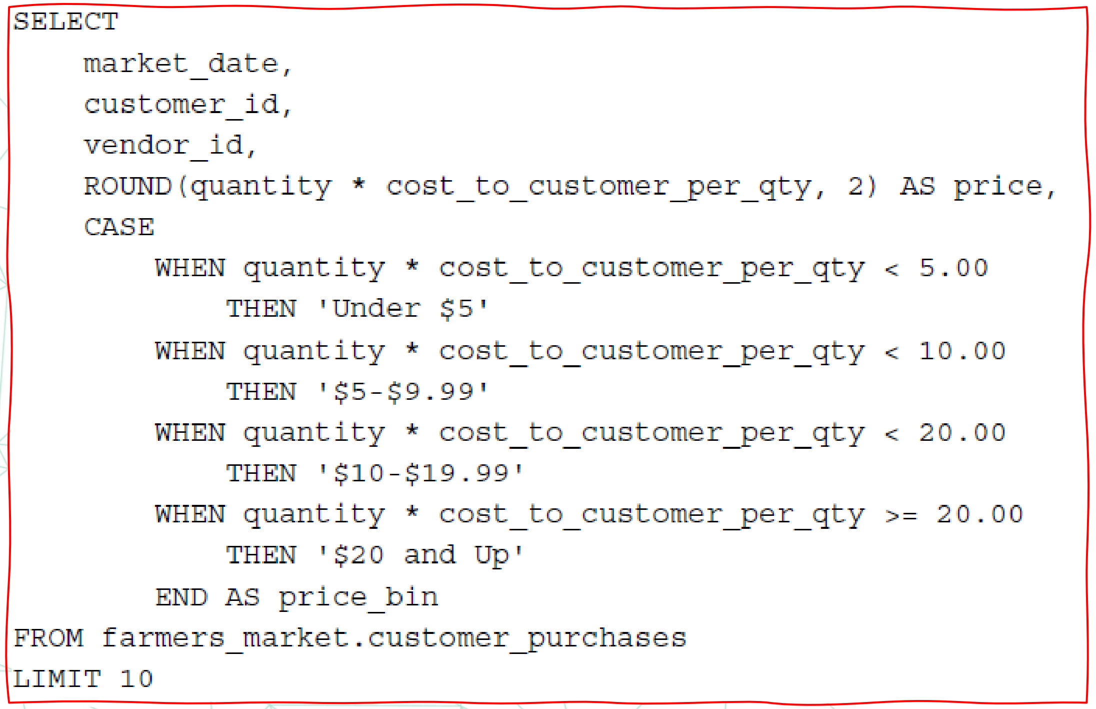
The final column now contains a flag indicating which item purchases were over $50.



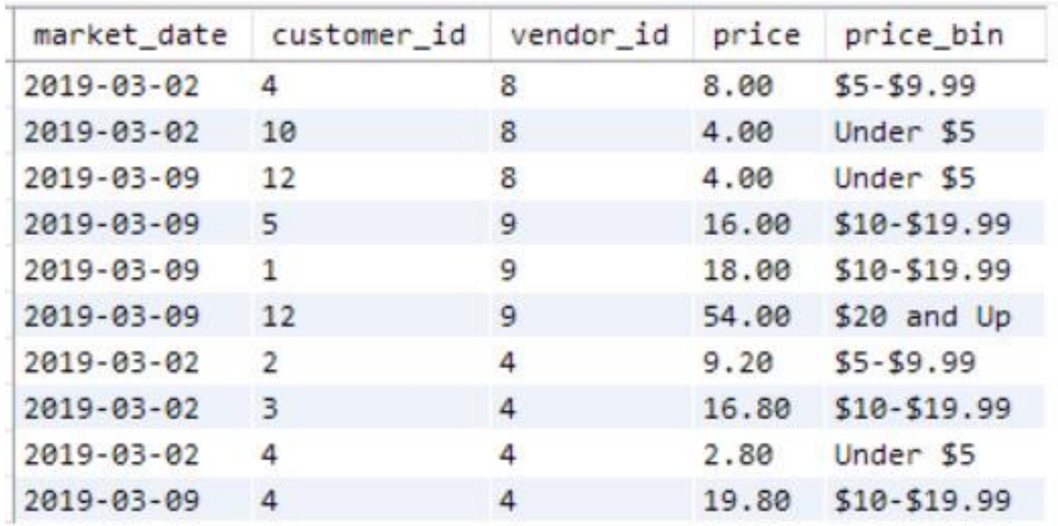
market_date	customer_id	vendor_id	price	price_over_50
2019-03-02	4	8	8.00	0
2019-03-02	10	8	4.00	0
2019-03-09	12	8	4.00	0
2019-03-09	5	9	16.00	0
2019-03-09	1	9	18.00	0
2019-03-09	12	9	54.00	1
2019-03-02	2	4	9.20	0
2019-03-02	3	4	16.80	0
2019-03-02	4	4	2.80	0
2019-03-09	4	4	19.80	0

CASE Statements: Binning Continuous Values

CASE statements can "bin" a continuous variable, like price. For example, we can categorize line-item customer purchases into bins: under $5.00, $5.00–$9.99, $10.00 $19.99, and $20.00 and over.



```
SELECT
    market_date,
    customer_id,
    vendor_id,
    ROUND(quantity * cost_to_customer_per_qty, 2) AS price,
    CASE
        WHEN quantity * cost_to_customer_per_qty < 5.00
            THEN 'Under $5'
        WHEN quantity * cost_to_customer_per_qty < 10.00
            THEN '$5-$9.99'
        WHEN quantity * cost_to_customer_per_qty < 20.00
            THEN '$10-$19.99'
        WHEN quantity * cost_to_customer_per_qty >= 20.00
            THEN '$20 and Up'
        END AS price_bin
FROM farmers_market.customer_purchases
LIMIT 10
```

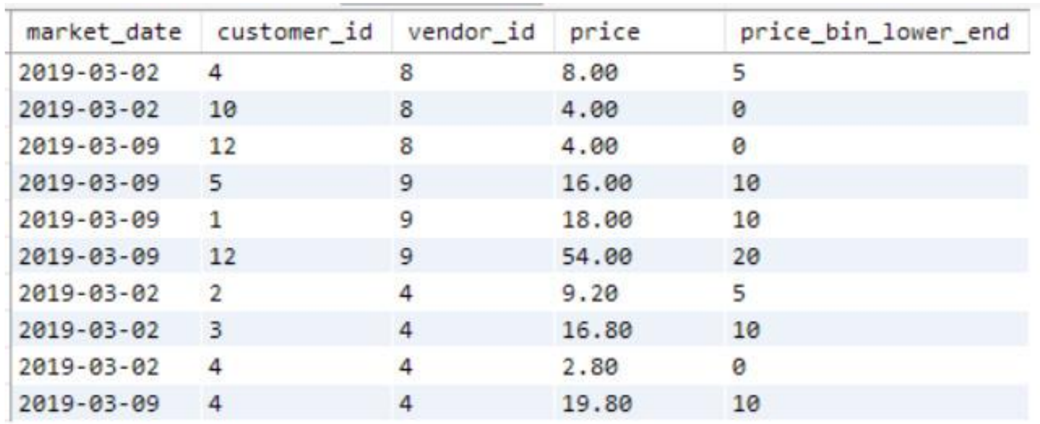


market_date	customer_id	vendor_id	price	price_bin
2019-03-02	4	8	8.00	$5-$9.99
2019-03-02	10	8	4.00	Under $5
2019-03-09	12	8	4.00	Under $5
2019-03-09	5	9	16.00	$10-$19.99
2019-03-09	1	9	18.00	$10-$19.99
2019-03-09	12	9	54.00	$20 and Up
2019-03-02	2	4	9.20	$5-$9.99
2019-03-02	3	4	16.80	$10-$19.99
2019-03-02	4	4	2.80	Under $5
2019-03-09	4	4	19.80	$10-$19.99

CASE Statements: Binning Continuous Values

Or, if the result needs to be numeric, a different approach is to output the bottom end of the numeric range:

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    ROUND(quantity * cost_to_customer_per_qty, 2) AS price,
    CASE
        WHEN quantity * cost_to_customer_per_qty < 5.00
            THEN 0
        WHEN quantity * cost_to_customer_per_qty < 10.00
            THEN 5
        WHEN quantity * cost_to_customer_per_qty < 20.00
            THEN 10
        WHEN quantity * cost_to_customer_per_qty >= 20.00
            THEN 20
    END AS price_bin_lower_end
FROM farmers_market.customer_purchases
LIMIT 10
```

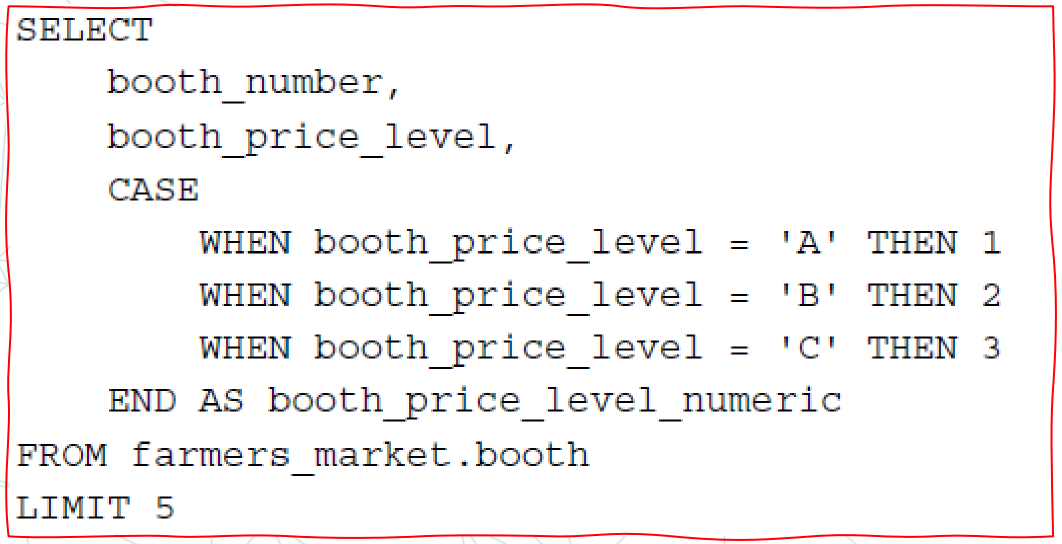


market_date	customer_id	vendor_id	price	price_bin_lower_end
2019-03-02	4	8	8.00	5
2019-03-02	10	8	4.00	0
2019-03-09	12	8	4.00	0
2019-03-09	5	9	16.00	10
2019-03-09	1	9	18.00	10
2019-03-09	12	9	54.00	20
2019-03-02	2	4	9.20	5
2019-03-02	3	4	16.80	10
2019-03-02	4	4	2.80	0
2019-03-09	4	4	19.80	10

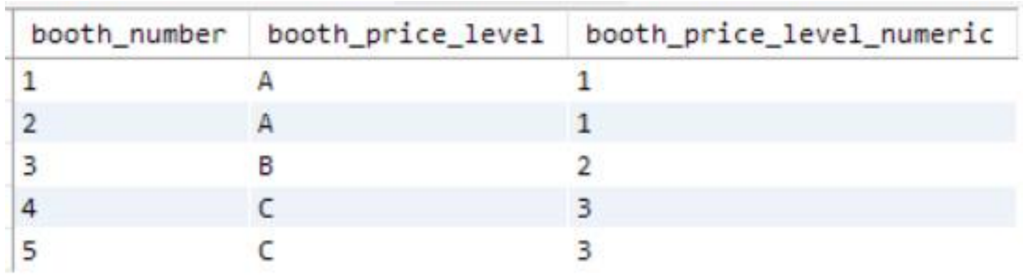
CASE Statements: Categorical Encoding

When creating datasets for machine learning, you often need to encode categorical strings as numeric values for algorithms to use them.

If the categories represent something that can be sorted in a rank order, it might make sense to convert the string variables into numeric values that represent that rank order.



```
SELECT
    booth_number,
    booth_price_level,
    CASE
        WHEN booth_price_level = 'A' THEN 1
        WHEN booth_price_level = 'B' THEN 2
        WHEN booth_price_level = 'C' THEN 3
    END AS booth_price_level_numeric
FROM farmers_market.booth
LIMIT 5
```

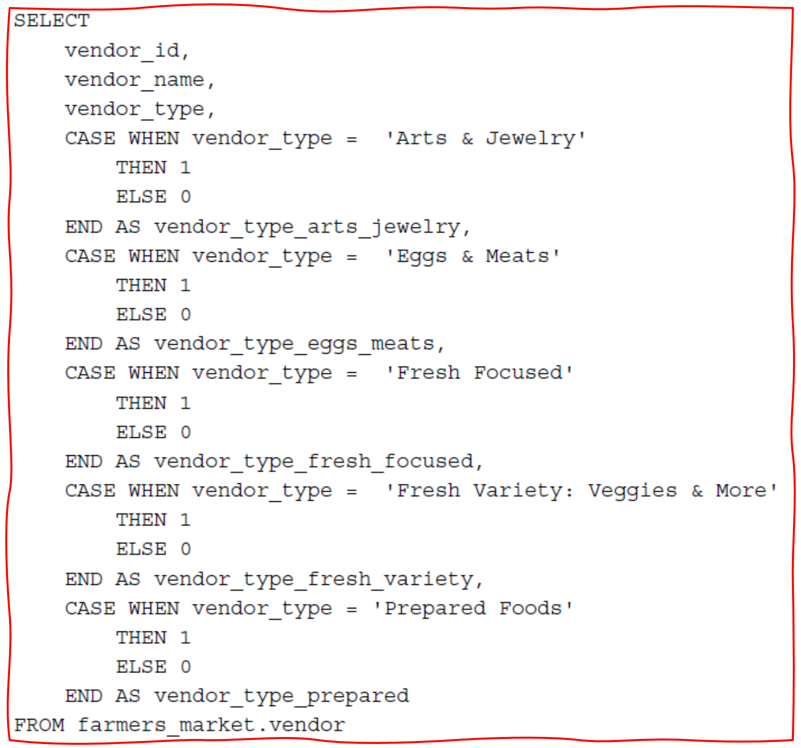


booth_number	booth_price_level	booth_price_level_numeric
1	A	1
2	A	1
3	B	2
4	C	3
5	C	3

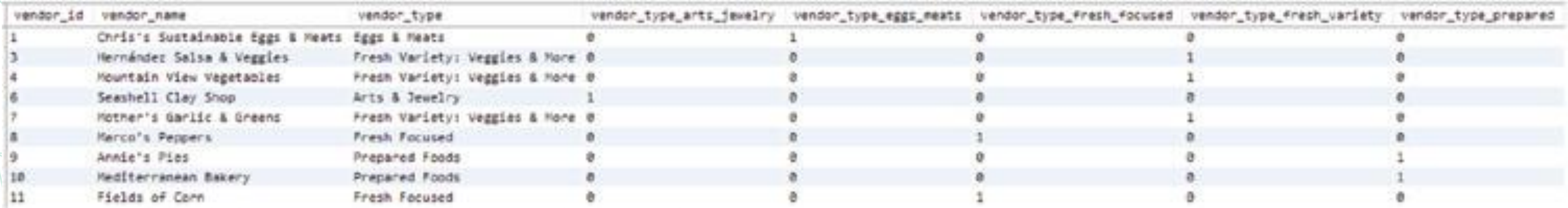
CASE Statements: Categorical Encoding

If the categories aren't necessarily in any kind of rank order, like our vendor type categories, we might use a method called "one-hot encoding."

This helps us avoid inadvertently indicating a sort order when none exists.



```
SELECT
    vendor_id,
    vendor_name,
    vendor_type,
    CASE WHEN vendor_type =  'Arts & Jewelry'
        THEN 1
        ELSE 0
    END AS vendor_type_arts_jewelry,
    CASE WHEN vendor_type =  'Eggs & Meats'
        THEN 1
        ELSE 0
    END AS vendor_type_eggs_meats,
    CASE WHEN vendor_type =  'Fresh Focused'
        THEN 1
        ELSE 0
    END AS vendor_type_fresh_focused,
    CASE WHEN vendor_type =  'Fresh Variety: Veggies & More'
        THEN 1
        ELSE 0
    END AS vendor_type_fresh_variety,
    CASE WHEN vendor_type = 'Prepared Foods'
        THEN 1
        ELSE 0
    END AS vendor_type_prepared
FROM farmers_market.vendor
```



vendor_id	vendor_name	vendor_type	vendor_type_arts_jewelry	vendor_type_eggs_meats	vendor_type_fresh_focused	vendor_type_fresh_variety	vendor_type_prepared
1	Chris's Sustainable Eggs & Meats	Eggs & Meats	0	1	0	0	0
3	Hernández Salsa & Veggies	Fresh Variety: Veggies & More	0	0	0	1	0
4	Mountain View Vegetables	Fresh Variety: Veggies & More	0	0	0	1	0
6	Seashell Clay Shop	Arts & Jewelry	1	0	0	0	0
7	Mother's Garlic & Greens	Fresh Variety: Veggies & More	0	0	0	1	0
8	Marco's Peppers	Fresh Focused	0	0	1	0	0
9	Annie's Pies	Prepared Foods	0	0	0	0	1
10	Mediterranean Bakery	Prepared Foods	0	0	0	0	1
11	Fields of Corn	Fresh Focused	0	0	1	0	0

CASE Statements: Categorical Encoding

A situation to be aware of when manually encoding one-hot categorical variables this way is that if a new category is added (a new type of vendor in this case), there will be no column in your dataset for the new vendor type until you add another CASE statement.

Exercise

For the product table:

1. Products can be sold by the individual unit or by bulk measures like lbs. or oz. Write a query that outputs the product_id and product_name columns from the product table, and add a column called prod_qty_type_condensed that displays the word “unit” if the product_qty_type is “unit,” and otherwise displays the word “bulk.”

2. We want to flag all of the different types of pepper products that are sold at the market. Add a column to the previous query called pepper_flag that outputs a 1 if the product_name contains the word “pepper” (regardless of capitalization), and otherwise outputs 0.

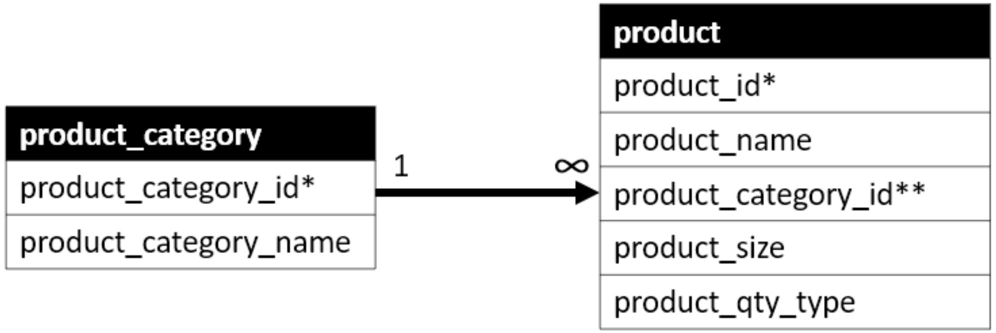
SQL - JOINs

SQL JOINs

The type of relationship between database tables, and the key fields that connect them, give us information we need to combine them using a JOIN statement in SQL.

For example, to list each product with its category name, we need to join the product table (which has only category IDs) with the product_category table (which contains category names).

Figure illustrates a one-to-many relationship: each product belongs to one category, but each category can include many products. The primary key in the product_category table is `product_category_id`, which is also used as a foreign key in the product table to link products to their categories.

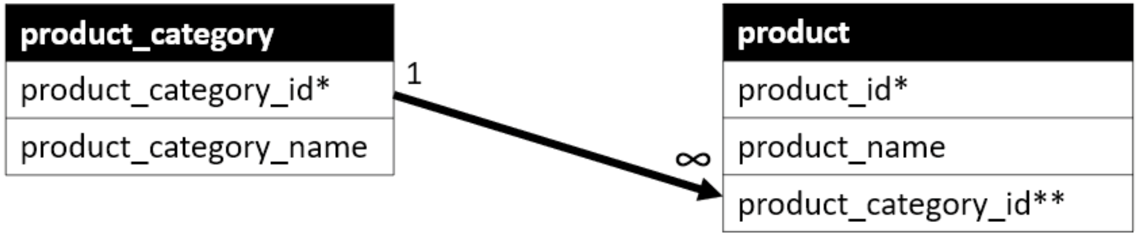


product_category
product_category_id*
product_category_name

product
product_id*
product_name
product_category_id**
product_size
product_qty_type

SQL JOINs

In order to combine these tables, we need to figure out which type of JOIN to use. To illustrate the different types of SQL JOINs, we'll use the two tables from the Farmer's Market database:



product_category. product_category_id*	product_category. product_category_name
1	Fresh Fruits & Vegetables
3	Packaged Prepared Food
5	Plants & Flowers
6	Eggs & Meat

product. product_id	product. product_name	product. product_category_id**
2	Jalapeno Peppers - Organic	1
4	Banana Peppers - Jar	3
5	Whole Wheat Bread	3
6	Cut Zinnias Bouquet	5
7	Apple Pie	3
13	Baby Salad Lettuce Mix	1
99	Handmade Candle	NULL

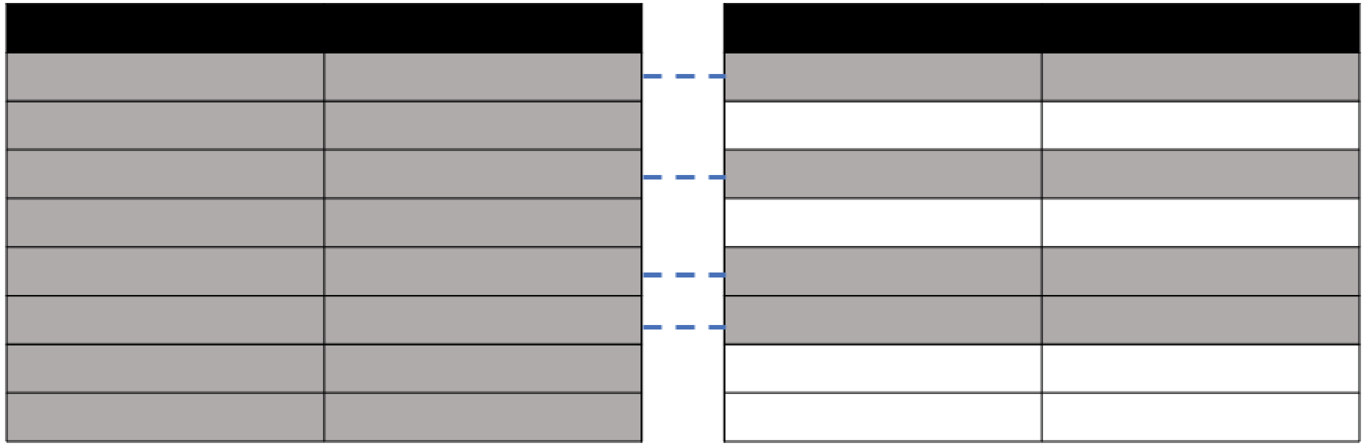
SQL JOINs: **LEFT** JOIN

The first type of JOIN we'll cover is the one I've seen used most frequently when building analytical datasets: the LEFT JOIN.

This tells the query to pull all records from the table on the "left side" of the JOIN, and only the matching records (based on the criteria specified in the JOIN clause) from the table on the "right side" of the JOIN.

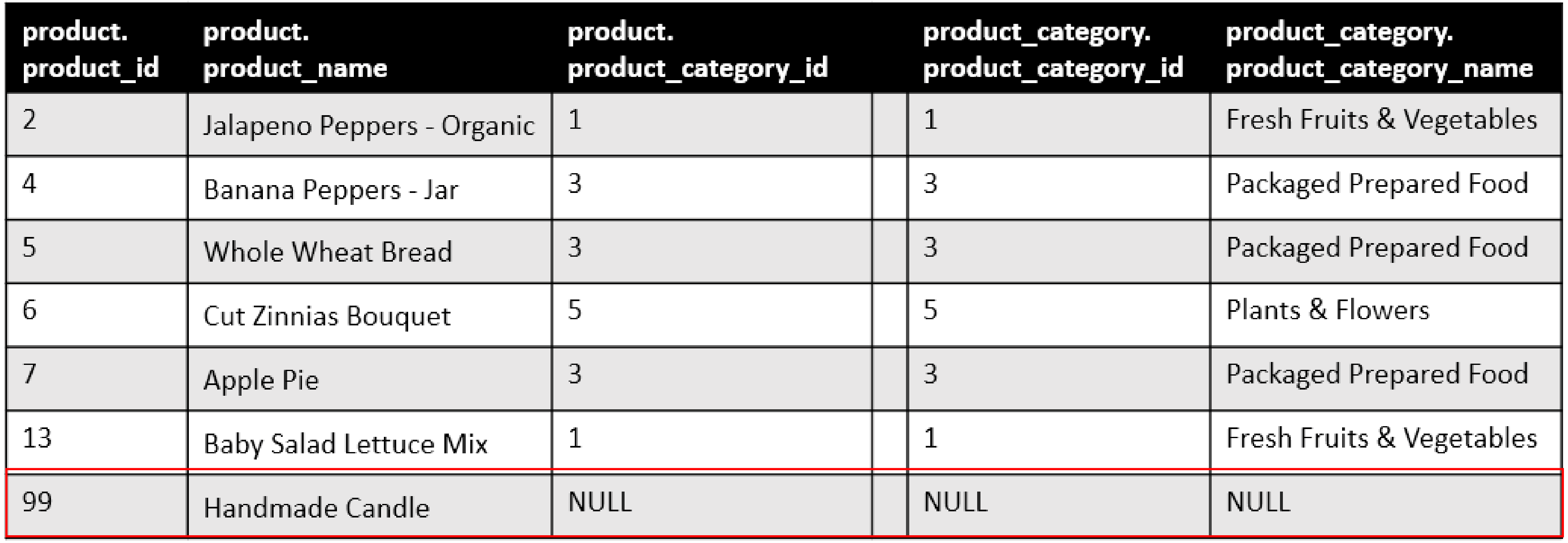
Left Join

All rows from the "left table", and only rows from the "right table" with matching values in the specified fields



SQL JOINs: LEFT JOIN

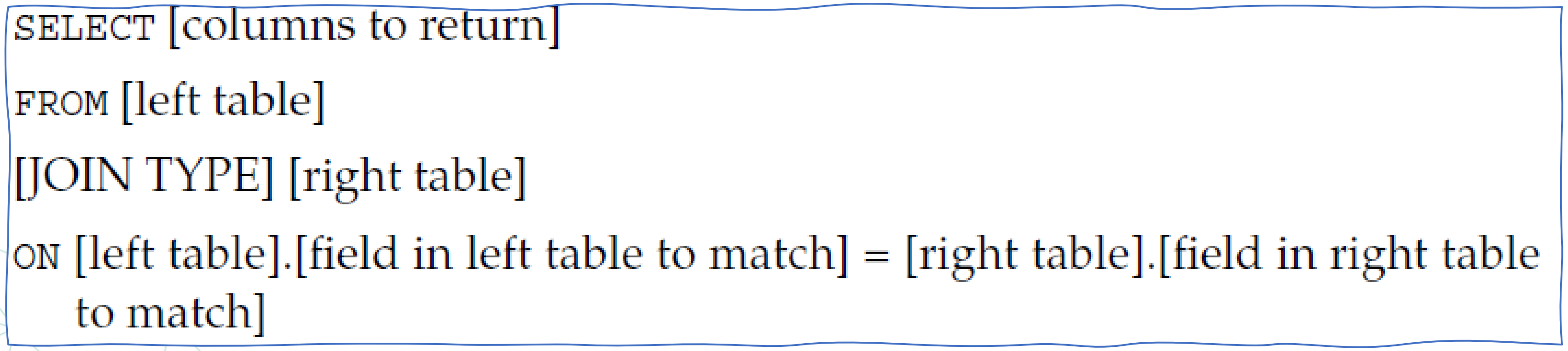
In Figure, the row with `product_id` 99 is included, but the product_category columns are NULL because there was no matching `product_category_id`. The row with `product_category_id` 6 is not included because it has no matching record in the left table, as this is a LEFT JOIN.



product. product_id	product. product_name	product. product_category_id		product_category. product_category_id	product_category. product_category_name
2	Jalapeno Peppers - Organic	1		1	Fresh Fruits & Vegetables
4	Banana Peppers - Jar	3		3	Packaged Prepared Food
5	Whole Wheat Bread	3		3	Packaged Prepared Food
6	Cut Zinnias Bouquet	5		5	Plants & Flowers
7	Apple Pie	3		3	Packaged Prepared Food
13	Baby Salad Lettuce Mix	1		1	Fresh Fruits & Vegetables
99	Handmade Candle	NULL		NULL	NULL

SQL JOINs: **LEFT** JOIN

The syntax for creating this output is:



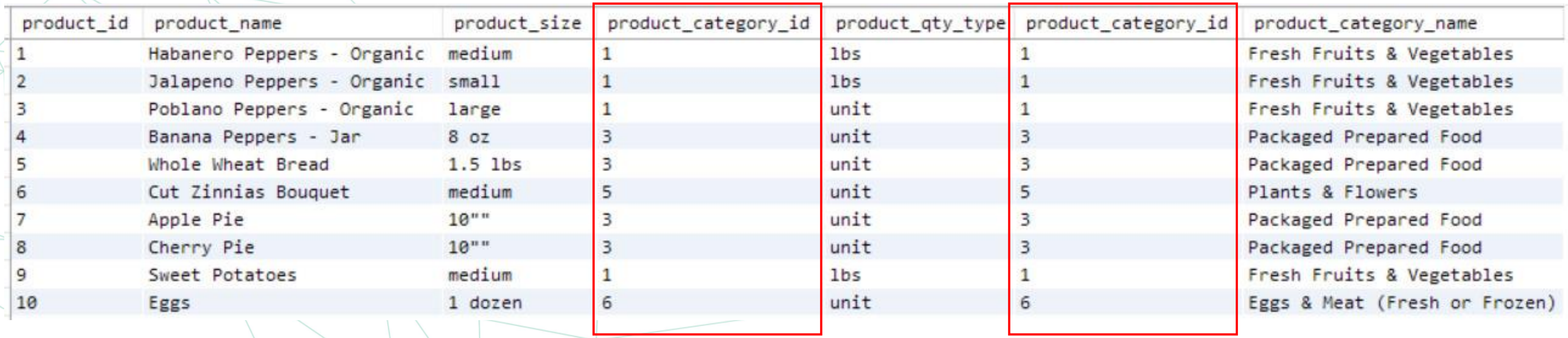
```
SELECT [columns to return]
FROM [left table]
[JOIN TYPE] [right table]
ON [left table].[field in left table to match] = [right table].[field in right table
    to match]
```

SQL JOINs: **LEFT** JOIN

This is how a query pulling all fields from both tables looks:

```
        SELECT * FROM product
              LEFT JOIN product_category
              ON product.product_category_id = product_category.
product_category_id
```

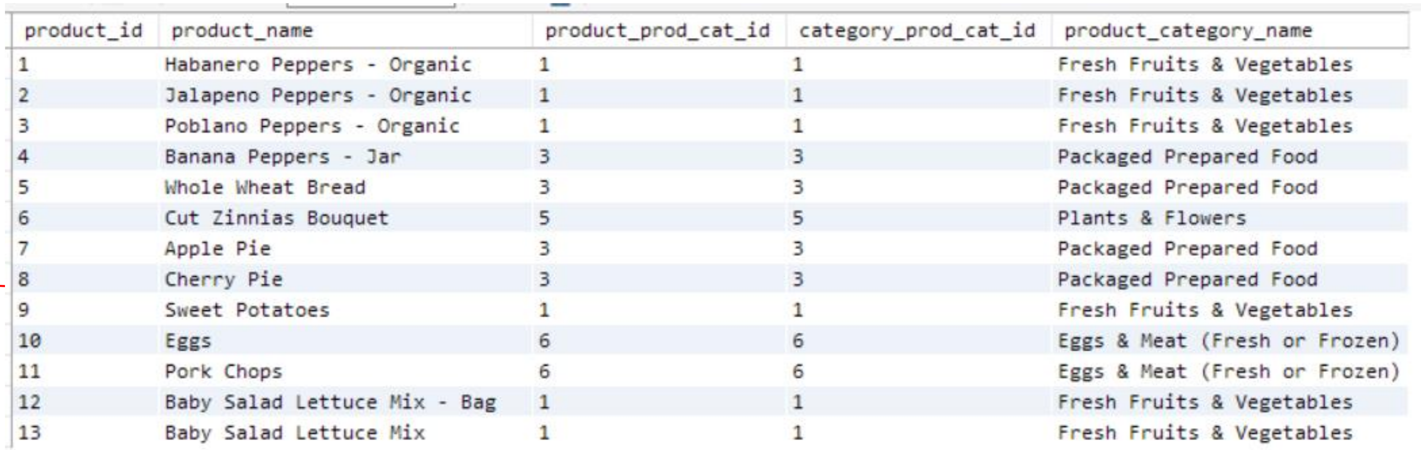
This query can be read as “Select all columns and rows from the product table, and all columns from the product_category table for rows where the product_category’s product_category_id matches a product’s product_category_id.”



product_id	product_name	product_size	product_category_id	product_qty_type	product_category_id	product_category_name
1	Habanero Peppers - Organic	medium	1	lbs	1	Fresh Fruits & Vegetables
2	Jalapeno Peppers - Organic	small	1	lbs	1	Fresh Fruits & Vegetables
3	Poblano Peppers - Organic	large	1	unit	1	Fresh Fruits & Vegetables
4	Banana Peppers - Jar	8 oz	3	unit	3	Packaged Prepared Food
5	Whole Wheat Bread	1.5 lbs	3	unit	3	Packaged Prepared Food
6	Cut Zinnias Bouquet	medium	5	unit	5	Plants & Flowers
7	Apple Pie	10""	3	unit	3	Packaged Prepared Food
8	Cherry Pie	10""	3	unit	3	Packaged Prepared Food
9	Sweet Potatoes	medium	1	lbs	1	Fresh Fruits & Vegetables
10	Eggs	1 dozen	6	unit	6	Eggs & Meat (Fresh or Frozen)

SQL JOINs: **LEFT** JOIN

Now, if we want to retrieve specific columns from the merged dataset, we have to specify which table each column is from, since it’s possible to have identically named columns in different tables. And we can alias identically named columns to differentiate them.



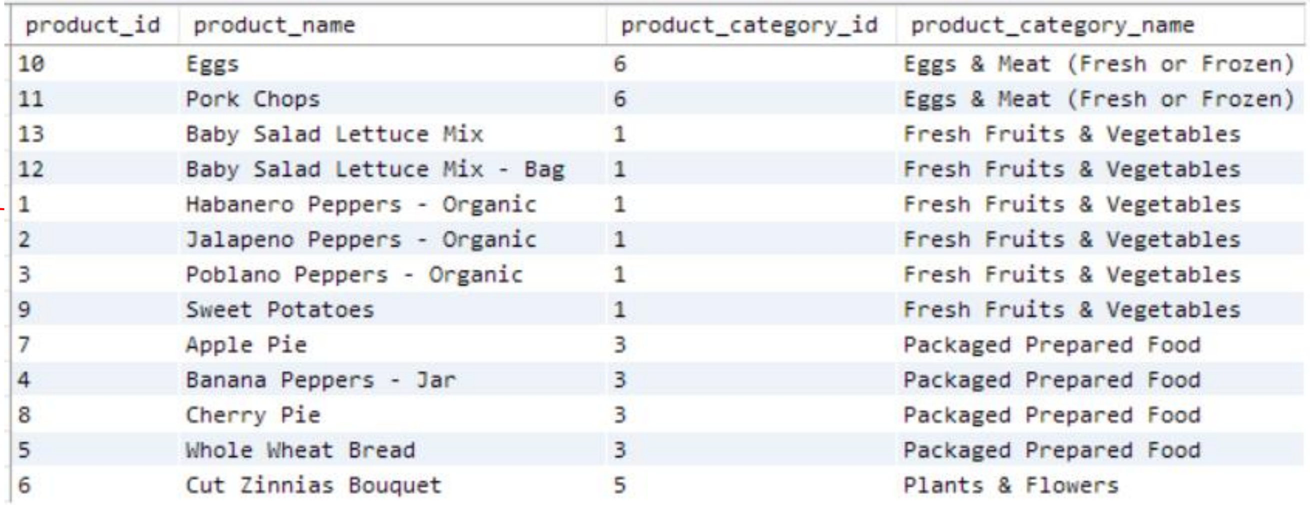
product_id	product_name	product_prod_cat_id	category_prod_cat_id	product_category_name
1	Habanero Peppers - Organic	1	1	Fresh Fruits & Vegetables
2	Jalapeno Peppers - Organic	1	1	Fresh Fruits & Vegetables
3	Poblano Peppers - Organic	1	1	Fresh Fruits & Vegetables
4	Banana Peppers - Jar	3	3	Packaged Prepared Food
5	Whole Wheat Bread	3	3	Packaged Prepared Food
6	Cut Zinnias Bouquet	5	5	Plants & Flowers
7	Apple Pie	3	3	Packaged Prepared Food
8	Cherry Pie	3	3	Packaged Prepared Food
9	Sweet Potatoes	1	1	Fresh Fruits & Vegetables
10	Eggs	6	6	Eggs & Meat (Fresh or Frozen)
11	Pork Chops	6	6	Eggs & Meat (Fresh or Frozen)
12	Baby Salad Lettuce Mix - Bag	1	1	Fresh Fruits & Vegetables
13	Baby Salad Lettuce Mix	1	1	Fresh Fruits & Vegetables

```
SELECT
    product.product_id,
    product.product_name,
    product.product_category_id AS product_prod_cat_id,
    product_category.product_category_id AS category_prod_cat_id,
    product_category.product_category_name
FROM product
    LEFT JOIN product_category
         ON product.product_category_id = product_category.
product_category_id
```

SQL JOINs: **LEFT JOIN**

Table aliasing simplifies SQL queries by allowing you to use short aliases instead of full table names, as shown with aliases “p” and “pc” in the query.

We'll also show only one `product_category_id` now that the matching between tables is confirmed.



product_id	product_name	product_category_id	product_category_name
10	Eggs	6	Eggs & Meat (Fresh or Frozen)
11	Pork Chops	6	Eggs & Meat (Fresh or Frozen)
13	Baby Salad Lettuce Mix	1	Fresh Fruits & Vegetables
12	Baby Salad Lettuce Mix - Bag	1	Fresh Fruits & Vegetables
1	Habanero Peppers - Organic	1	Fresh Fruits & Vegetables
2	Jalapeno Peppers - Organic	1	Fresh Fruits & Vegetables
3	Poblano Peppers - Organic	1	Fresh Fruits & Vegetables
9	Sweet Potatoes	1	Fresh Fruits & Vegetables
7	Apple Pie	3	Packaged Prepared Food
4	Banana Peppers - Jar	3	Packaged Prepared Food
8	Cherry Pie	3	Packaged Prepared Food
5	Whole Wheat Bread	3	Packaged Prepared Food
6	Cut Zinnias Bouquet	5	Plants & Flowers

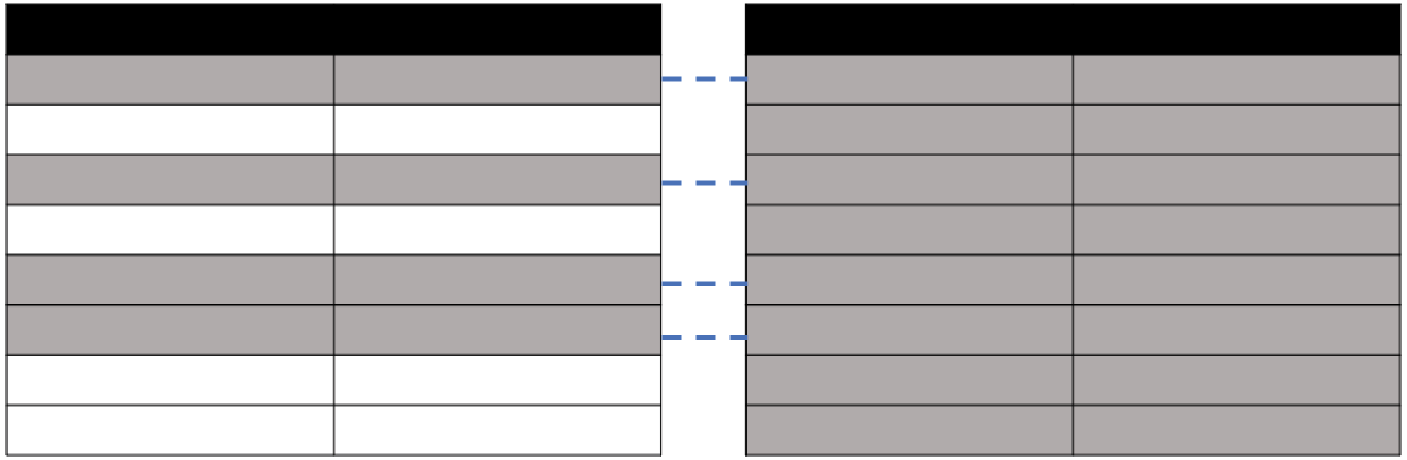
```
SELECT
    p.product_id,
    p.product_name,
    pc.product_category_id,
    pc.product_category_name
FROM product AS p
    LEFT JOIN product_category AS pc
         ON p.product_category_id = pc.product_category_id
ORDER BY pc.product_category_name, p.product_name
```

SQL JOINs: **RIGHT** JOIN

In a RIGHT JOIN, all of the rows from the “right table” are returned, along with only the matching rows from the “left table,” using the fields specified in the ON part of the query.

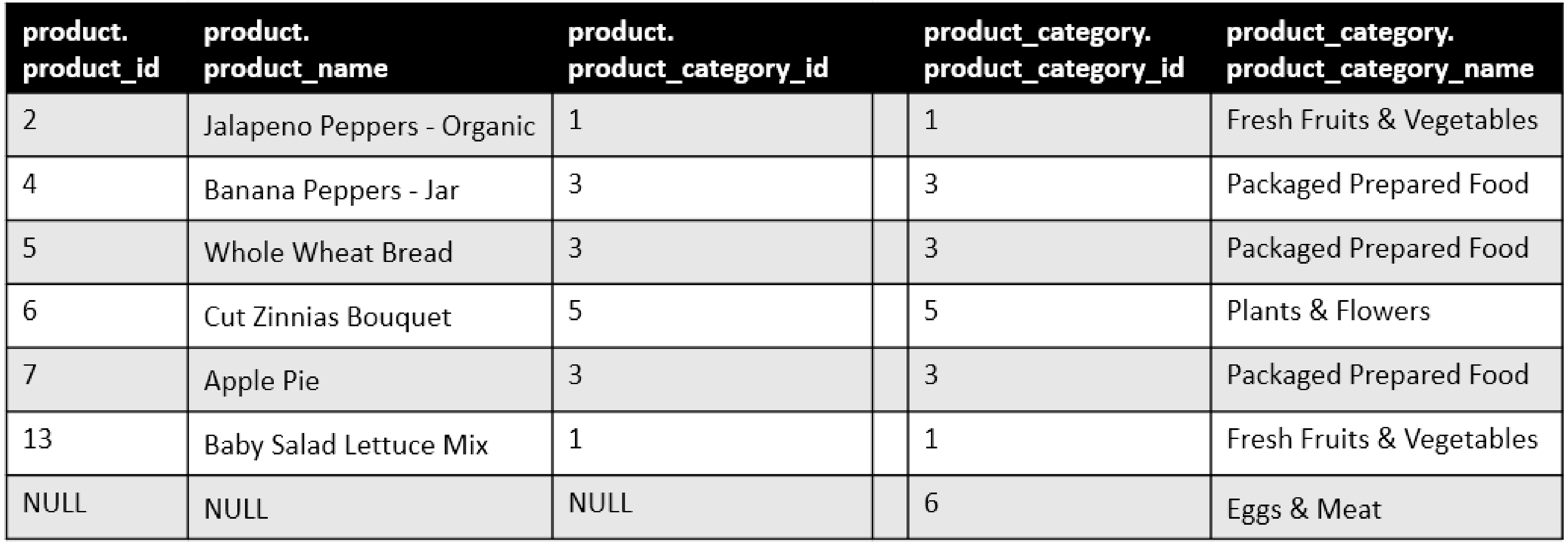
Right Join

All rows from the “right table”, and only rows from the “left table” with matching values in the specified fields



SQL JOINs: **RIGHT** JOIN

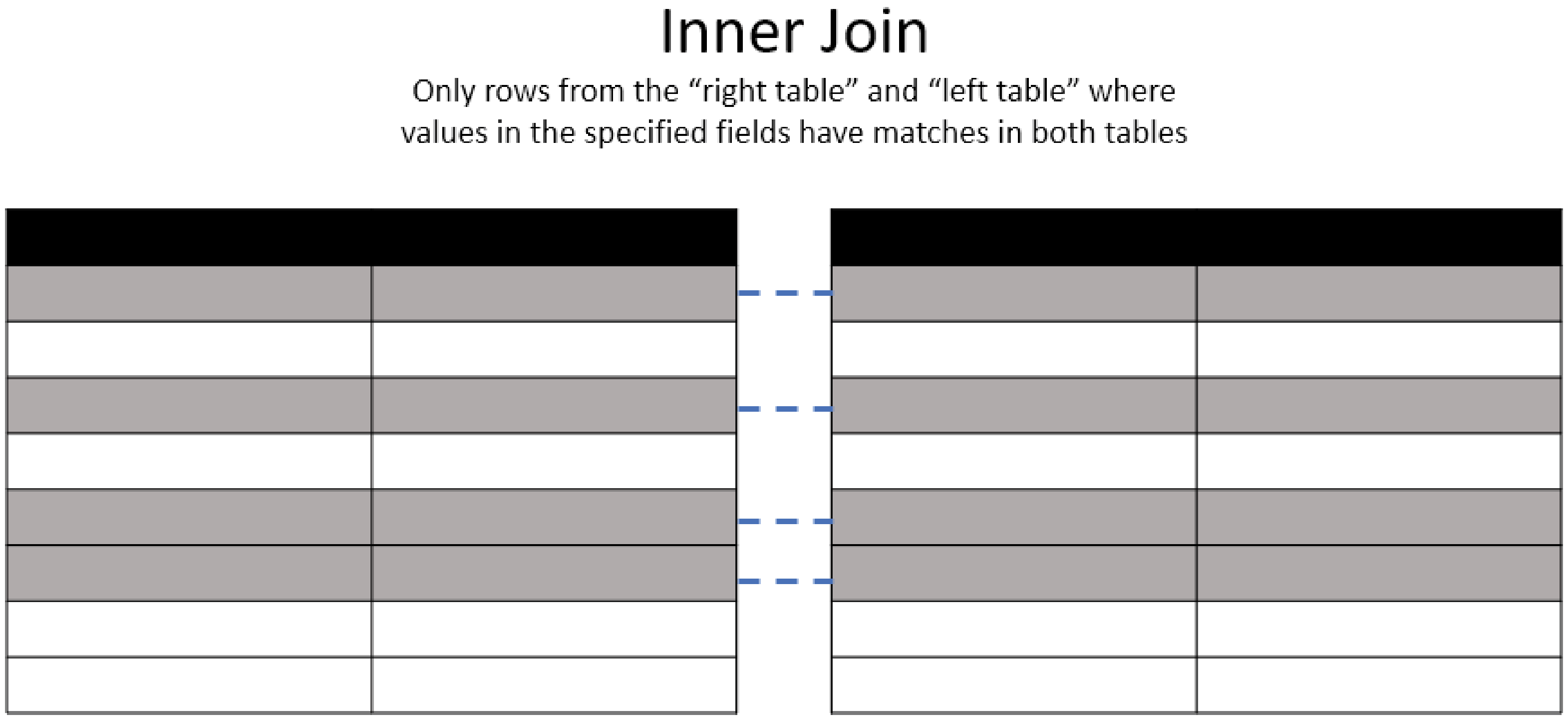
You would use a RIGHT JOIN if you wanted to list all product categories and the products in each. (And you didn’t care about products that were not put into a category, but you did care about categories that didn’t contain any products.)



product. product_id	product. product_name	product. product_category_id		product_category. product_category_id	product_category. product_category_name
2	Jalapeno Peppers - Organic	1		1	Fresh Fruits & Vegetables
4	Banana Peppers - Jar	3		3	Packaged Prepared Food
5	Whole Wheat Bread	3		3	Packaged Prepared Food
6	Cut Zinnias Bouquet	5		5	Plants & Flowers
7	Apple Pie	3		3	Packaged Prepared Food
13	Baby Salad Lettuce Mix	1		1	Fresh Fruits & Vegetables
NULL	NULL	NULL		6	Eggs & Meat

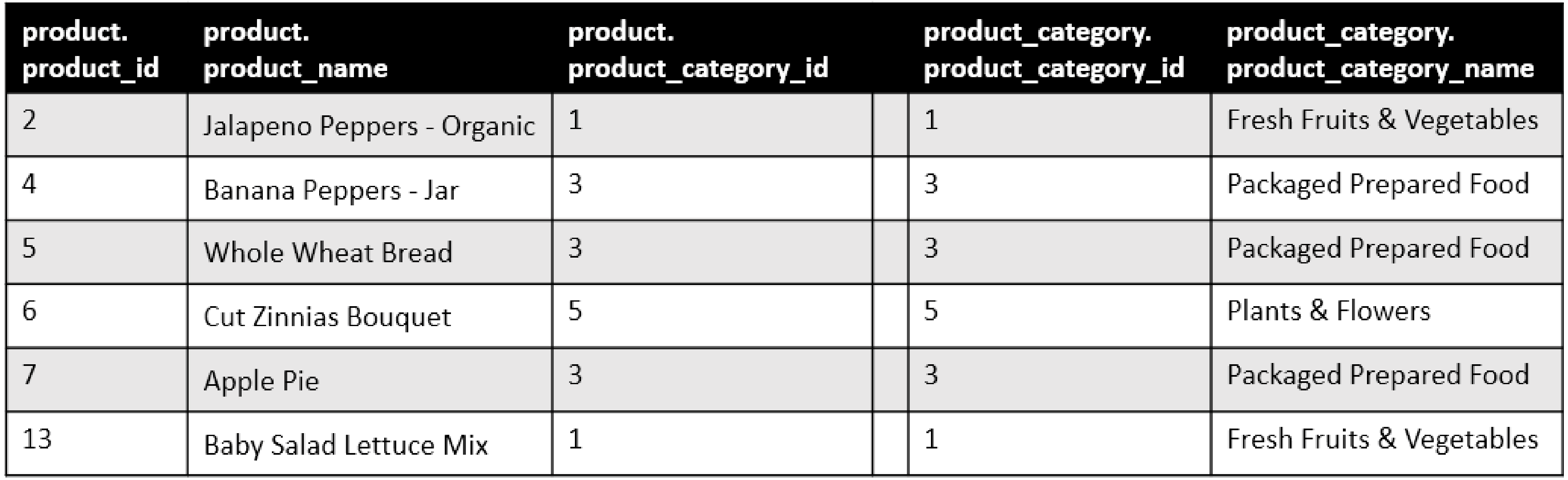
SQL JOINs: **INNER** JOIN

An INNER JOIN only returns records that have matches in both tables.


Inner Join
Only rows from the “right table” and “left table” where values in the specified fields have matches in both tables

SQL JOINs: **INNER** JOIN

In the INNER JOINed output, there is a row for every matched pair of product_category_ids, and no rows without a matching product_category_id on the other side.



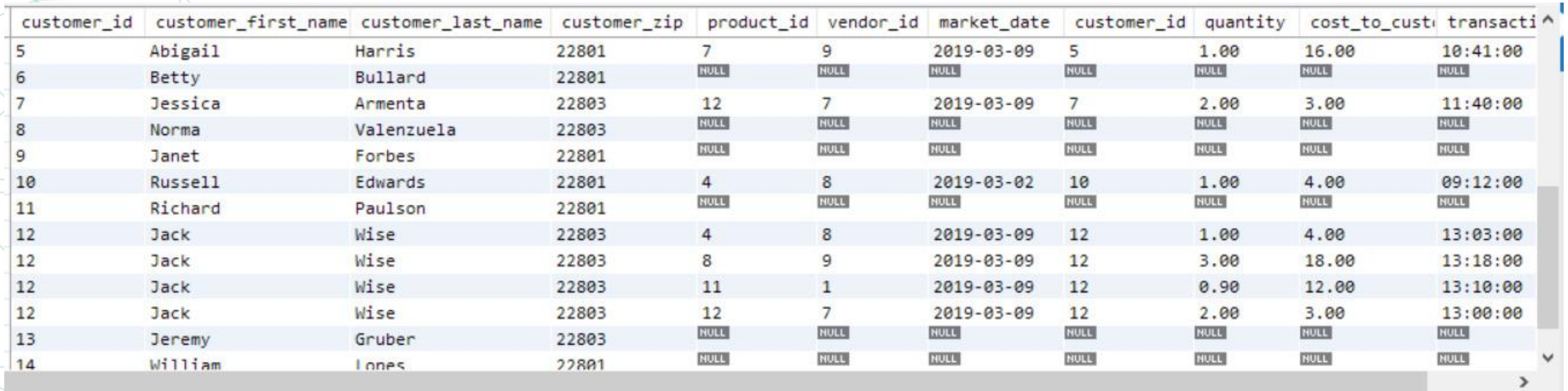
product. product_id	product. product_name	product. product_category_id		product_category. product_category_id	product_category. product_category_name
2	Jalapeno Peppers - Organic	1		1	Fresh Fruits & Vegetables
4	Banana Peppers - Jar	3		3	Packaged Prepared Food
5	Whole Wheat Bread	3		3	Packaged Prepared Food
6	Cut Zinnias Bouquet	5		5	Plants & Flowers
7	Apple Pie	3		3	Packaged Prepared Food
13	Baby Salad Lettuce Mix	1		1	Fresh Fruits & Vegetables

SQL JOINs

To practice all of these types of JOINs, let’s now look at the customer and customer_purchase tables in the Farmer’s Market database. Again, this is a one-to-many type relationship.

We do a LEFT JOIN, using the following query:

```
SELECT *
FROM customer AS c
LEFT JOIN customer_purchases AS cp
    ON c.customer_id = cp.customer_id
```



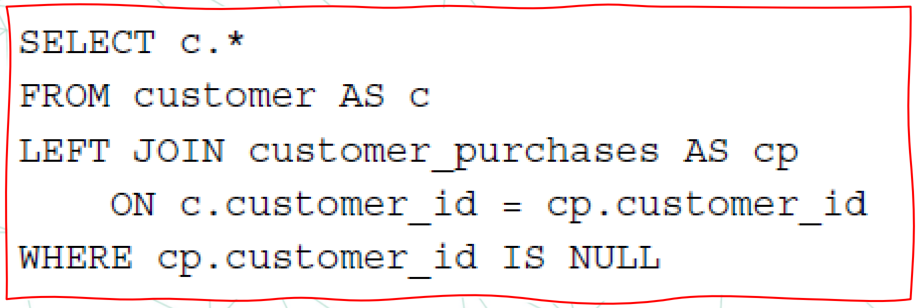
customer_id	customer_first_name	customer_last_name	customer_zip	product_id	vendor_id	market_date	customer_id	quantity	cost_to_cust	transacti
5	Abigail	Harris	22801	7	9	2019-03-09	5	1.00	16.00	10:41:00
6	Betty	Bullard	22801	NULL	NULL	NULL	NULL	NULL	NULL	NULL
7	Jessica	Armenta	22803	12	7	2019-03-09	7	2.00	3.00	11:40:00
8	Norma	Valenzuela	22803	NULL	NULL	NULL	NULL	NULL	NULL	NULL
9	Janet	Forbes	22801	NULL	NULL	NULL	NULL	NULL	NULL	NULL
10	Russell	Edwards	22801	4	8	2019-03-02	10	1.00	4.00	09:12:00
11	Richard	Paulson	22801	NULL	NULL	NULL	NULL	NULL	NULL	NULL
12	Jack	Wise	22803	4	8	2019-03-09	12	1.00	4.00	13:03:00
12	Jack	Wise	22803	8	9	2019-03-09	12	3.00	18.00	13:18:00
12	Jack	Wise	22803	11	1	2019-03-09	12	0.90	12.00	13:10:00
12	Jack	Wise	22803	12	7	2019-03-09	12	2.00	3.00	13:00:00
13	Jeremy	Gruber	22803	NULL	NULL	NULL	NULL	NULL	NULL	NULL
14	William	Lopes	22801	NULL	NULL	NULL	NULL	NULL	NULL	NULL

SQL JOINs

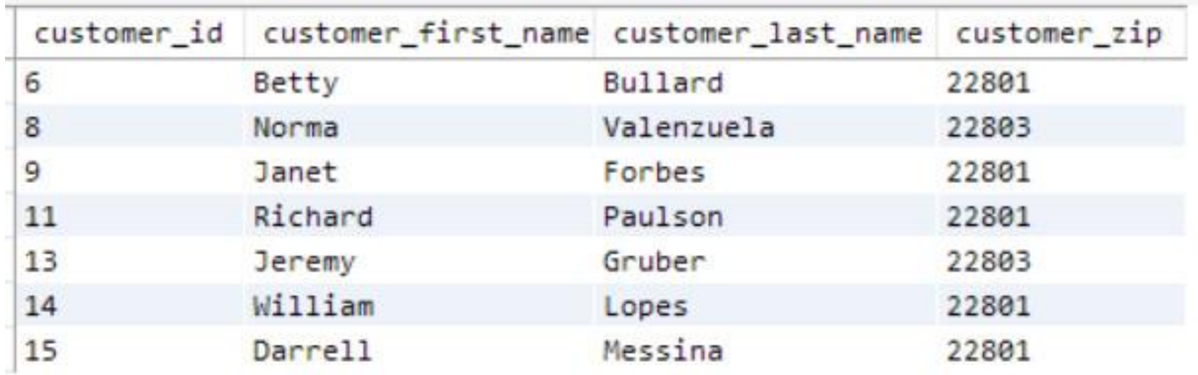
Unlike the product-product_category relationship, some customers may not have made any purchases. These customers, who signed up for the loyalty card but haven't bought anything yet, will appear in the output due to the LEFT JOIN.

The list includes all customers with their purchases if available. Customers with multiple purchases will appear multiple times, while those with no purchases will show NULLs for fields from the customer_purchases table.

We can use the WHERE clause to filter the list to only customers with no purchases, if we'd like:



```
SELECT c.*
FROM customer AS c
LEFT JOIN customer_purchases AS cp
    ON c.customer_id = cp.customer_id
WHERE cp.customer_id IS NULL
```



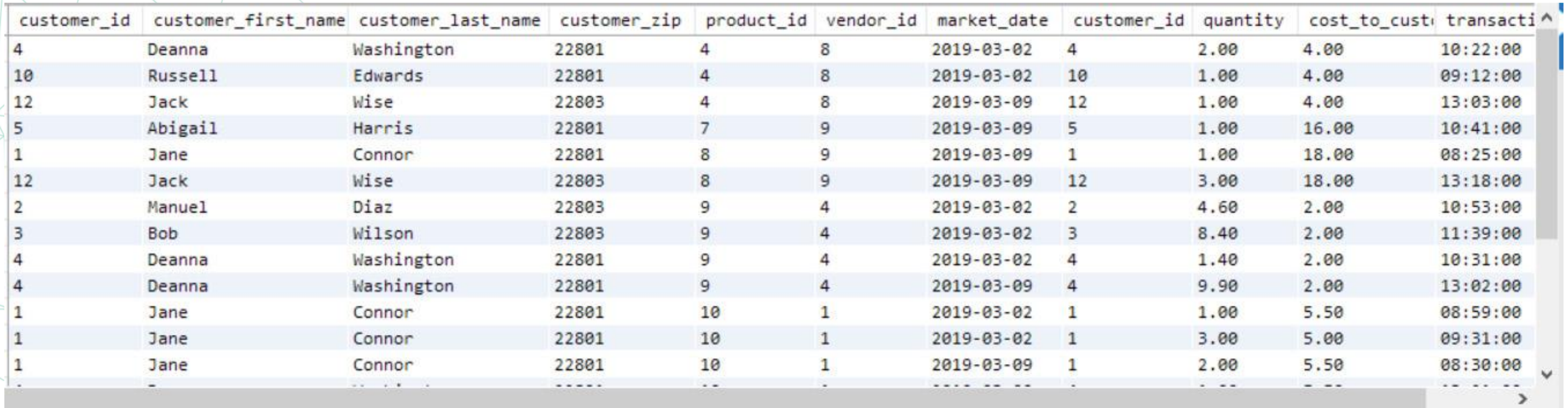
customer_id	customer_first_name	customer_last_name	customer_zip
6	Betty	Bullard	22801
8	Norma	Valenzuela	22803
9	Janet	Forbes	22801
11	Richard	Paulson	22801
13	Jeremy	Gruber	22803
14	William	Lopes	22801
15	Darrell	Messina	22801

SQL JOINs

To list all purchases and their associated customers, we would use a RIGHT JOIN. This will pull all records from the customer_purchases table and only include customers from the customer table who have made a purchase.

```
SELECT *
FROM customer AS c
RIGHT JOIN customer_purchases AS cp
    ON c.customer_id = cp.customer_id
```

there are no rows returned with NULL values in the customer table columns, because there is no such thing as a purchase without a customer_id.



customer_id	customer_first_name	customer_last_name	customer_zip	product_id	vendor_id	market_date	customer_id	quantity	cost_to_cust	transacti
4	Deanna	Washington	22801	4	8	2019-03-02	4	2.00	4.00	10:22:00
10	Russell	Edwards	22801	4	8	2019-03-02	10	1.00	4.00	09:12:00
12	Jack	Wise	22803	4	8	2019-03-09	12	1.00	4.00	13:03:00
5	Abigail	Harris	22801	7	9	2019-03-09	5	1.00	16.00	10:41:00
1	Jane	Connor	22801	8	9	2019-03-09	1	1.00	18.00	08:25:00
12	Jack	Wise	22803	8	9	2019-03-09	12	3.00	18.00	13:18:00
2	Manuel	Diaz	22803	9	4	2019-03-02	2	4.60	2.00	10:53:00
3	Bob	Wilson	22803	9	4	2019-03-02	3	8.40	2.00	11:39:00
4	Deanna	Washington	22801	9	4	2019-03-02	4	1.40	2.00	10:31:00
4	Deanna	Washington	22801	9	4	2019-03-09	4	9.90	2.00	13:02:00
1	Jane	Connor	22801	10	1	2019-03-02	1	1.00	5.50	08:59:00
1	Jane	Connor	22801	10	1	2019-03-02	1	3.00	5.00	09:31:00
1	Jane	Connor	22801	10	1	2019-03-09	1	2.00	5.50	08:30:00

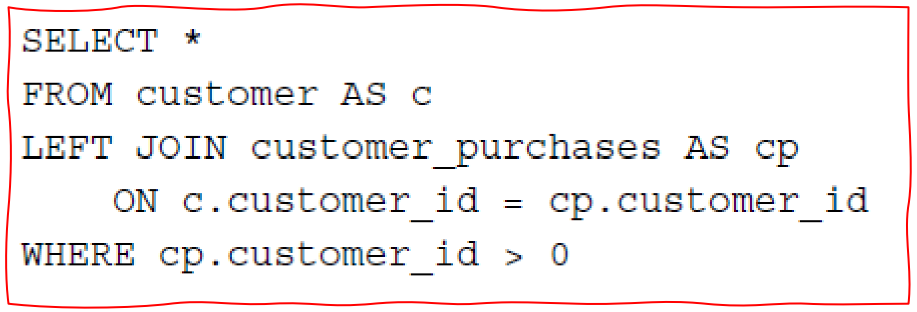
SQL JOINs

If you only want records from each table that have matches in both tables, use an INNER JOIN.

Using these customer and customer_purchases tables, an INNER JOIN happens to return the same results as the RIGHT JOIN, because there aren't any records on the "right side" of the join without matches on the "left side"—every purchase is associated with a customer.

SQL JOINs: Pitfall when Filtering Joined Data

How do you think the output of the following query will differ from the original LEFT JOIN query without the added WHERE clause?



```
SELECT *
FROM customer AS c
LEFT JOIN customer_purchases AS cp
    ON c.customer_id = cp.customer_id
WHERE cp.customer_id > 0
```

Although `customer_id` values are all integers above 0, adding this WHERE clause filters based on the `customer_id` in the `customer_purchases` table (alias `cp`). This removes customers without purchases, making the query behave like an INNER JOIN by excluding records with NULL values in the right-side table columns.

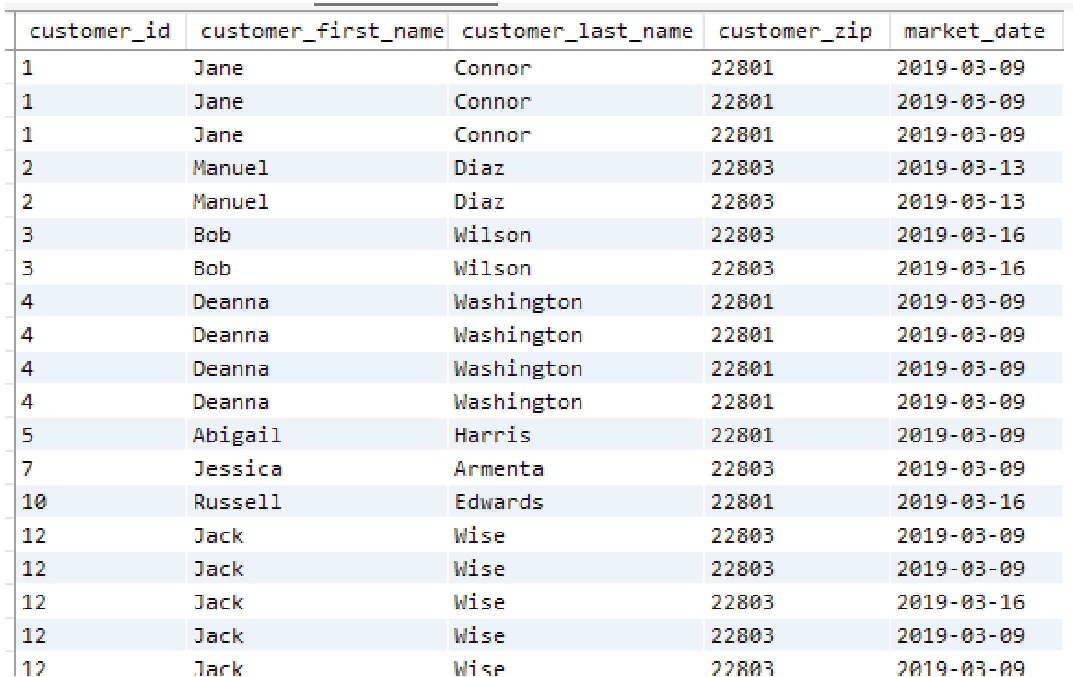
When using a LEFT JOIN to return all rows from the left table, avoid filtering on fields from the right table unless you allow NULLs. Otherwise, you might filter out rows you intended to keep.

SQL JOINs: Pitfall when Filtering Joined Data

Let’s say we want to write a query that returns a list of all customers who did not make a purchase at the March 2, 2019, farmer’s market.

We will use a LEFT JOIN, since we want to include the customers who have never made a purchase at any farmer’s market, so wouldn’t have any records in the customer_purchases table:

```
SELECT c.*, cp.market_date
FROM customer AS c
LEFT JOIN customer_purchases AS cp
    ON c.customer_id = cp.customer_id
WHERE cp.market_date <> '2019-03-02'
```



customer_id	customer_first_name	customer_last_name	customer_zip	market_date
1	Jane	Connor	22801	2019-03-09
1	Jane	Connor	22801	2019-03-09
1	Jane	Connor	22801	2019-03-09
2	Manuel	Diaz	22803	2019-03-13
2	Manuel	Diaz	22803	2019-03-13
3	Bob	Wilson	22803	2019-03-16
3	Bob	Wilson	22803	2019-03-16
4	Deanna	Washington	22801	2019-03-09
4	Deanna	Washington	22801	2019-03-09
4	Deanna	Washington	22801	2019-03-09
4	Deanna	Washington	22801	2019-03-09
5	Abigail	Harris	22801	2019-03-09
7	Jessica	Armenta	22803	2019-03-09
10	Russell	Edwards	22801	2019-03-16
12	Jack	Wise	22803	2019-03-09
12	Jack	Wise	22803	2019-03-09
12	Jack	Wise	22803	2019-03-16
12	Jack	Wise	22803	2019-03-09
12	Jack	Wise	22803	2019-03-09

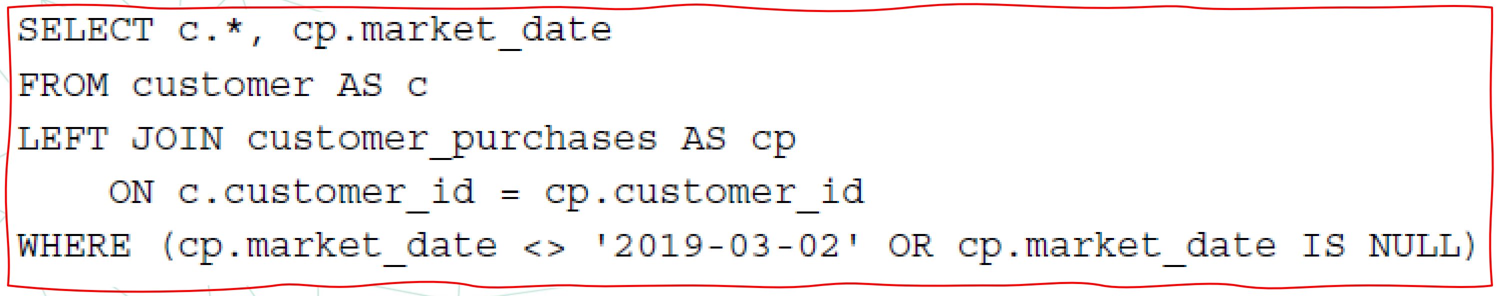
SQL JOINs: Pitfall when Filtering Joined Data

There are multiple problems with this output:

1. The problem is that we’re missing customers who never made a purchase because we filtered on `market_date` from the `customer_purchases` table (the right side of the JOIN).

   Since SQL doesn’t compare NULL values to TRUE, this filter excludes customers without purchases, although we need it to exclude those who made a purchase on that day.

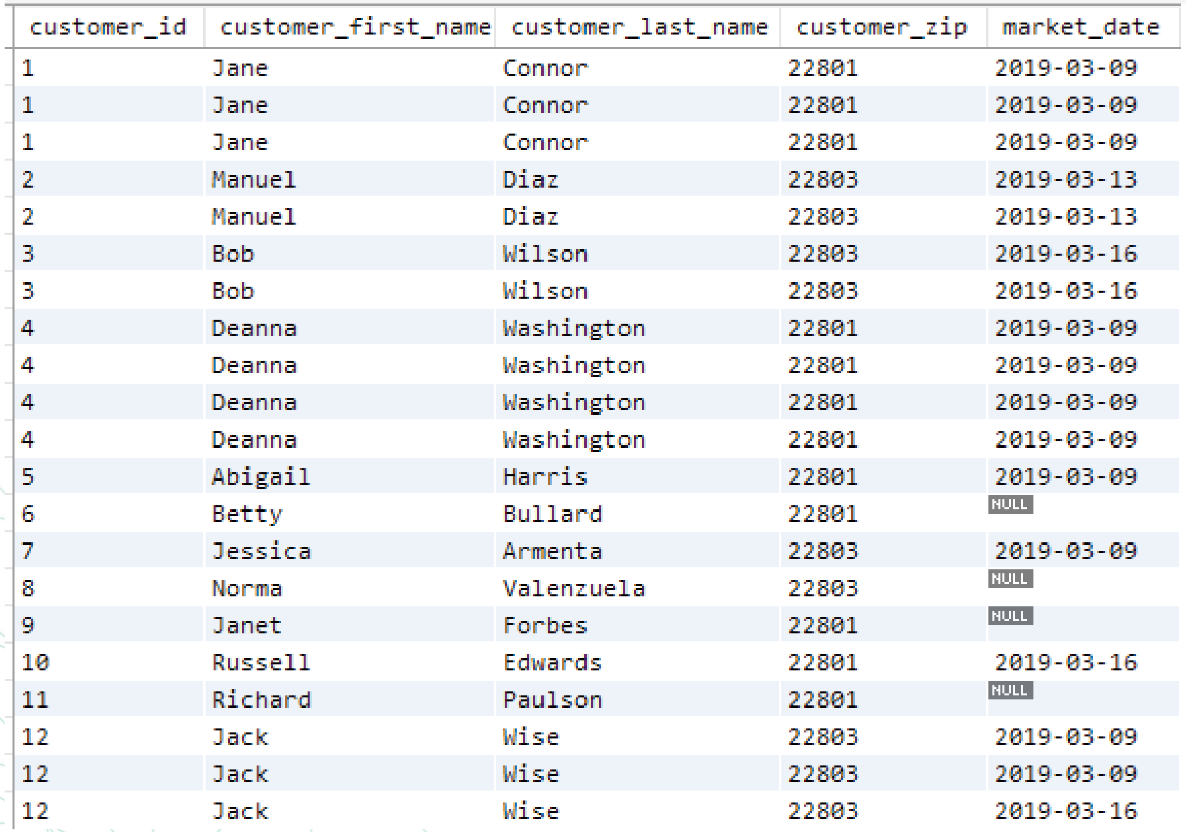
Solution: To filter results using a field from the right table while still returning records from the left table, adjust the WHERE clause to allow NULL values.



```
SELECT c.*, cp.market_date
FROM customer AS c
LEFT JOIN customer_purchases AS cp
    ON c.customer_id = cp.customer_id
WHERE (cp.market_date <> '2019-03-02' OR cp.market_date IS NULL)
```

SQL JOINs: Pitfall when Filtering Joined Data

Now we see customers without purchases:



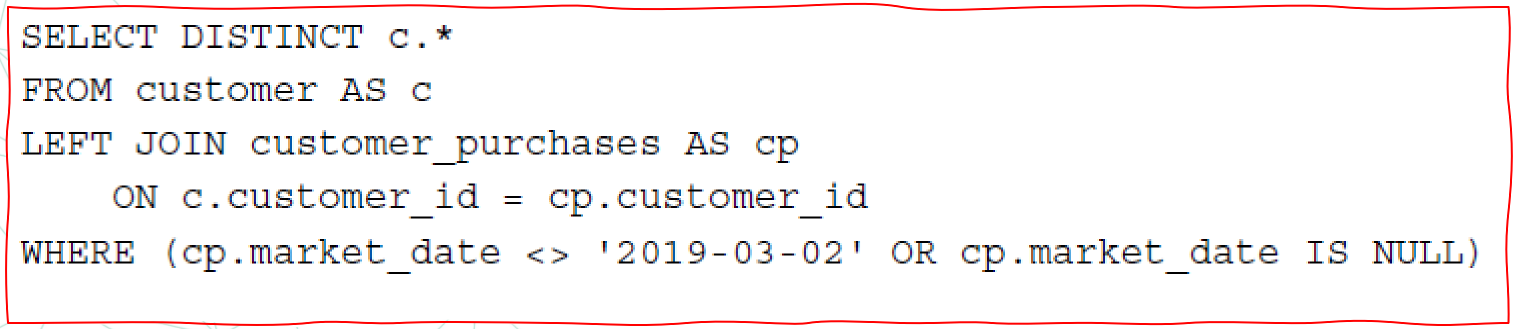
customer_id	customer_first_name	customer_last_name	customer_zip	market_date
1	Jane	Connor	22801	2019-03-09
1	Jane	Connor	22801	2019-03-09
1	Jane	Connor	22801	2019-03-09
2	Manuel	Diaz	22803	2019-03-13
2	Manuel	Diaz	22803	2019-03-13
3	Bob	Wilson	22803	2019-03-16
3	Bob	Wilson	22803	2019-03-16
4	Deanna	Washington	22801	2019-03-09
4	Deanna	Washington	22801	2019-03-09
4	Deanna	Washington	22801	2019-03-09
4	Deanna	Washington	22801	2019-03-09
5	Abigail	Harris	22801	2019-03-09
6	Betty	Bullard	22801	NULL
7	Jessica	Armenta	22803	2019-03-09
8	Norma	Valenzuela	22803	NULL
9	Janet	Forbes	22801	NULL
10	Russell	Edwards	22801	2019-03-16
11	Richard	Paulson	22801	NULL
12	Jack	Wise	22803	2019-03-09
12	Jack	Wise	22803	2019-03-09
12	Jack	Wise	22803	2019-03-16

SQL JOINs: Pitfall when Filtering Joined Data

There are multiple problems with this output:

2. The output shows one row per customer per item purchased, rather than just a list of customers, because the `customer_purchases` table includes a record for each purchased item.

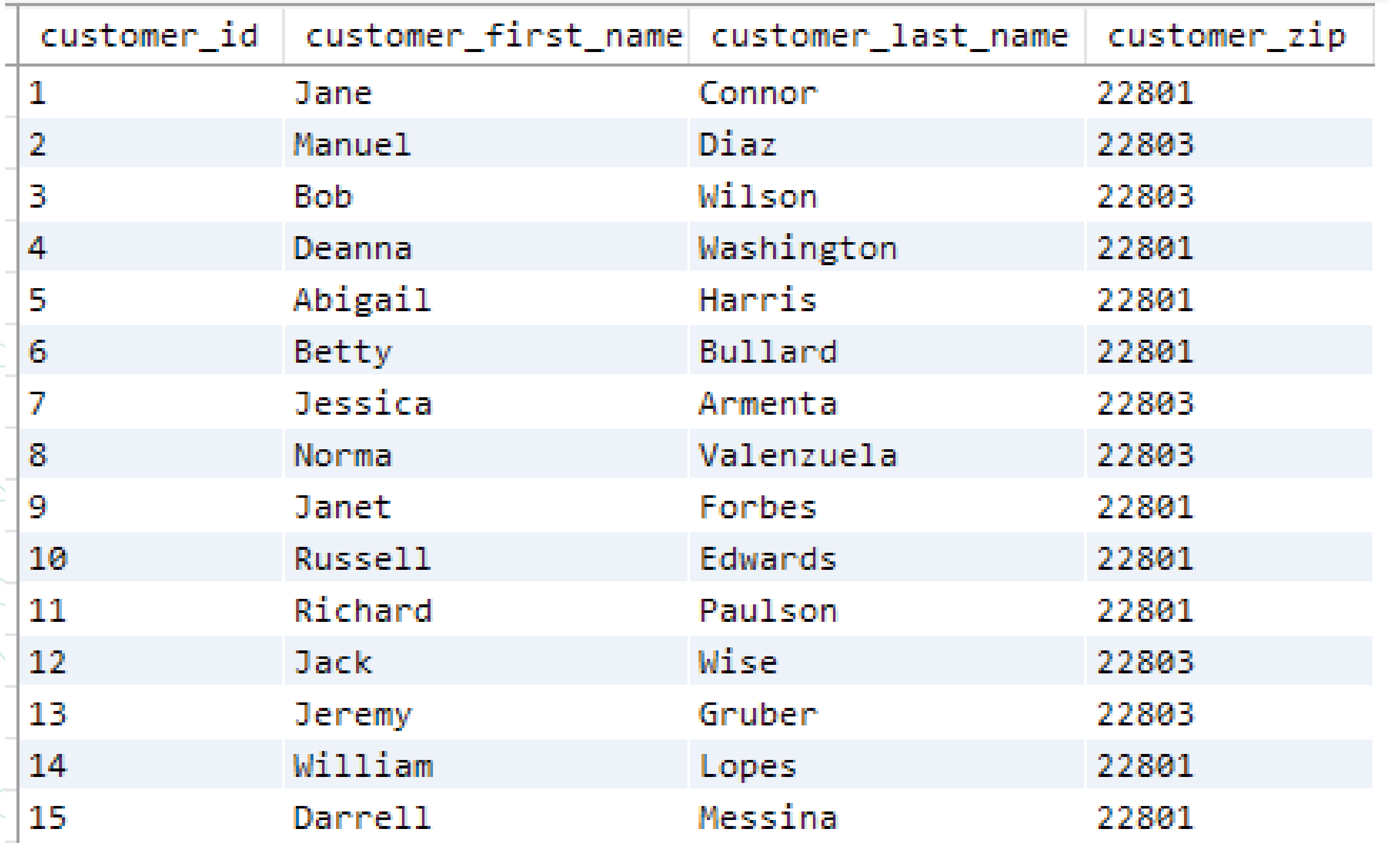
Solution: Removing the `market_date` field from the `customer_purchases` table and using the `DISTINCT` keyword to display only unique results.



```
SELECT DISTINCT c.*
FROM customer AS c
LEFT JOIN customer_purchases AS cp
    ON c.customer_id = cp.customer_id
WHERE (cp.market_date <> '2019-03-02' OR cp.market_date IS NULL)
```

SQL JOINs: Pitfall when Filtering Joined Data

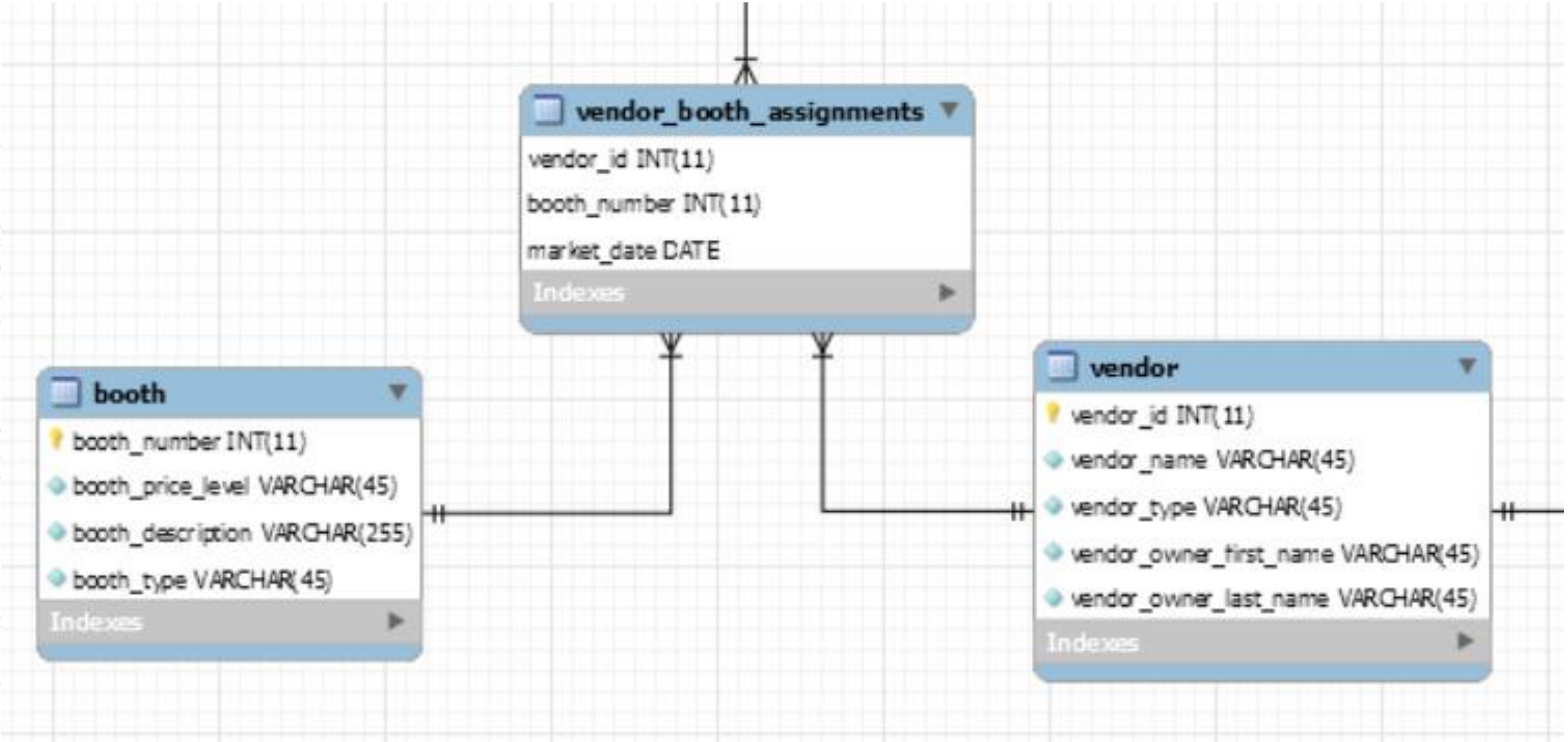
This results:



customer_id	customer_first_name	customer_last_name	customer_zip
1	Jane	Connor	22801
2	Manuel	Diaz	22803
3	Bob	Wilson	22803
4	Deanna	Washington	22801
5	Abigail	Harris	22801
6	Betty	Bullard	22801
7	Jessica	Armenta	22803
8	Norma	Valenzuela	22803
9	Janet	Forbes	22801
10	Russell	Edwards	22801
11	Richard	Paulson	22801
12	Jack	Wise	22803
13	Jeremy	Gruber	22803
14	William	Lopes	22801
15	Darrell	Messina	22801

SQL JOINs: More than Two Tables

To get details about all farmer's market booths and vendor assignments for every market date, we need to join the three tables to create a merged dataset:



vendor_booth_assignments

vendor_id INT(11)
booth_number INT(11)
market_date DATE

Indexes

booth

booth_number INT(11)
booth_price_level VARCHAR(45)
booth_description VARCHAR(255)
booth_type VARCHAR(45)

Indexes

vendor

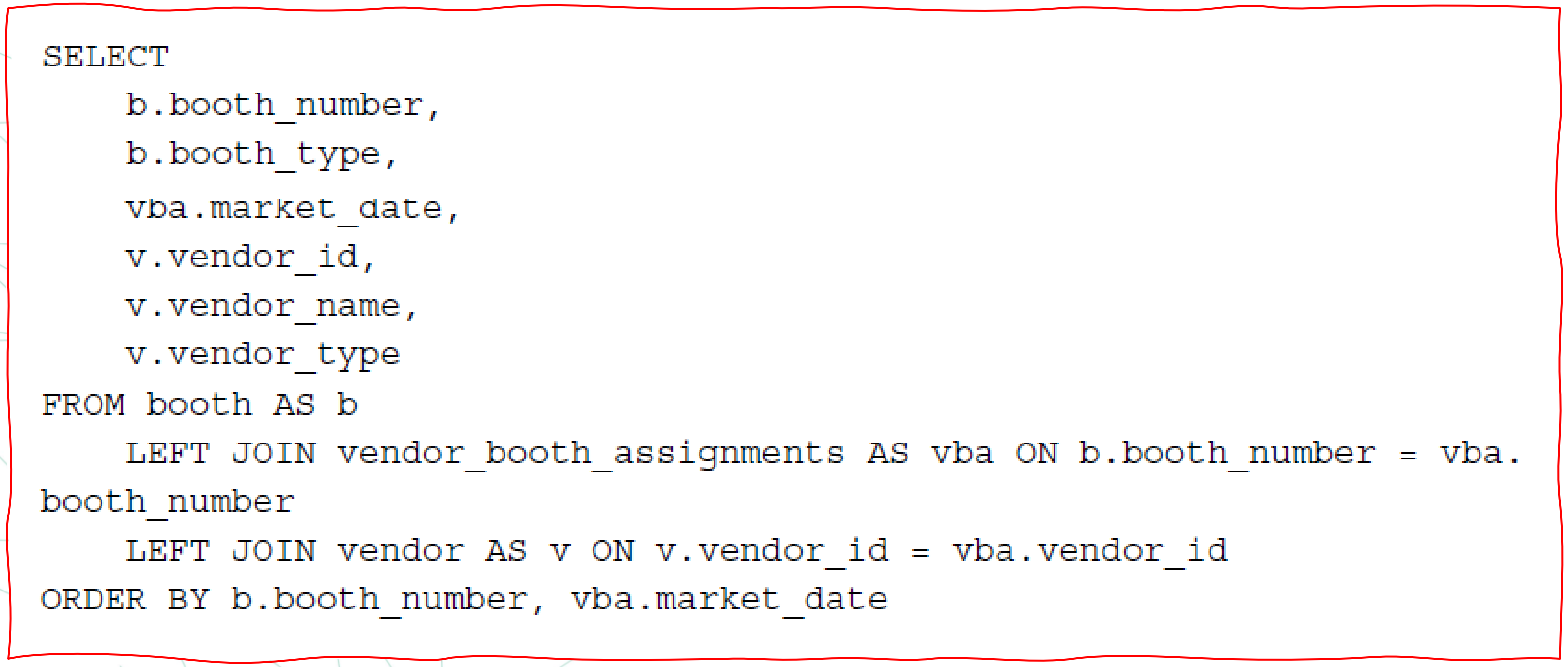
vendor_id INT(11)
vendor_name VARCHAR(45)
vendor_type VARCHAR(45)
vendor_owner_first_name VARCHAR(45)
vendor_owner_last_name VARCHAR(45)

Indexes

SQL JOINs: More than Two Tables

What JOINs can we use to include all booths, even those not assigned to a vendor, and all vendors assigned to booths?

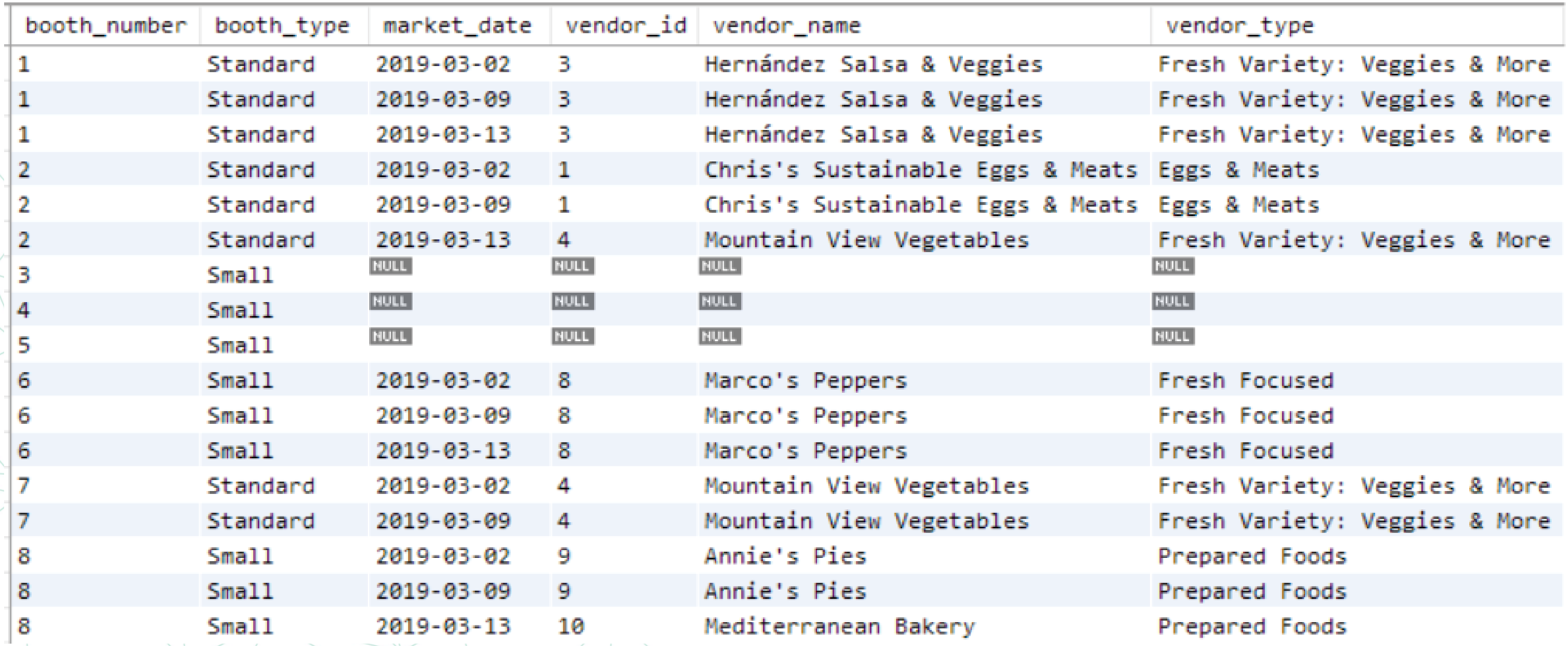
We can LEFT JOIN the vendor_booth_assignments to booth, therefore including all of the booths, and LEFT JOIN vendor to vendor_booth_assignments in the results.



```
SELECT
    b.booth_number,
    b.booth_type,
    vba.market_date,
    v.vendor_id,
    v.vendor_name,
    v.vendor_type
FROM booth AS b
    LEFT JOIN vendor_booth_assignments AS vba ON b.booth_number = vba.
booth_number
    LEFT JOIN vendor AS v ON v.vendor_id = vba.vendor_id
ORDER BY b.booth_number, vba.market_date
```

SQL JOINs: More than Two Tables

This results:



booth_number	booth_type	market_date	vendor_id	vendor_name	vendor_type
1	Standard	2019-03-02	3	Hernández Salsa & Veggies	Fresh Variety: Veggies & More
1	Standard	2019-03-09	3	Hernández Salsa & Veggies	Fresh Variety: Veggies & More
1	Standard	2019-03-13	3	Hernández Salsa & Veggies	Fresh Variety: Veggies & More
2	Standard	2019-03-02	1	Chris's Sustainable Eggs & Meats	Eggs & Meats
2	Standard	2019-03-09	1	Chris's Sustainable Eggs & Meats	Eggs & Meats
2	Standard	2019-03-13	4	Mountain View Vegetables	Fresh Variety: Veggies & More
3	Small	NULL	NULL	NULL	NULL
4	Small	NULL	NULL	NULL	NULL
5	Small	NULL	NULL	NULL	NULL
6	Small	2019-03-02	8	Marco's Peppers	Fresh Focused
6	Small	2019-03-09	8	Marco's Peppers	Fresh Focused
6	Small	2019-03-13	8	Marco's Peppers	Fresh Focused
7	Standard	2019-03-02	4	Mountain View Vegetables	Fresh Variety: Veggies & More
7	Standard	2019-03-09	4	Mountain View Vegetables	Fresh Variety: Veggies & More
8	Small	2019-03-02	9	Annie's Pies	Prepared Foods
8	Small	2019-03-09	9	Annie's Pies	Prepared Foods
8	Small	2019-03-13	10	Mediterranean Bakery	Prepared Foods

SQL JOINs: More than Two Tables

You can think of the second JOIN as being merged into the result of the first JOIN.

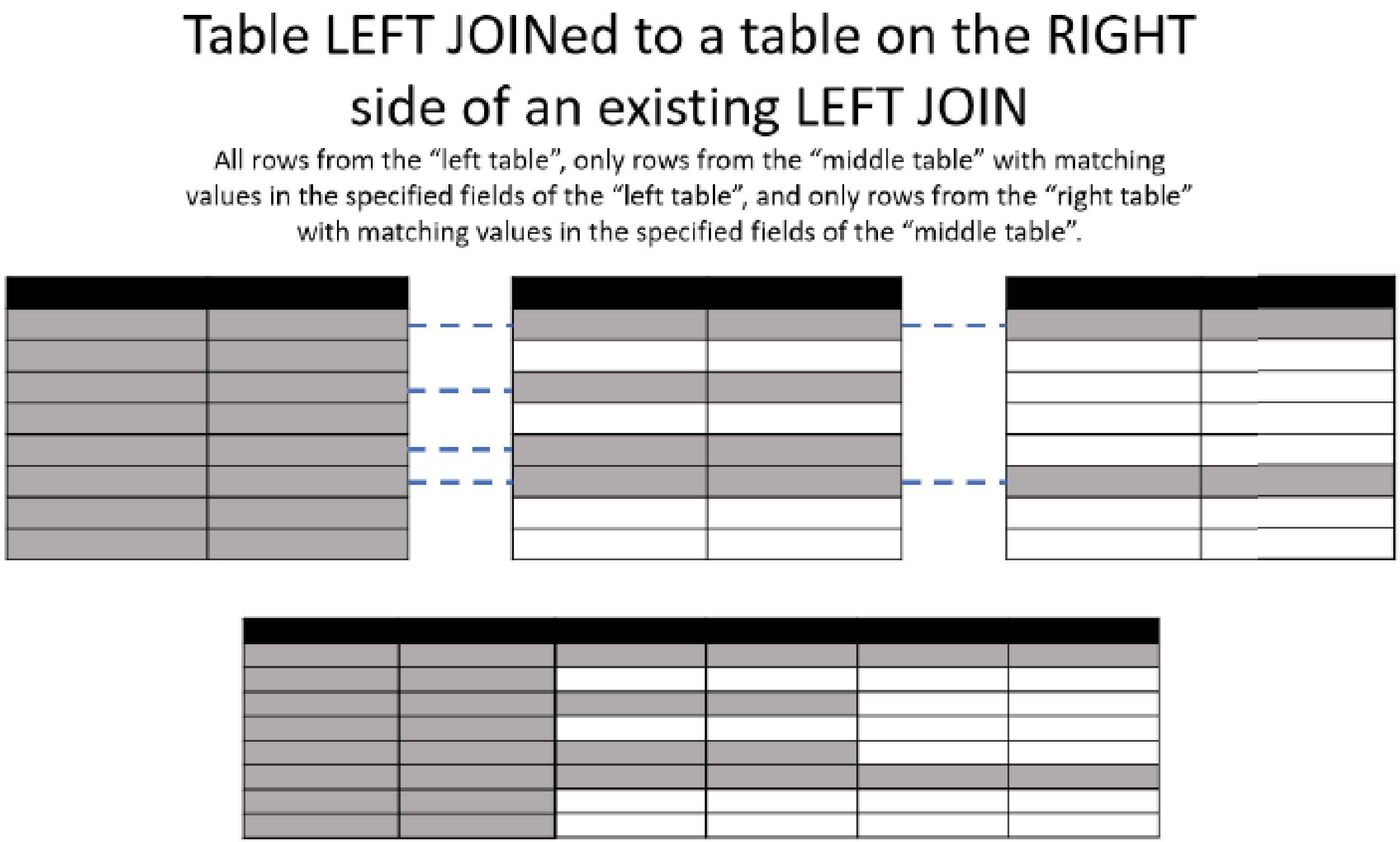
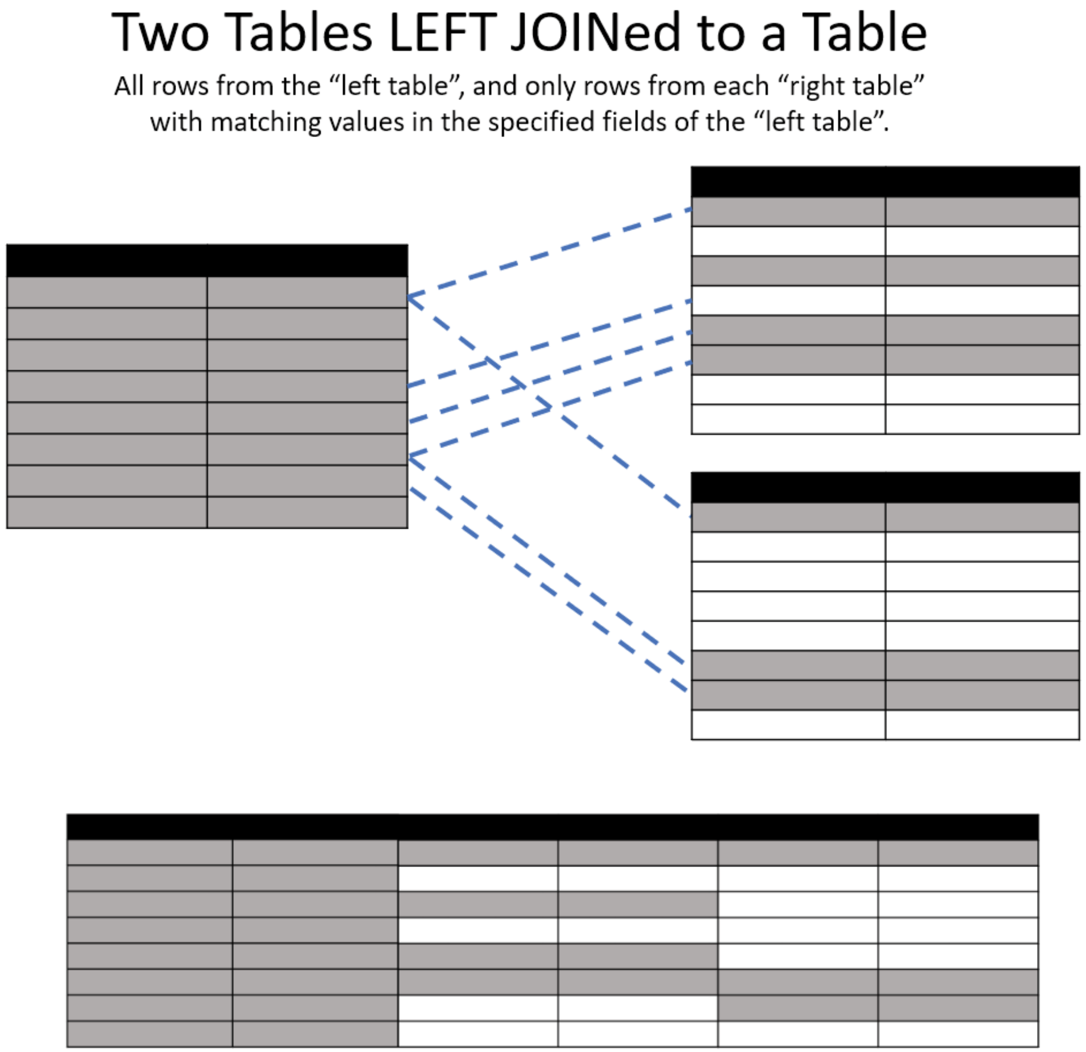


Table LEFT JOINed to a table on the RIGHT side of an existing LEFT JOIN

All rows from the "left table", only rows from the "middle table" with matching values in the specified fields of the "left table", and only rows from the "right table" with matching values in the specified fields of the "middle table".

SQL JOINs: More than Two Tables

If the third table joined the first table using a common field (though not possible with the Farmer's Market database since no other tables join the booth table), the arrangement would look like this diagram.



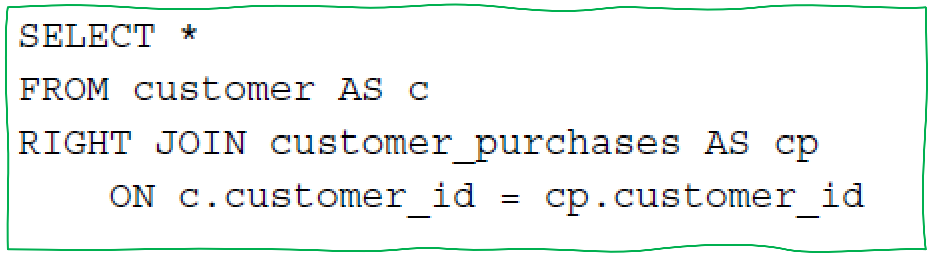
Two Tables LEFT JOINed to a Table

All rows from the "left table", and only rows from each "right table" with matching values in the specified fields of the "left table".

Exercise

1. Write a query that INNER JOINs the vendor table to the vendor_booth_assignments table on the vendor_id field they both have in common, and sorts the result by vendor_name, then market_date.

2. Is it possible to write a query that produces an output identical to the output of the following query, but using a LEFT JOIN instead of a RIGHT JOIN?

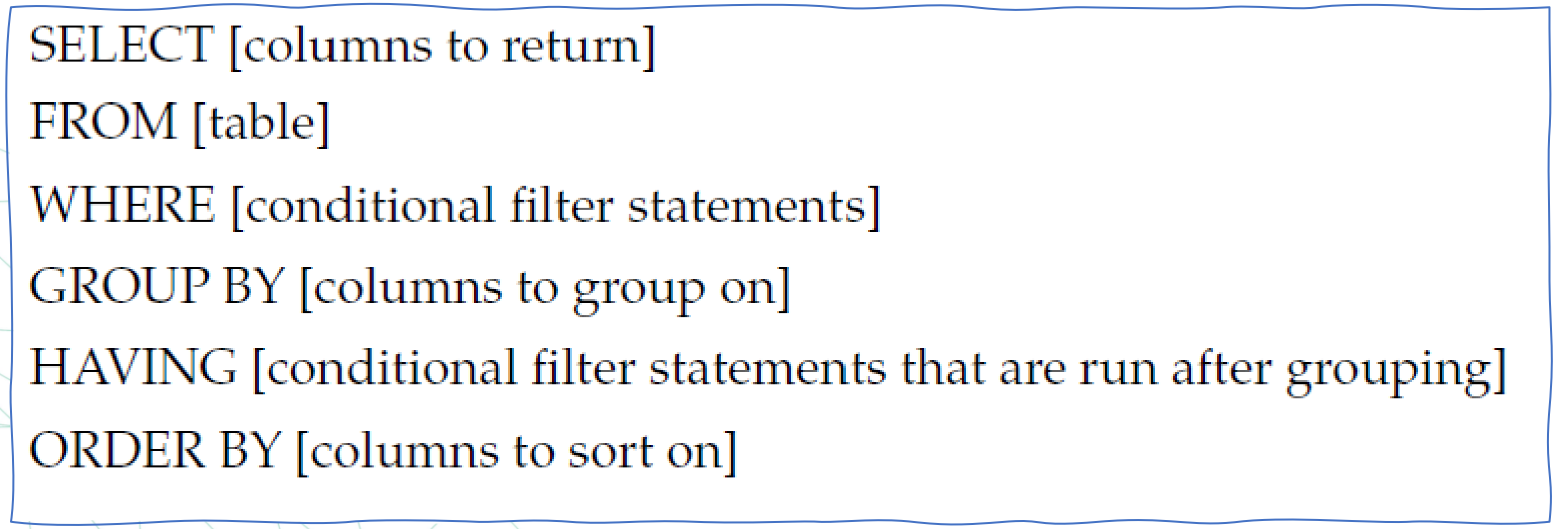


```
SELECT *
FROM customer AS c
RIGHT JOIN customer_purchases AS cp
    ON c.customer_id = cp.customer_id
```

SQL - Aggregating Results for Analysis

Aggregation - **GROUP BY** Syntax

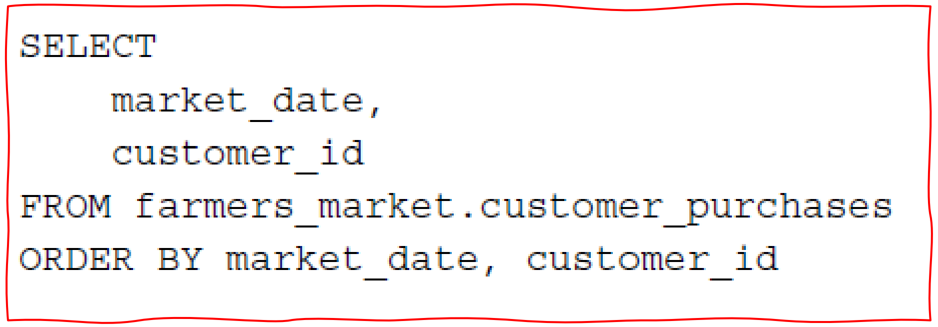
We saw this basic SQL SELECT query syntax, But two sections of this query that we haven’t yet covered, which are both related to aggregation, are the GROUP BY and HAVING clauses:



```
SELECT [columns to return]
FROM [table]
WHERE [conditional filter statements]
GROUP BY [columns to group on]
HAVING [conditional filter statements that are run after grouping]
ORDER BY [columns to sort on]
```

Aggregation - **GROUP BY** Syntax

Using what you’ve learned so far without grouping, you might write a query like the following to get a list of the customer IDs of customers who made purchases on each market date:

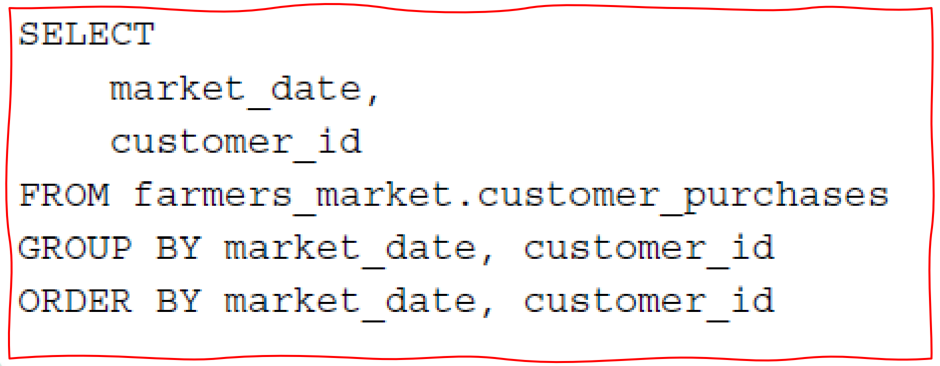


```
SELECT
    market_date,
    customer_id
FROM farmers_market.customer_purchases
ORDER BY market_date, customer_id
```

However, this approach would result in one row per item each customer purchased, displaying duplicates in the output.

Aggregation - **GROUP BY** Syntax

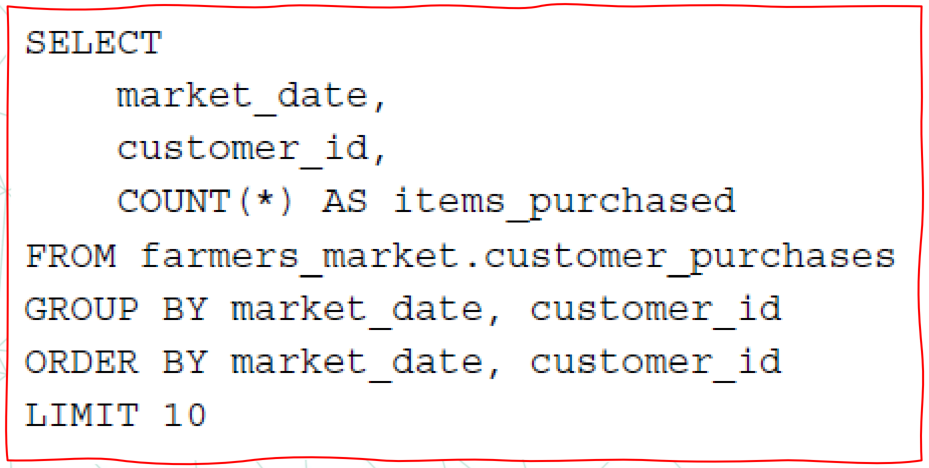
To instead get one row per customer per market date, you can group the results by adding a GROUP BY clause that specifies that you want to summarize the results by the customer_id and market_date fields:



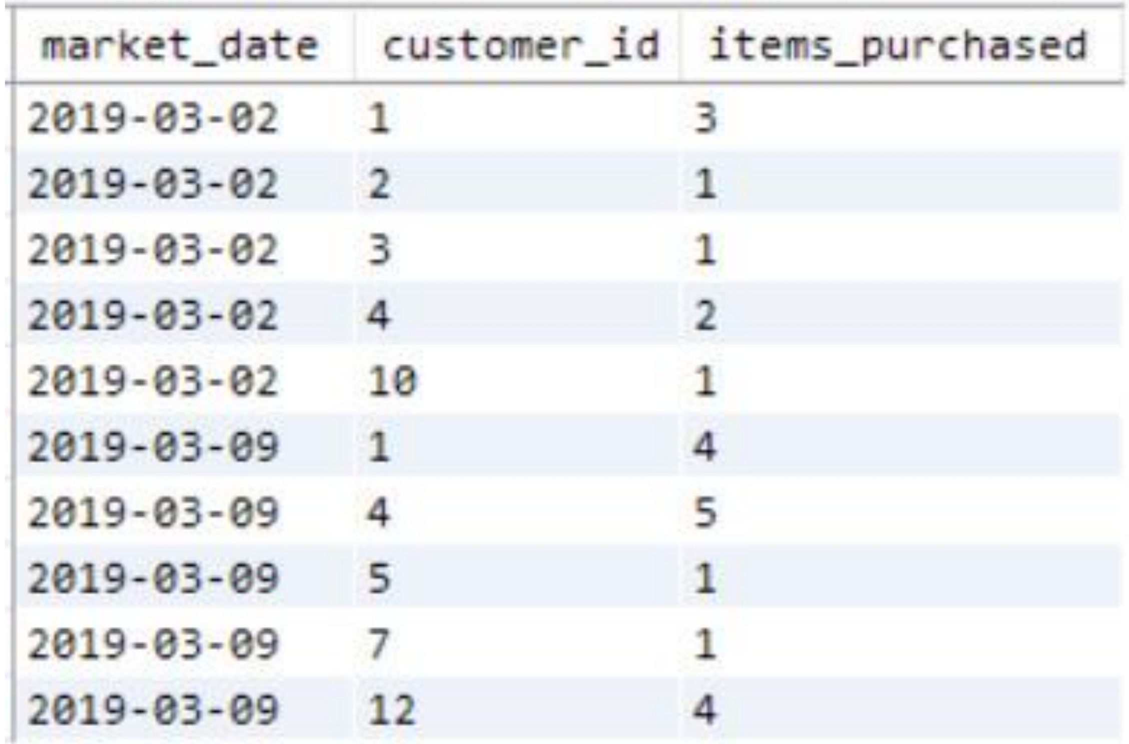
```
SELECT
    market_date,
    customer_id
FROM farmers_market.customer_purchases
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
```

Aggregation - Displaying Group Summaries

Now that you have grouped the data at the desired level, you can add aggregate functions like SUM and COUNT to return summaries of the customer_purchases data per group.



```
SELECT
    market_date,
    customer_id,
    COUNT(*) AS items_purchased
FROM farmers_market.customer_purchases
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
LIMIT 10
```

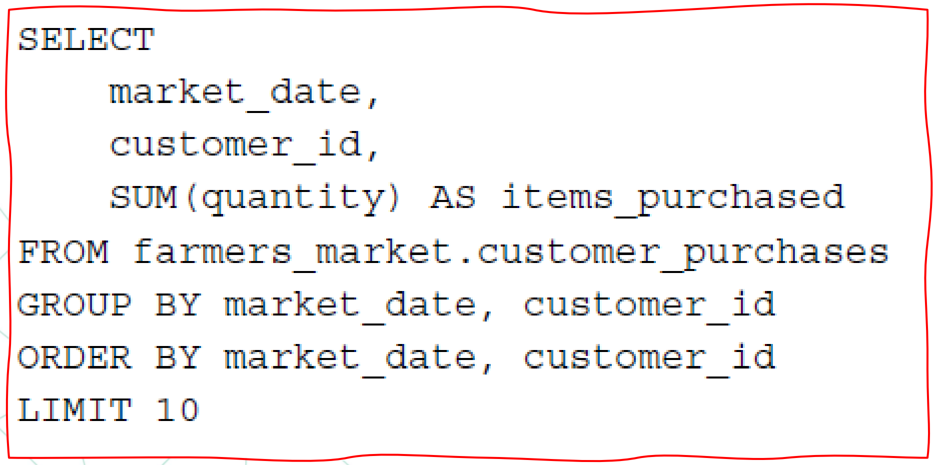


market_date	customer_id	items_purchased
2019-03-02	1	3
2019-03-02	2	1
2019-03-02	3	1
2019-03-02	4	2
2019-03-02	10	1
2019-03-09	1	4
2019-03-09	4	5
2019-03-09	5	1
2019-03-09	7	1
2019-03-09	12	4

Aggregation - Displaying Group Summaries

Remember that the granularity of the customer_purchases table means that if a customer buys three identical items at once, it appears as one row with a quantity of 3. However, if they make separate purchases, it generates multiple rows.

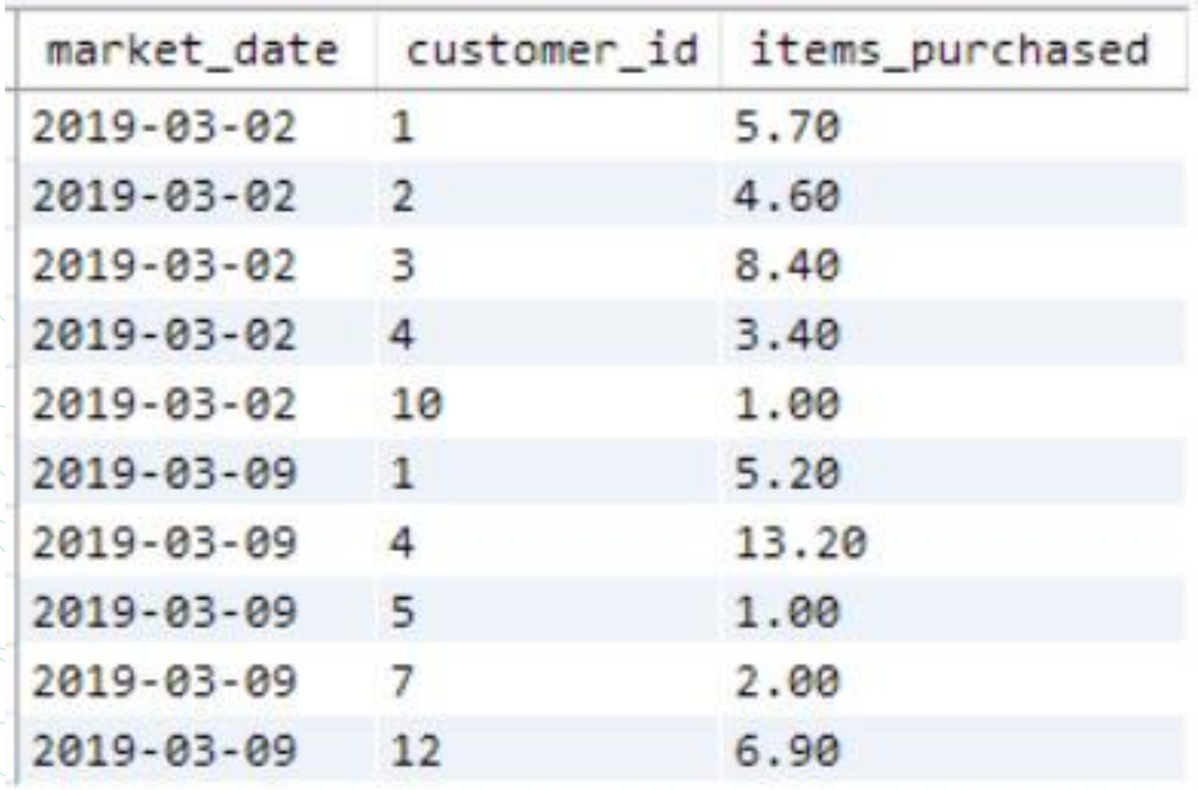
To count all quantities purchased rather than just line items, sum the quantity column with the following query.



```
SELECT
    market_date,
    customer_id,
    SUM(quantity) AS items_purchased
FROM farmers_market.customer_purchases
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
LIMIT 10
```

Aggregation - Displaying Group Summaries

The items_purchased column is no longer an integer, because some of the quantities we’re adding up are bulk product weights.



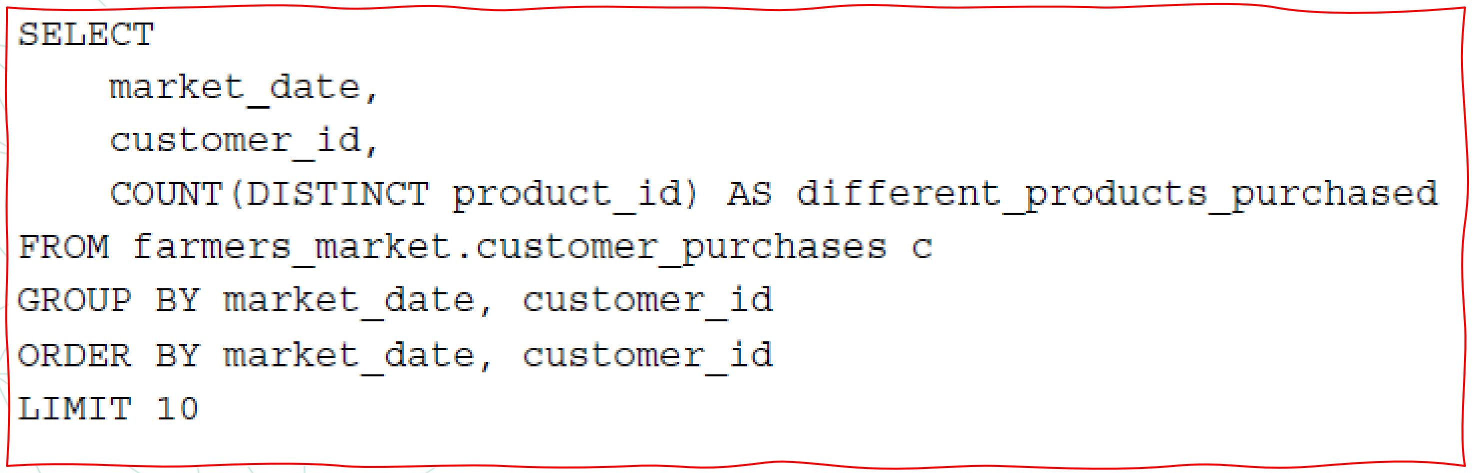
market_date	customer_id	items_purchased
2019-03-02	1	5.70
2019-03-02	2	4.60
2019-03-02	3	8.40
2019-03-02	4	3.40
2019-03-02	10	1.00
2019-03-09	1	5.20
2019-03-09	4	13.20
2019-03-09	5	1.00
2019-03-09	7	2.00
2019-03-09	12	6.90

Aggregation - Displaying Group Summaries

After seeing these results and realizing bulk quantities are included, you may decide it makes more sense to know how many different kinds of items each customer purchased.

So, you only want to count "1" if they bought tomatoes, regardless of the quantity or number of purchases, and add to that count only if they bought other items.

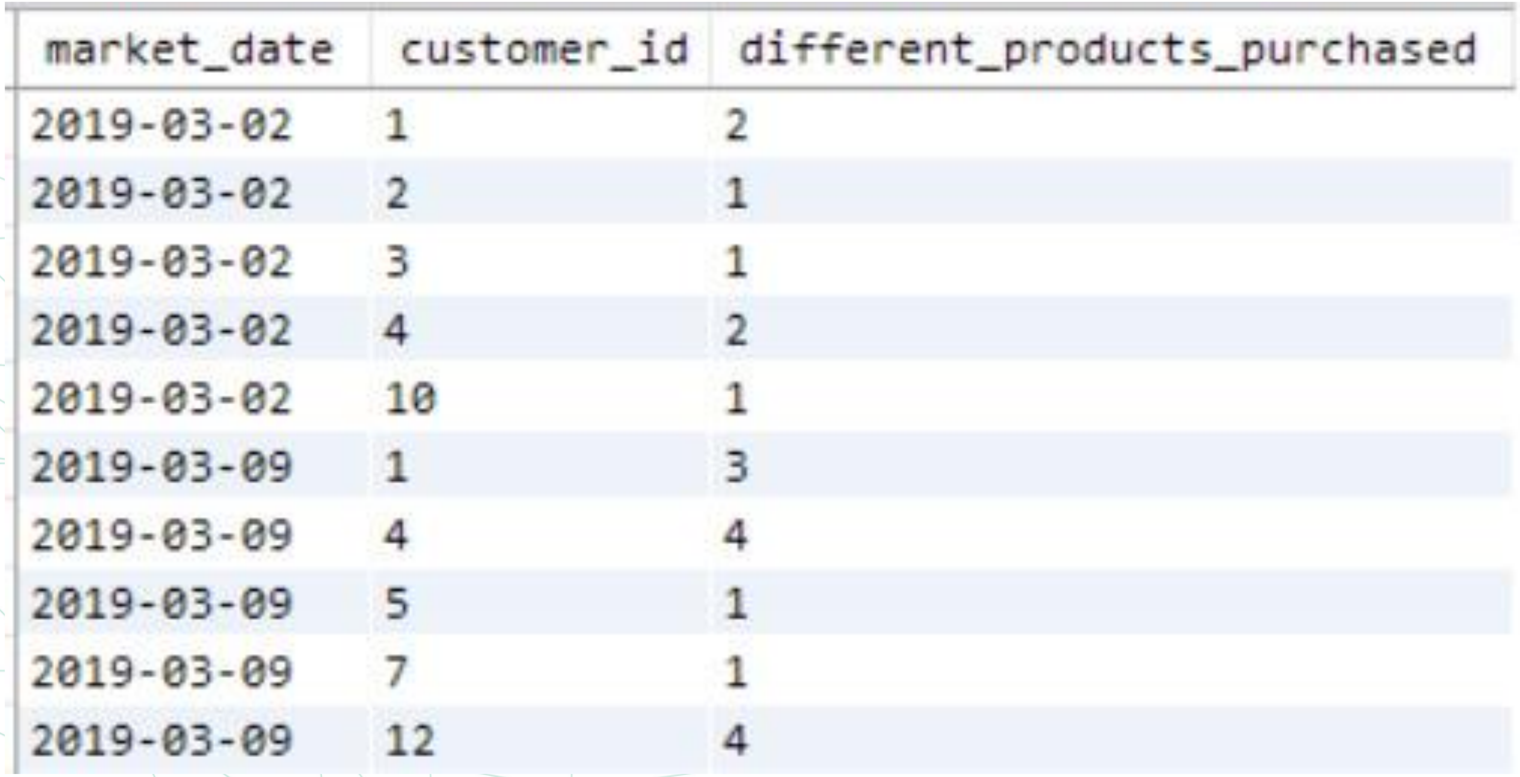
What you want now is a DISTINCT count of product IDs, shown in the following query:



```
SELECT
    market_date,
    customer_id,
    COUNT(DISTINCT product_id) AS different_products_purchased
FROM farmers_market.customer_purchases c
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
LIMIT 10
```

Aggregation - Displaying Group Summaries

Now we're identifying how many unique product_id values exist across those rows in the group—how many different kinds of products were purchased by each customer on each market date:

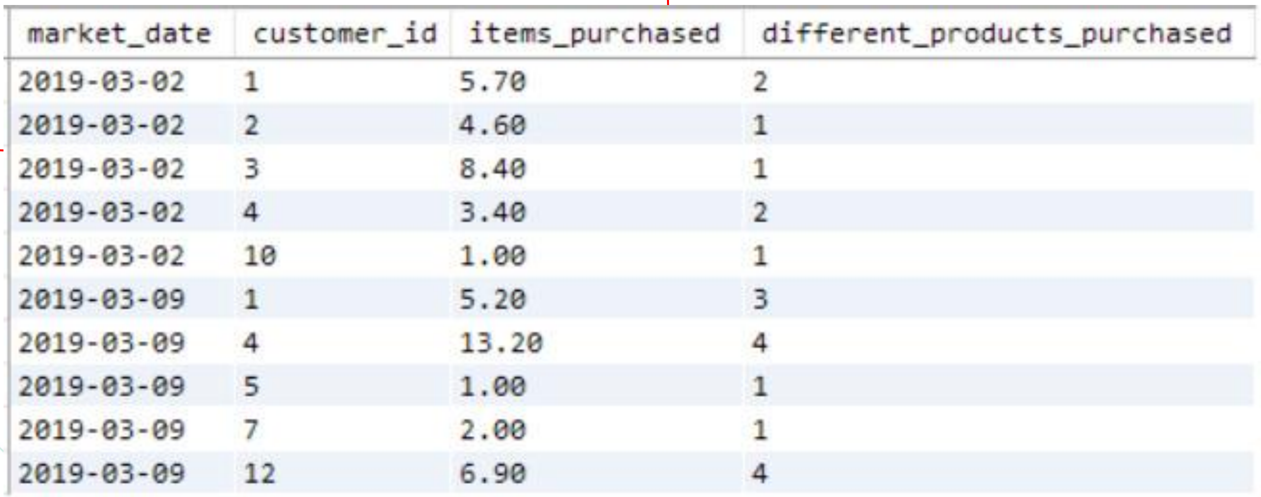


market_date	customer_id	different_products_purchased
2019-03-02	1	2
2019-03-02	2	1
2019-03-02	3	1
2019-03-02	4	2
2019-03-02	10	1
2019-03-09	1	3
2019-03-09	4	4
2019-03-09	5	1
2019-03-09	7	1
2019-03-09	12	4

Aggregation - Displaying Group Summaries

We can also combine these summaries into a single query:

```
SELECT
    market_date,
    customer_id,
    SUM(quantity) AS items_purchased,
    COUNT(DISTINCT product_id) AS different_products_purchased
FROM farmers_market.customer_purchases
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
LIMIT 10
```

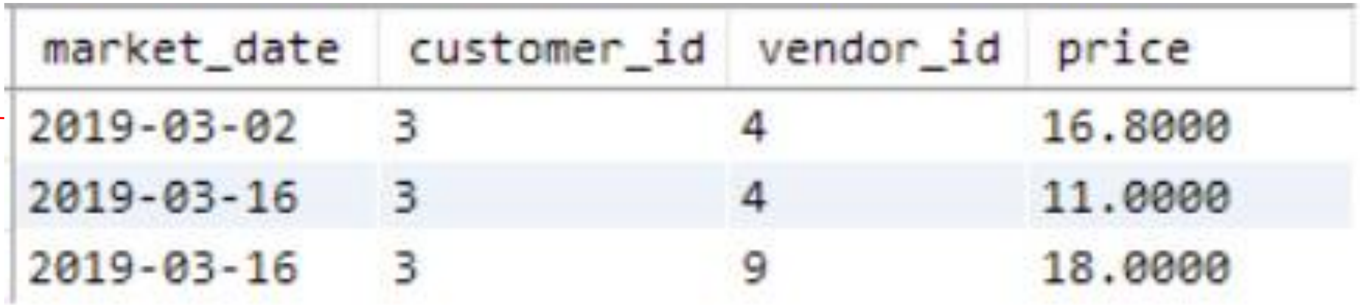


market_date	customer_id	items_purchased	different_products_purchased
2019-03-02	1	5.70	2
2019-03-02	2	4.60	1
2019-03-02	3	8.40	1
2019-03-02	4	3.40	2
2019-03-02	10	1.00	1
2019-03-09	1	5.20	3
2019-03-09	4	13.20	4
2019-03-09	5	1.00	1
2019-03-09	7	2.00	1
2019-03-09	12	6.90	4

Aggregation - Calculations Inside

You can also include mathematical operations, which are calculated at the row level prior to summarization, inside the aggregate functions.

You learned to display a list of customer purchases at the farmer’s market using a WHERE clause to filter for a specific customer.



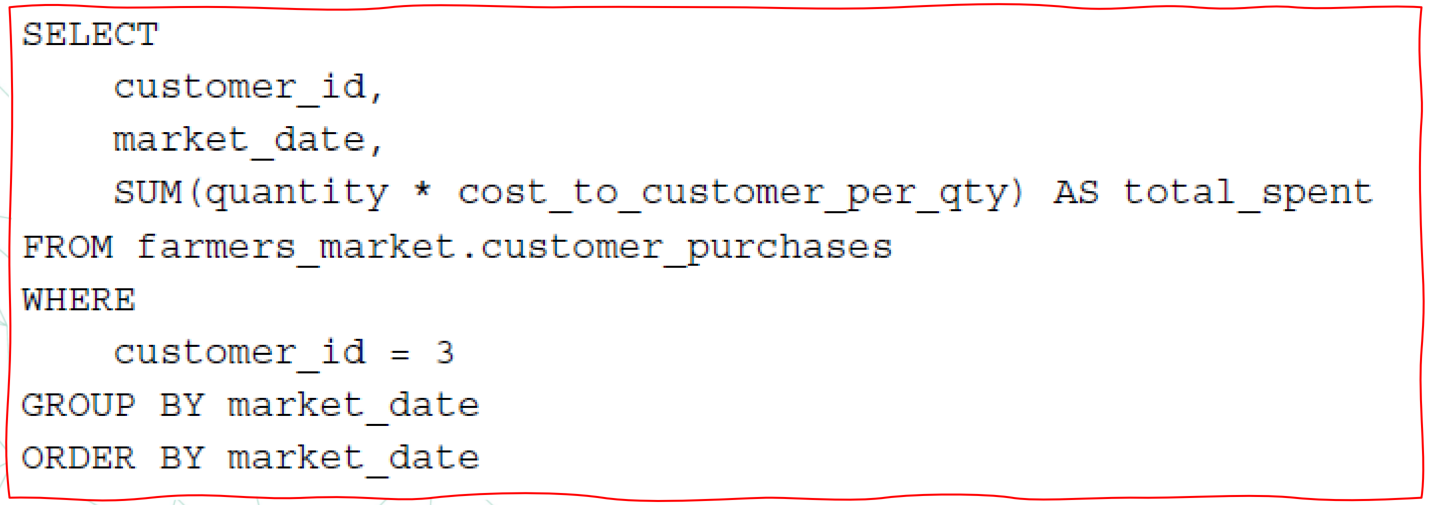
market_date	customer_id	vendor_id	price
2019-03-02	3	4	16.8000
2019-03-16	3	4	11.0000
2019-03-16	3	9	18.0000

```
SELECT
    market_date,
    customer_id,
    vendor_id,
    quantity * cost_to_customer_per_qty AS price
FROM farmers_market.customer_purchases
WHERE
    customer_id = 3
ORDER BY market_date, vendor_id
```

Aggregation - Calculations Inside

Let’s say we wanted to know how much money this customer spent total on each market_date, regardless of item or vendor.

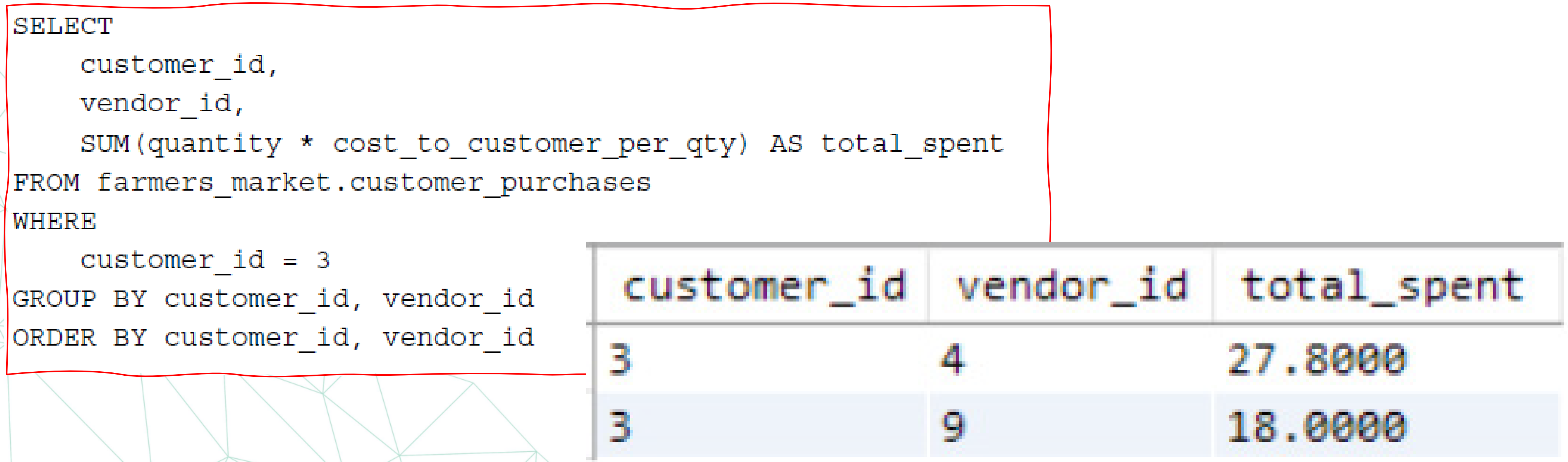
We can GROUP BY market_date, and use the SUM aggregate function on the price calculation to add up the prices of the items purchased:



```
SELECT
    customer_id,
    market_date,
    SUM(quantity * cost_to_customer_per_qty) AS total_spent
FROM farmers_market.customer_purchases
WHERE
    customer_id = 3
GROUP BY market_date
ORDER BY market_date
```

Aggregation - Calculations Inside

What if we wanted to find out how much this customer had spent at each vendor, regardless of date? Then we can group by customer_id and vendor_id:

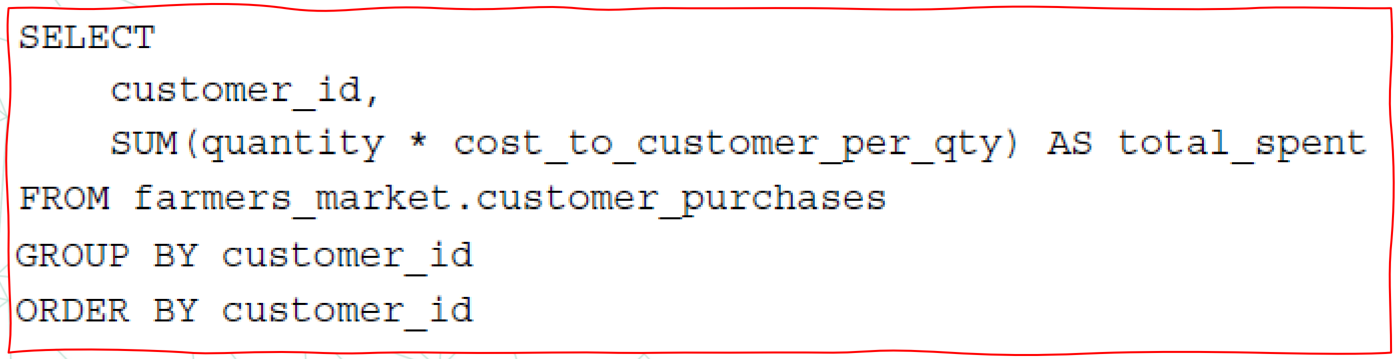


```
SELECT
    customer_id,
    vendor_id,
    SUM(quantity * cost_to_customer_per_qty) AS total_spent
FROM farmers_market.customer_purchases
WHERE
    customer_id = 3
GROUP BY customer_id, vendor_id
ORDER BY customer_id, vendor_id
```

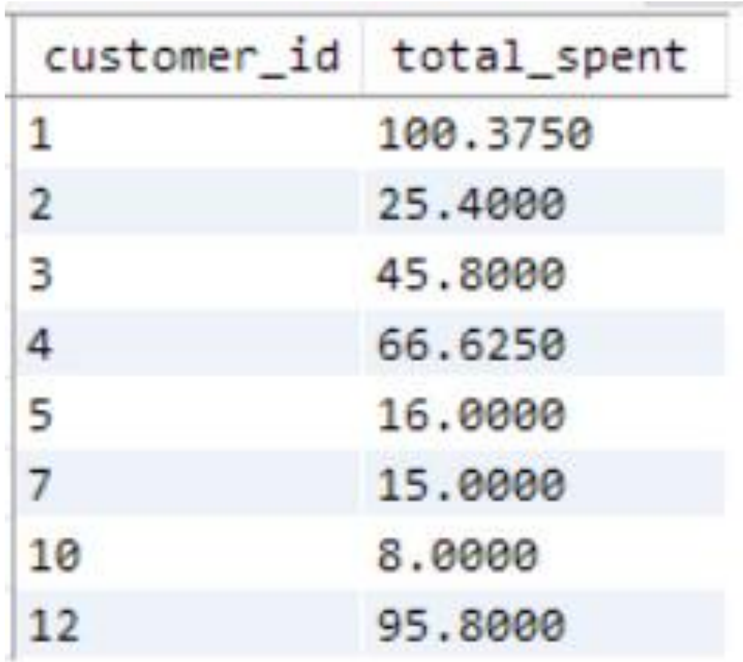
customer_id	vendor_id	total_spent
3	4	27.8000
3	9	18.0000

Aggregation - Calculations Inside

We can also remove the customer_id filter—in this case by removing the entire WHERE clause —and GROUP BY customer_id only, to get a list of every customer and how much they have ever spent at the farmer’s market.



```
SELECT
    customer_id,
    SUM(quantity * cost_to_customer_per_qty) AS total_spent
FROM farmers_market.customer_purchases
GROUP BY customer_id
ORDER BY customer_id
```



customer_id	total_spent
1	100.3750
2	25.4000
3	45.8000
4	66.6250
5	16.0000
7	15.0000
10	8.0000
12	95.8000

Aggregation - Calculations Inside

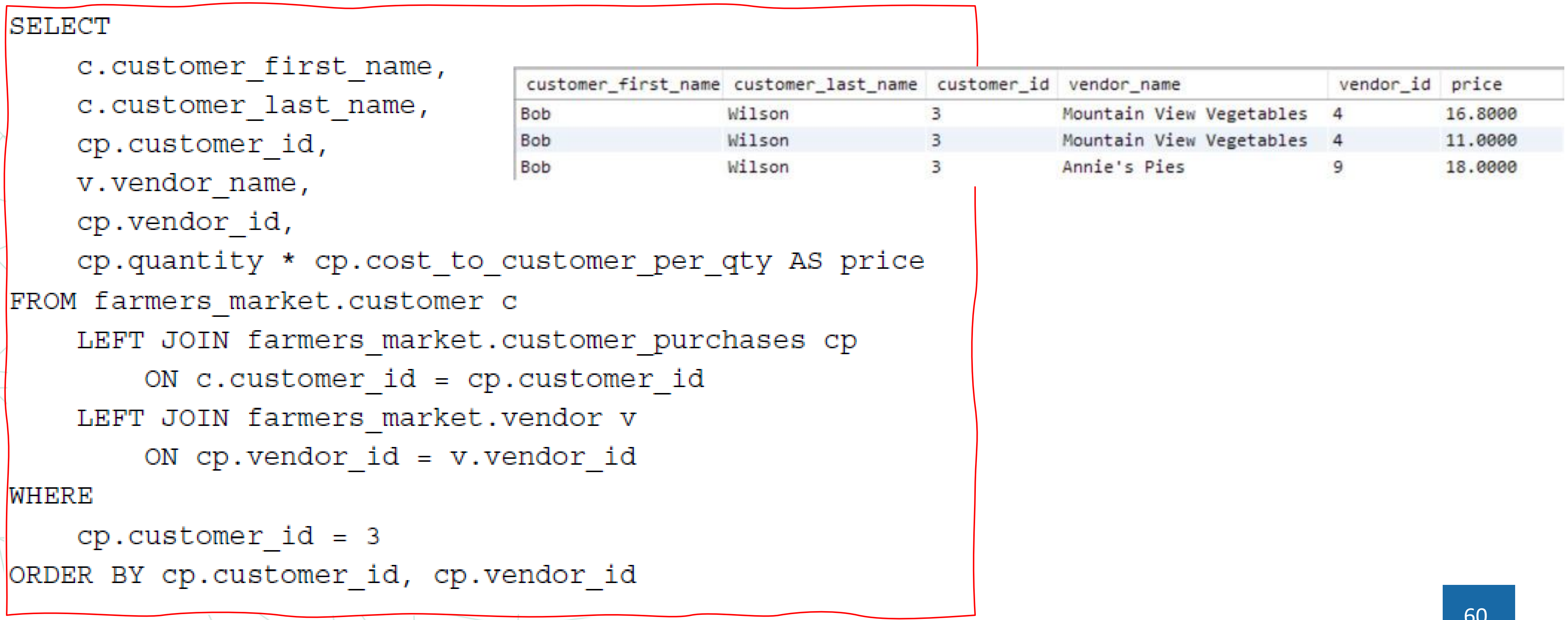
So far, we have been doing all of this aggregation on a single table, but it can be done on joined tables, as well.

It’s a good idea to join the tables without the aggregate functions first, to make sure the data is at the level of granularity you expect (and not generating duplicates) before adding the GROUP BY.

Let’s say that for the query that was grouped by customer_id and vendor_id, we want to bring in some customer details, such as first and last name, and the vendor name.

Aggregation - Calculations Inside

We can first join the three tables together, select columns from all of the tables, and inspect the output before grouping.



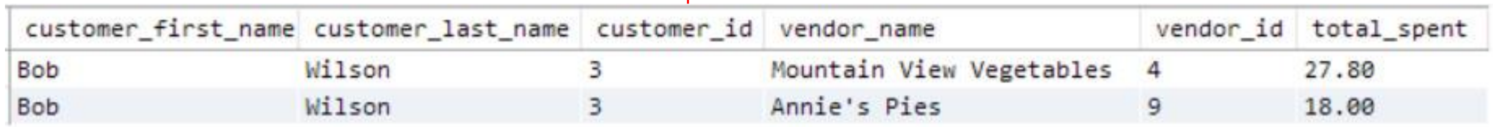
```
SELECT
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id,
    cp.quantity * cp.cost_to_customer_per_qty AS price
FROM farmers_market.customer c
    LEFT JOIN farmers_market.customer_purchases cp
        ON c.customer_id = cp.customer_id
    LEFT JOIN farmers_market.vendor v
        ON cp.vendor_id = v.vendor_id
WHERE
    cp.customer_id = 3
ORDER BY cp.customer_id, cp.vendor_id
```

customer_first_name	customer_last_name	customer_id	vendor_name	vendor_id	price
Bob	Wilson	3	Mountain View Vegetables	4	16.8000
Bob	Wilson	3	Mountain View Vegetables	4	11.0000
Bob	Wilson	3	Annie's Pies	9	18.0000

Aggregation - Calculations Inside

To summarize with one row per customer per vendor, group by all fields from both the customer and vendor tables that aren't aggregate functions.

```
SELECT
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id,
    ROUND(SUM(quantity * cost_to_customer_per_qty), 2) AS total_spent
FROM farmers_market.customer c
    LEFT JOIN farmers_market.customer_purchases cp
        ON c.customer_id = cp.customer_id
    LEFT JOIN farmers_market.vendor v
        ON cp.vendor_id = v.vendor_id
WHERE
    cp.customer_id = 3
GROUP BY
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id
ORDER BY cp.customer_id, cp.vendor_id
```



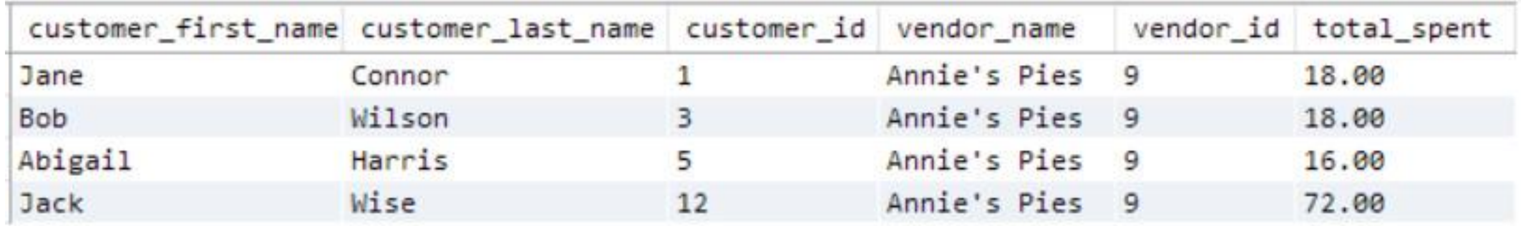
customer_first_name	customer_last_name	customer_id	vendor_name	vendor_id	total_spent
Bob	Wilson	3	Mountain View Vegetables	4	27.80
Bob	Wilson	3	Annie's Pies	9	18.00

Aggregation - Calculations Inside

We can keep the same aggregation and filter to a single vendor instead of a single customer, to get a list of customers per vendor instead of vendors per customer

```
SELECT
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id,
    ROUND(SUM(quantity * cost_to_customer_per_qty), 2) AS total_spent
FROM farmers_market.customer c
    LEFT JOIN farmers_market.customer_purchases cp
        ON c.customer_id = cp.customer_id
    LEFT JOIN farmers_market.vendor v
        ON cp.vendor_id = v.vendor_id
WHERE
    cp.vendor_id = 9
GROUP BY
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id
ORDER BY cp.customer_id, cp.vendor_id
```

Or, we could remove the WHERE clause altogether and get one row for every customer-vendor pair in the database.



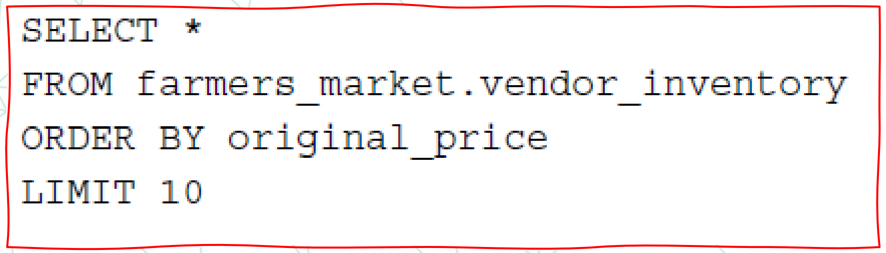
customer_first_name	customer_last_name	customer_id	vendor_name	vendor_id	total_spent
Jane	Connor	1	Annie's Pies	9	18.00
Bob	Wilson	3	Annie's Pies	9	18.00
Abigail	Harris	5	Annie's Pies	9	16.00
Jack	Wise	12	Annie's Pies	9	72.00

Aggregation - MIN and MAX

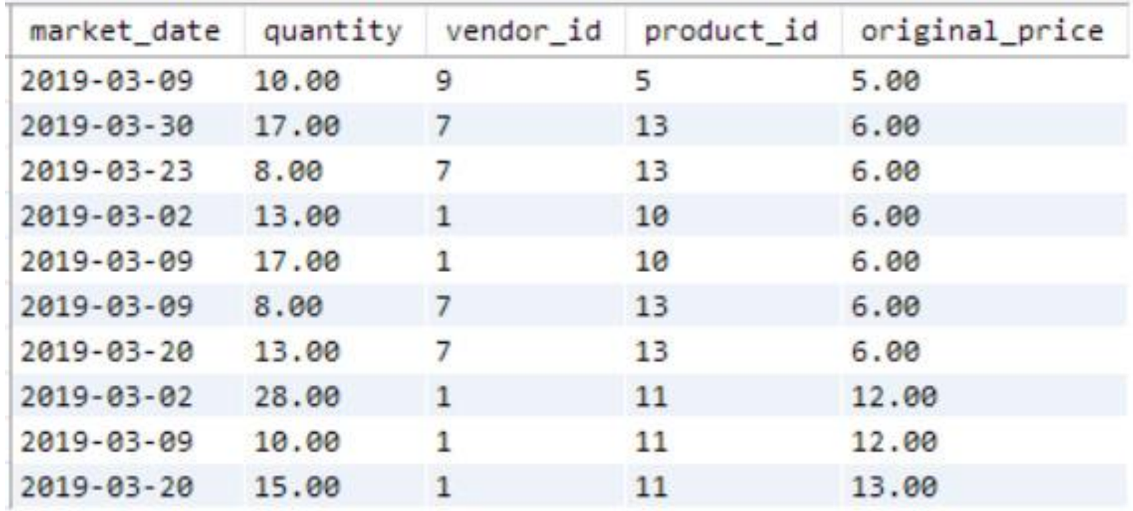
To get the most and least expensive items per product category, use the vendor_inventory table, which lists the original prices set by vendors.

This considers that vendors can adjust prices per customer, as reflected in the cost_to_customer_per_qty field in the customer_purchases table.

First, let's look at all of the available fields in the vendor_inventory table:



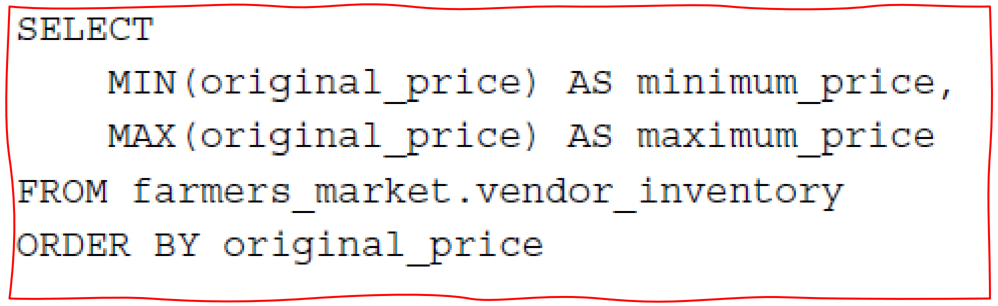
```
SELECT *
FROM farmers_market.vendor_inventory
ORDER BY original_price
LIMIT 10
```



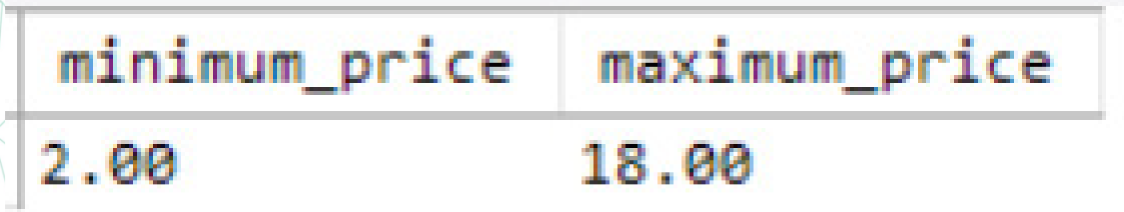
market_date	quantity	vendor_id	product_id	original_price
2019-03-09	10.00	9	5	5.00
2019-03-30	17.00	7	13	6.00
2019-03-23	8.00	7	13	6.00
2019-03-02	13.00	1	10	6.00
2019-03-09	17.00	1	10	6.00
2019-03-09	8.00	7	13	6.00
2019-03-20	13.00	7	13	6.00
2019-03-02	28.00	1	11	12.00
2019-03-09	10.00	1	11	12.00
2019-03-20	15.00	1	11	13.00

Aggregation - MIN and MAX

We can get the least and most expensive item prices in the entire table by using the MIN() and MAX() functions without grouping in MySQL:



```
SELECT
    MIN(original_price) AS minimum_price,
    MAX(original_price) AS maximum_price
FROM farmers_market.vendor_inventory
ORDER BY original_price
```

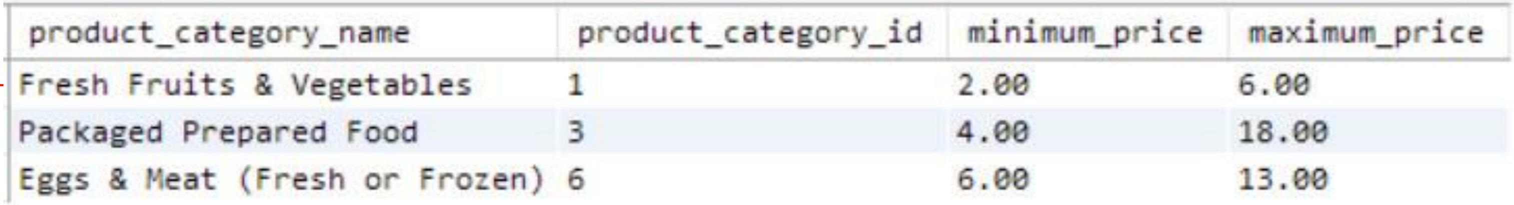


minimum_price	maximum_price
2.00	18.00

Aggregation - MIN and MAX

But if we want to get the lowest and highest prices within each product category, we have to group by the product_category_id (and product_category_name, if we want to display it), then the summary values will be calculated per group:

```
SELECT
    pc.product_category_name,
    p.product_category_id,
    MIN(vi.original_price) AS minimum_price,
    MAX(vi.original_price) AS maximum_price
FROM farmers_market.vendor_inventory AS vi
    INNER JOIN farmers_market.product AS p
        ON vi.product_id = p.product_id
    INNER JOIN farmers_market.product_category AS pc
        ON p.product_category_id = pc.product_category_id
GROUP BY pc.product_category_name, p.product_category_id
```

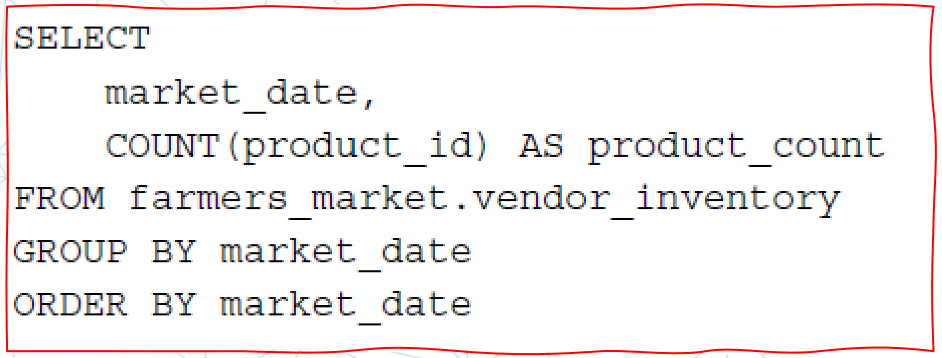


product_category_name	product_category_id	minimum_price	maximum_price
Fresh Fruits & Vegetables	1	2.00	6.00
Packaged Prepared Food	3	4.00	18.00
Eggs & Meat (Fresh or Frozen)	6	6.00	13.00

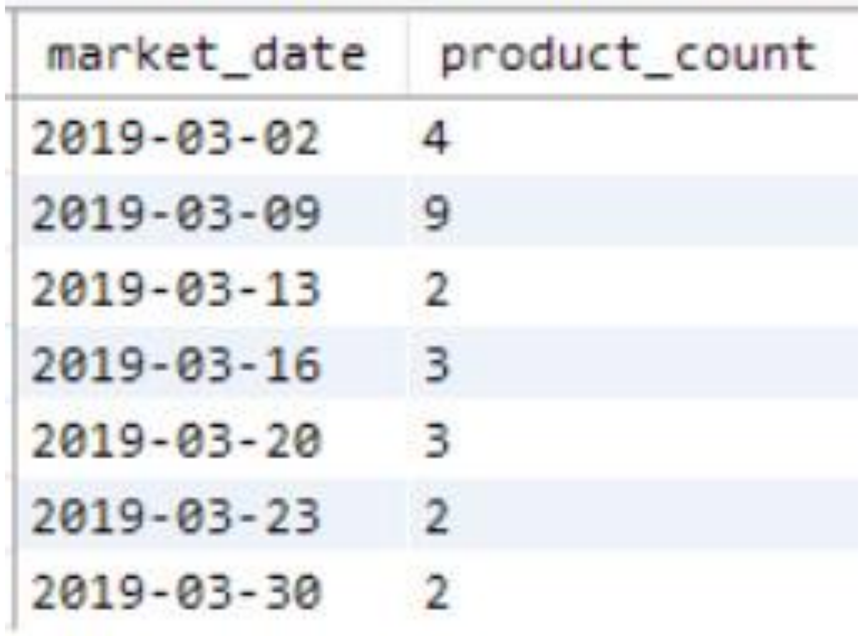
Aggregation - COUNT and COUNT DISTINCT

Suppose we wanted to count how many products were for sale on each market date, or how many different products each vendor offered. We can determine these values using COUNT and COUNT DISTINCT.

COUNT will count up the rows within a group when used with GROUP BY, and COUNT DISTINCT will count up the unique values present in the specified field within the group.



```
SELECT
    market_date,
    COUNT(product_id) AS product_count
FROM farmers_market.vendor_inventory
GROUP BY market_date
ORDER BY market_date
```

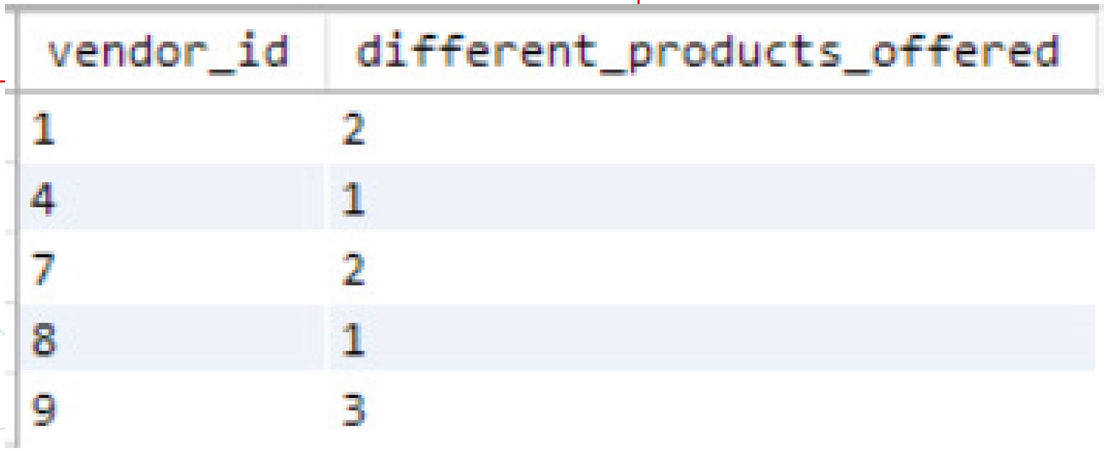


market_date	product_count
2019-03-02	4
2019-03-09	9
2019-03-13	2
2019-03-16	3
2019-03-20	3
2019-03-23	2
2019-03-30	2

Aggregation - COUNT and COUNT DISTINCT

If we wanted to know how many different products—with unique product IDs—each vendor brought to market during a date range, we could use COUNT DISTINCT on the product_id field, like so:

```
SELECT
    vendor_id,
    COUNT(DISTINCT product_id) AS different_products_offered
FROM farmers_market.vendor_inventory
WHERE market_date BETWEEN '2019-03-02' AND '2019-03-16'
GROUP BY vendor_id
ORDER BY vendor_id
```

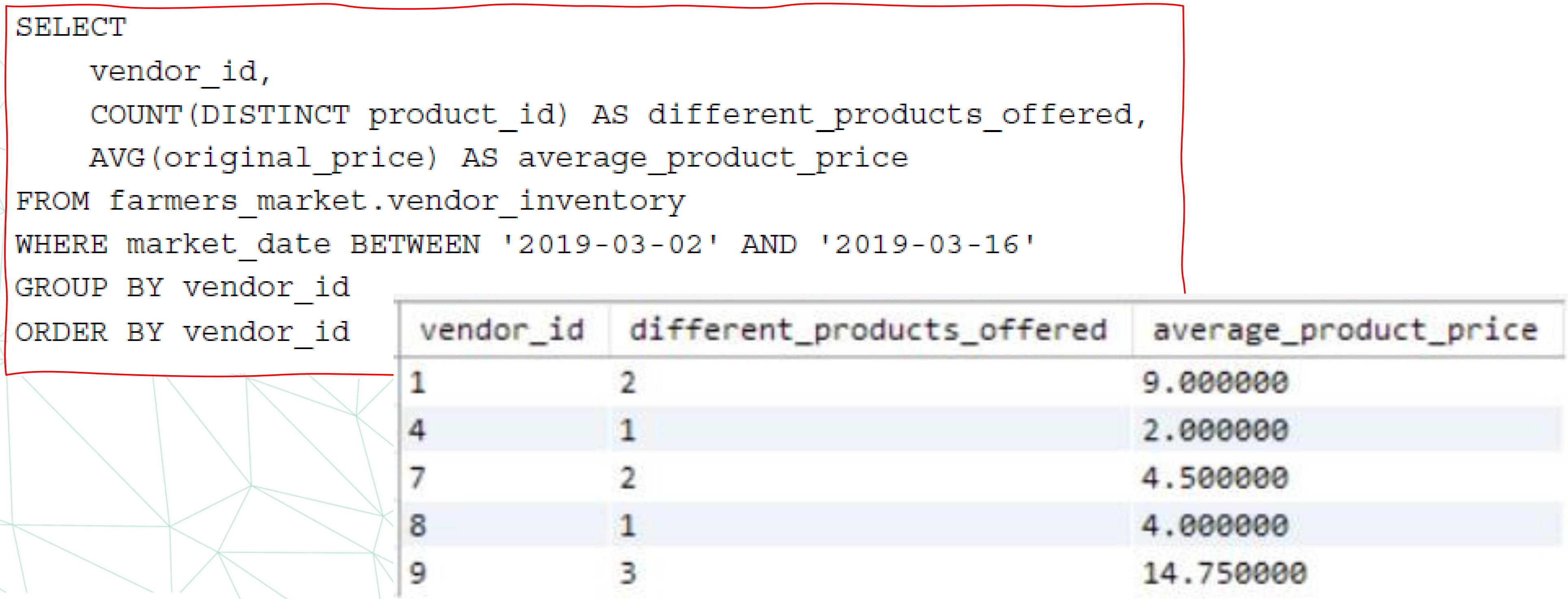


vendor_id	different_products_offered
1	2
4	1
7	2
8	1
9	3

Aggregation - Average

What if we also want the average original price of a product per vendor?

We can add a line to the preceding query, and use the AVG() function:



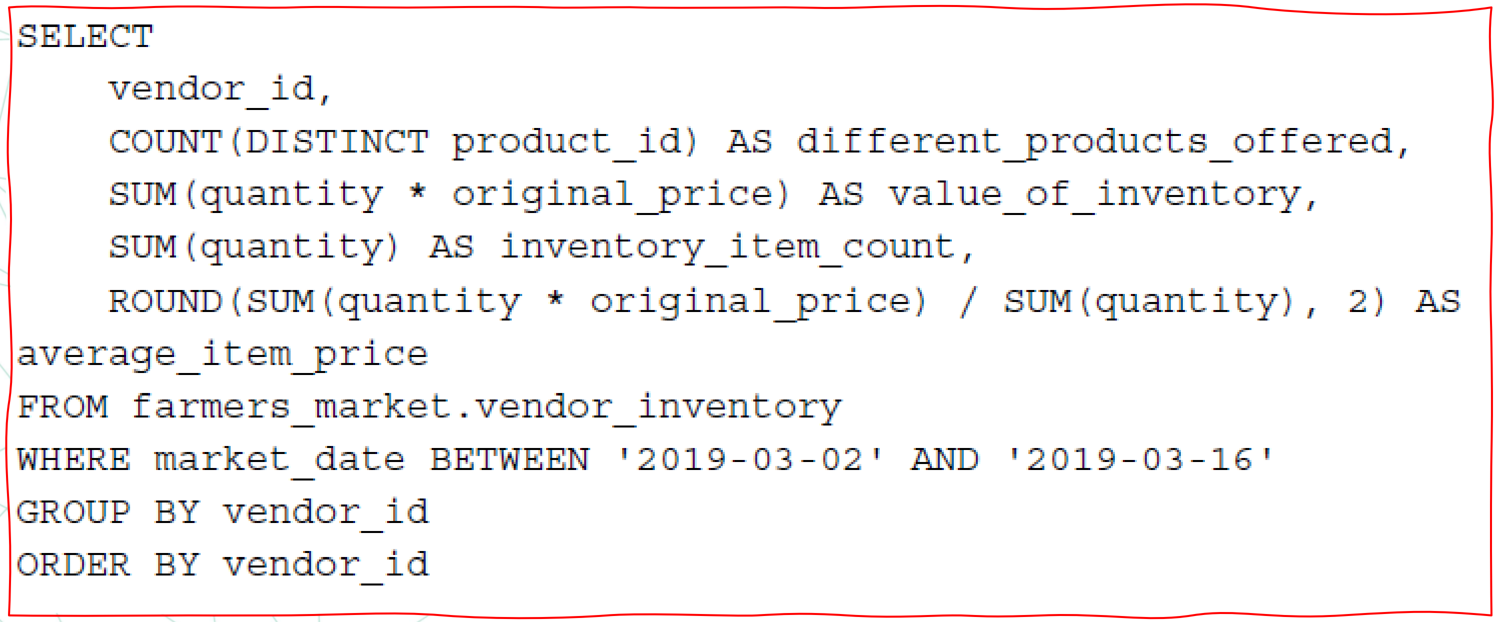
```
SELECT
    vendor_id,
    COUNT(DISTINCT product_id) AS different_products_offered,
    AVG(original_price) AS average_product_price
FROM farmers_market.vendor_inventory
WHERE market_date BETWEEN '2019-03-02' AND '2019-03-16'
GROUP BY vendor_id
ORDER BY vendor_id
```

vendor_id	different_products_offered	average_product_price
1	2	9.000000
4	1	2.000000
7	2	4.500000
8	1	4.000000
9	3	14.750000

Aggregation - Average

However, we have to think about what we’re actually averaging here. Is it fair to call it “average product price” when the underlying table has one row per type of product?

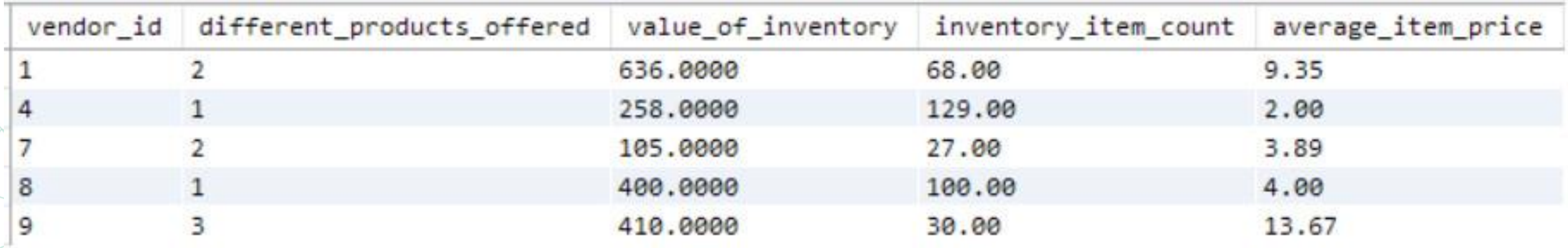
To get an average item price for each vendor's inventory within specified dates, multiply each item's quantity by its price per row, sum these products, and then divide by the total quantity of items per vendor.



```
SELECT
    vendor_id,
    COUNT(DISTINCT product_id) AS different_products_offered,
    SUM(quantity * original_price) AS value_of_inventory,
    SUM(quantity) AS inventory_item_count,
    ROUND(SUM(quantity * original_price) / SUM(quantity), 2) AS
average_item_price
FROM farmers_market.vendor_inventory
WHERE market_date BETWEEN '2019-03-02' AND '2019-03-16'
GROUP BY vendor_id
ORDER BY vendor_id
```

Aggregation - Average

We also surrounded the calculation with a ROUND() function to format the output in dollars:

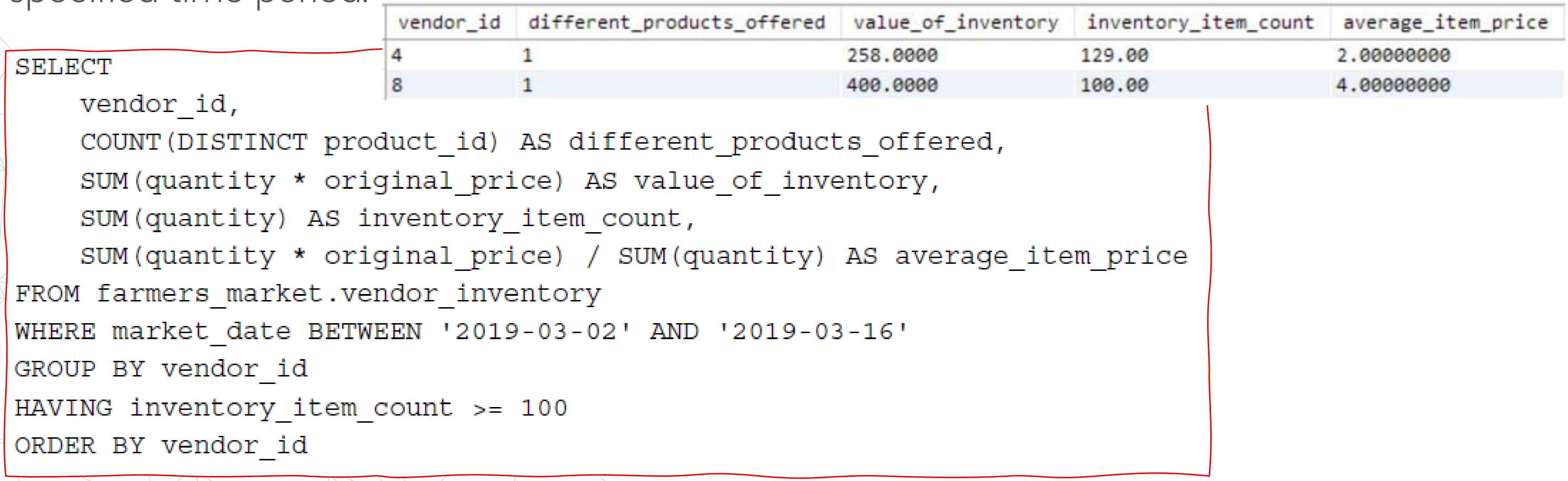


vendor_id	different_products_offered	value_of_inventory	inventory_item_count	average_item_price
1	2	636.0000	68.00	9.35
4	1	258.0000	129.00	2.00
7	2	105.0000	27.00	3.89
8	1	400.0000	100.00	4.00
9	3	410.0000	30.00	13.67

Aggregation - Filtering with HAVING

If you want to filter values after the aggregate functions are applied, you can add a HAVING clause to the query. This filters the groups based on the summary values.

Let's filter to vendors who brought at least 100 items to the farmer's market over the specified time period.



vendor_id	different_products_offered	value_of_inventory	inventory_item_count	average_item_price
4	1	258.0000	129.00	2.00000000
8	1	400.0000	100.00	4.00000000

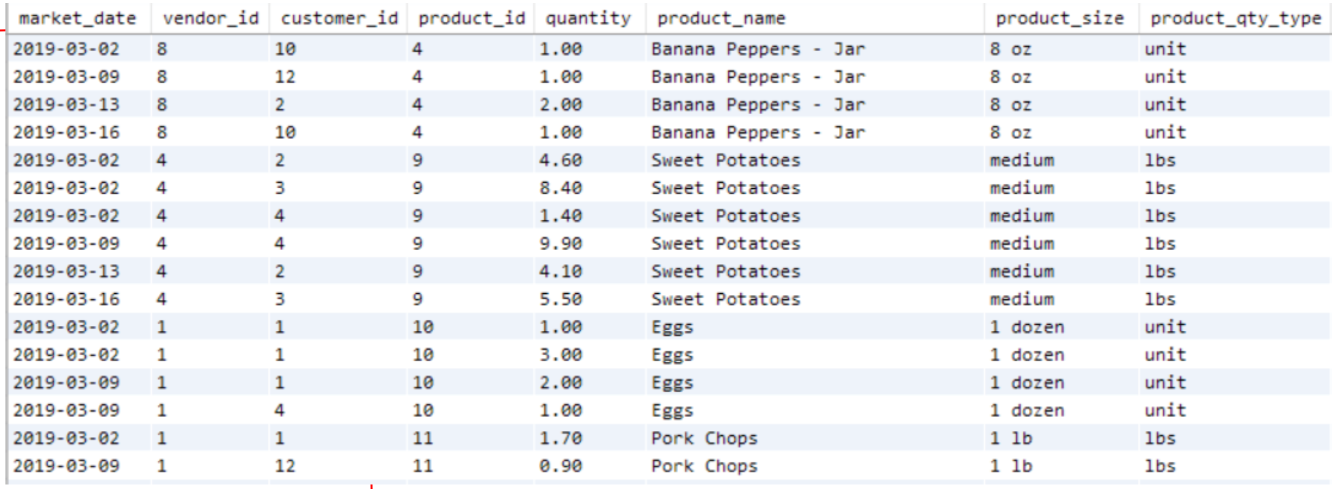
```
SELECT
    vendor_id,
    COUNT(DISTINCT product_id) AS different_products_offered,
    SUM(quantity * original_price) AS value_of_inventory,
    SUM(quantity) AS inventory_item_count,
    SUM(quantity * original_price) / SUM(quantity) AS average_item_price
FROM farmers_market.vendor_inventory
WHERE market_date BETWEEN '2019-03-02' AND '2019-03-16'
GROUP BY vendor_id
HAVING inventory_item_count >= 100
ORDER BY vendor_id
```

Aggregation - CASE Statements Inside

We'll use a CASE statement to specify which type of item quantities to add together using each SUM aggregate function.

First, we'll need to JOIN the customer_purchases table to the product table to pull in the product_qty_type column, which only contains the values "unit" and "lbs":

```
SELECT
    cp.market_date,
    cp.vendor_id,
    cp.customer_id,
    cp.product_id,
    cp.quantity,
    p.product_name,
    p.product_size,
    p.product_qty_type
FROM farmers_market.customer_purchases AS cp
    INNER JOIN farmers_market.product AS p
        ON cp.product_id = p.product_id
```

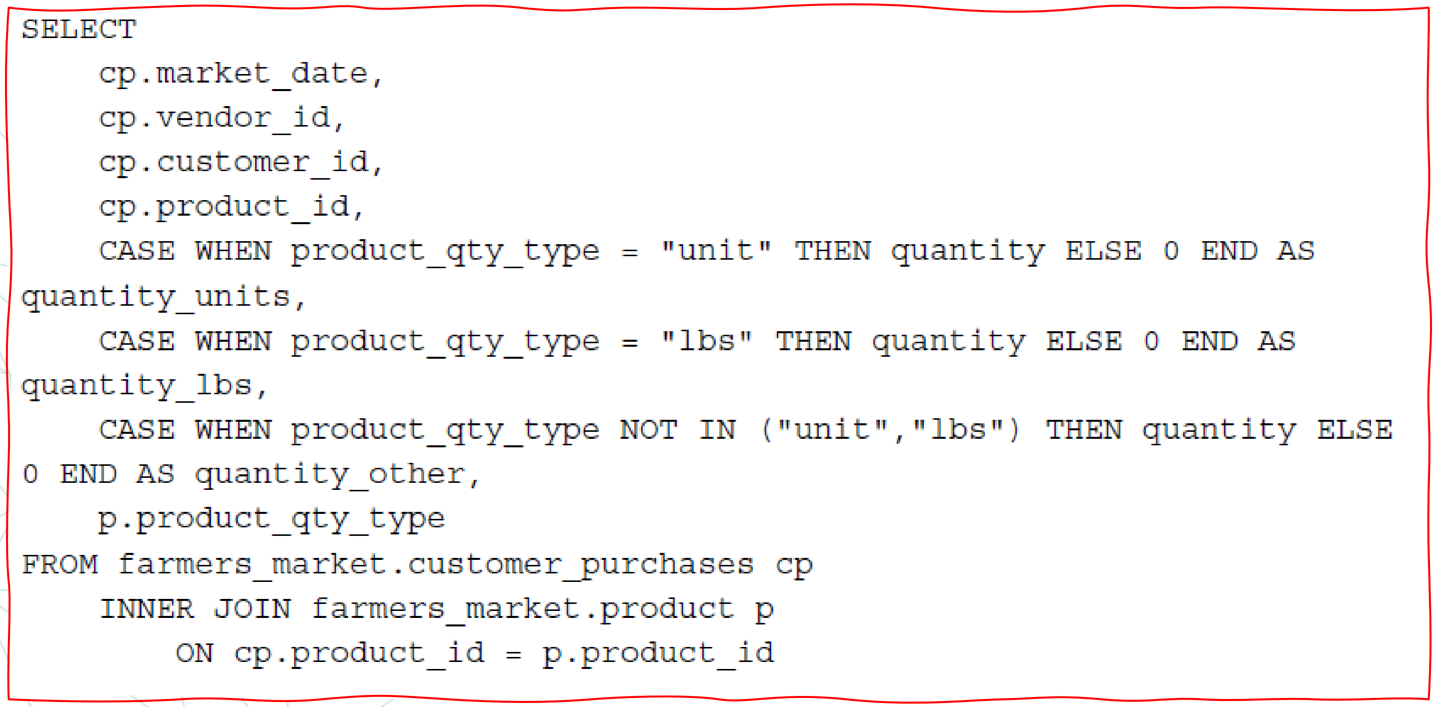


market_date	vendor_id	customer_id	product_id	quantity	product_name	product_size	product_qty_type
2019-03-02	8	10	4	1.00	Banana Peppers - Jar	8 oz	unit
2019-03-09	8	12	4	1.00	Banana Peppers - Jar	8 oz	unit
2019-03-13	8	2	4	2.00	Banana Peppers - Jar	8 oz	unit
2019-03-16	8	10	4	1.00	Banana Peppers - Jar	8 oz	unit
2019-03-02	4	2	9	4.60	Sweet Potatoes	medium	lbs
2019-03-02	4	3	9	8.40	Sweet Potatoes	medium	lbs
2019-03-02	4	4	9	1.40	Sweet Potatoes	medium	lbs
2019-03-09	4	4	9	9.90	Sweet Potatoes	medium	lbs
2019-03-13	4	2	9	4.10	Sweet Potatoes	medium	lbs
2019-03-16	4	3	9	5.50	Sweet Potatoes	medium	lbs
2019-03-02	1	1	10	1.00	Eggs	1 dozen	unit
2019-03-02	1	1	10	3.00	Eggs	1 dozen	unit
2019-03-09	1	1	10	2.00	Eggs	1 dozen	unit
2019-03-09	1	4	10	1.00	Eggs	1 dozen	unit
2019-03-02	1	1	11	1.70	Pork Chops	1 lb	lbs
2019-03-09	1	12	11	0.90	Pork Chops	1 lb	lbs

Aggregation - CASE Statements Inside

To create columns for quantities sold by unit, pound, and other future units, use CASE statements within SUM functions to specify which values to add up in each column.

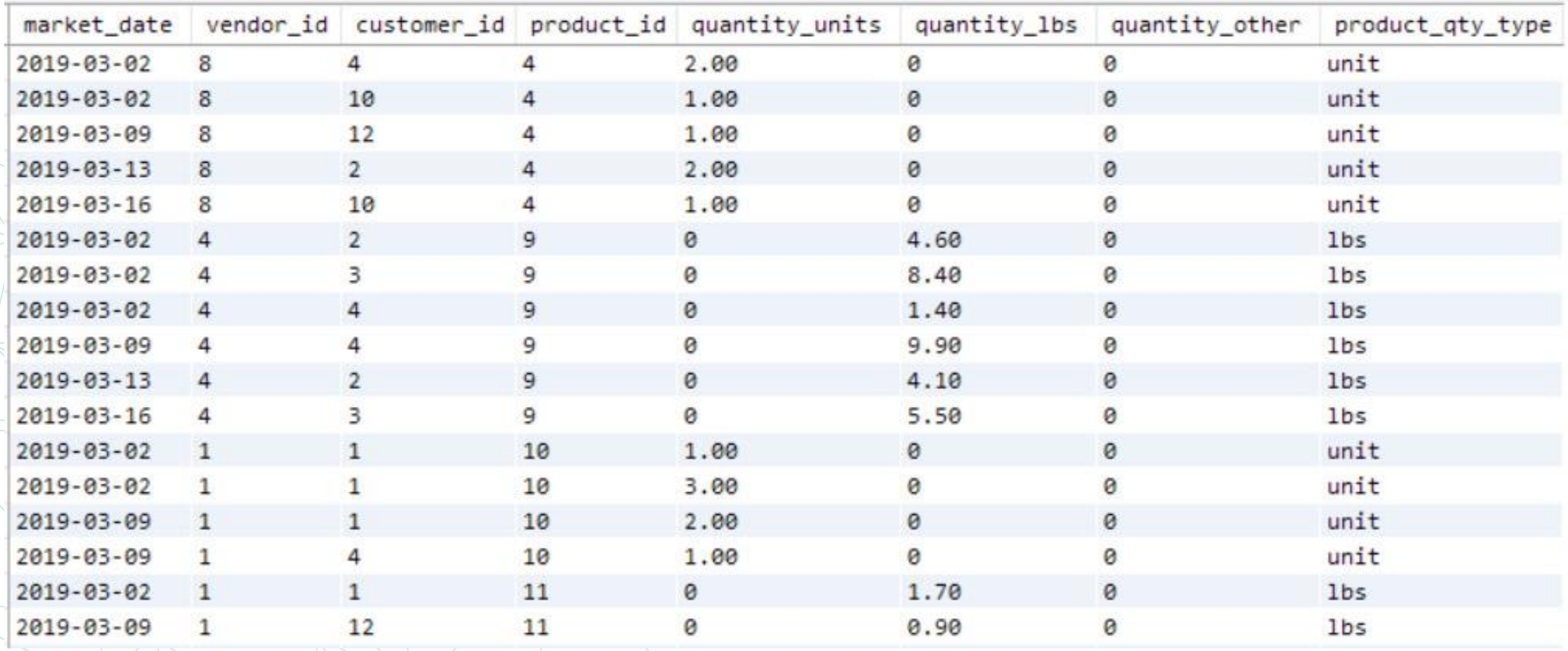
First, we'll review the results with the CASE statements included before grouping or using aggregate functions.



```
SELECT
    cp.market_date,
    cp.vendor_id,
    cp.customer_id,
    cp.product_id,
    CASE WHEN product_qty_type = "unit" THEN quantity ELSE 0 END AS
quantity_units,
    CASE WHEN product_qty_type = "lbs" THEN quantity ELSE 0 END AS
quantity_lbs,
    CASE WHEN product_qty_type NOT IN ("unit","lbs") THEN quantity ELSE
0 END AS quantity_other,
    p.product_qty_type
FROM farmers_market.customer_purchases cp
    INNER JOIN farmers_market.product p
        ON cp.product_id = p.product_id
```

Aggregation - CASE Statements Inside

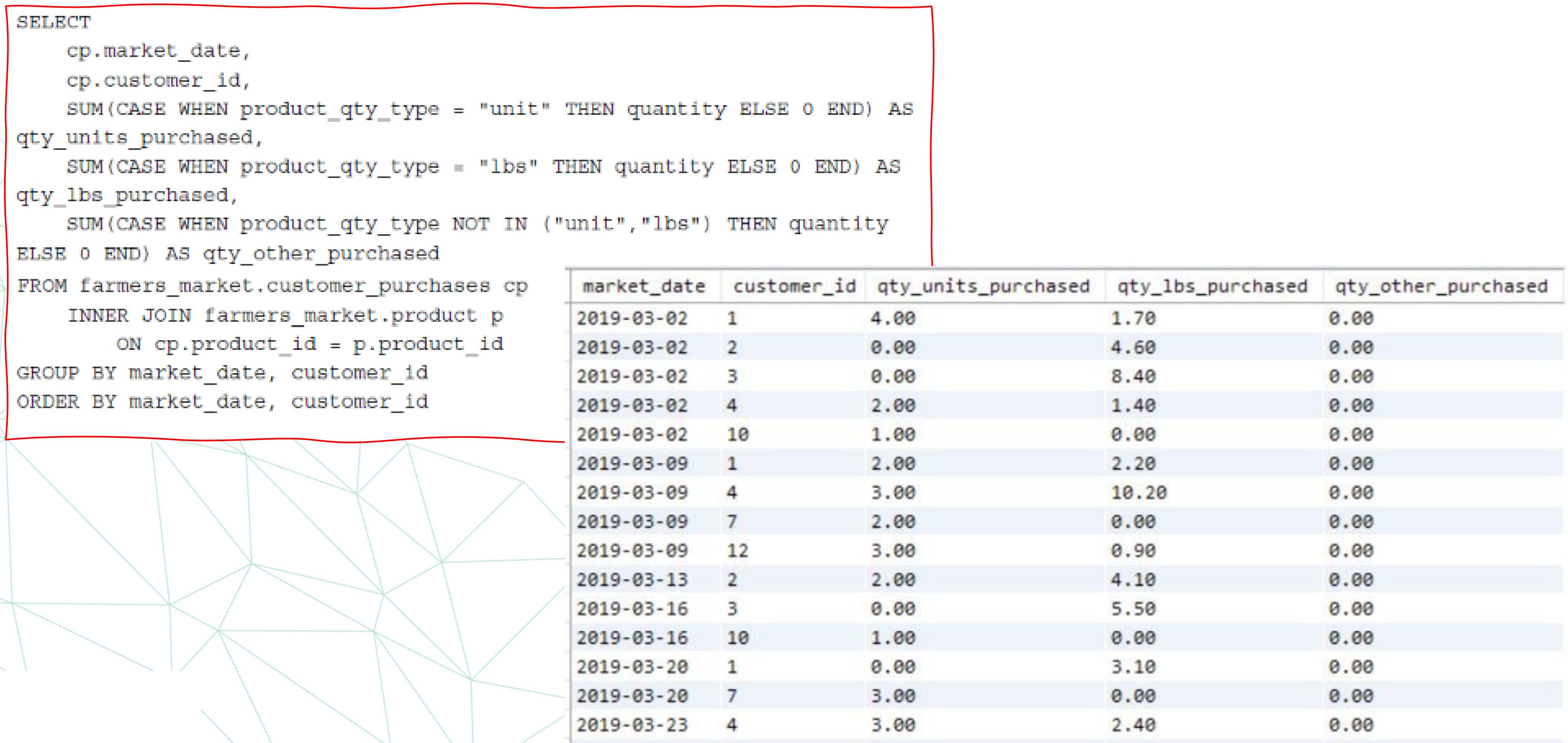
Notice that the CASE statements have separated the quantity values into three different columns, by product_qty_type.



market_date	vendor_id	customer_id	product_id	quantity_units	quantity_lbs	quantity_other	product_qty_type
2019-03-02	8	4	4	2.00	0	0	unit
2019-03-02	8	10	4	1.00	0	0	unit
2019-03-09	8	12	4	1.00	0	0	unit
2019-03-13	8	2	4	2.00	0	0	unit
2019-03-16	8	10	4	1.00	0	0	unit
2019-03-02	4	2	9	0	4.60	0	lbs
2019-03-02	4	3	9	0	8.40	0	lbs
2019-03-02	4	4	9	0	1.40	0	lbs
2019-03-09	4	4	9	0	9.90	0	lbs
2019-03-13	4	2	9	0	4.10	0	lbs
2019-03-16	4	3	9	0	5.50	0	lbs
2019-03-02	1	1	10	1.00	0	0	unit
2019-03-02	1	1	10	3.00	0	0	unit
2019-03-09	1	1	10	2.00	0	0	unit
2019-03-09	1	4	10	1.00	0	0	unit
2019-03-02	1	1	11	0	1.70	0	lbs
2019-03-09	1	12	11	0	0.90	0	lbs

Aggregation - CASE Statements Inside

Now we can add the SUM functions around each CASE statement to add up these values per market date per customer, as defined in the GROUP BY clause.



```
SELECT
    cp.market_date,
    cp.customer_id,
    SUM(CASE WHEN product_qty_type = "unit" THEN quantity ELSE 0 END) AS
qty_units_purchased,
    SUM(CASE WHEN product_qty_type = "lbs" THEN quantity ELSE 0 END) AS
qty_lbs_purchased,
    SUM(CASE WHEN product_qty_type NOT IN ("unit","lbs") THEN quantity
ELSE 0 END) AS qty_other_purchased
FROM farmers_market.customer_purchases cp
    INNER JOIN farmers_market.product p
        ON cp.product_id = p.product_id
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
```

market_date	customer_id	qty_units_purchased	qty_lbs_purchased	qty_other_purchased
2019-03-02	1	4.00	1.70	0.00
2019-03-02	2	0.00	4.60	0.00
2019-03-02	3	0.00	8.40	0.00
2019-03-02	4	2.00	1.40	0.00
2019-03-02	10	1.00	0.00	0.00
2019-03-09	1	2.00	2.20	0.00
2019-03-09	4	3.00	10.20	0.00
2019-03-09	7	2.00	0.00	0.00
2019-03-09	12	3.00	0.90	0.00
2019-03-13	2	2.00	4.10	0.00
2019-03-16	3	0.00	5.50	0.00
2019-03-16	10	1.00	0.00	0.00
2019-03-20	1	0.00	3.10	0.00
2019-03-20	7	3.00	0.00	0.00
2019-03-23	4	3.00	2.40	0.00

Exercise

1. Write a query that determines how many times each vendor has rented a booth at the farmer’s market. In other words, count the vendor booth assignments per vendor_id.

2. We asked earlier “When is each type of fresh fruit or vegetable in season, locally?” Write a query that displays the product category name, product name, earliest date available, and latest date available for every product in the “Fresh Fruits & Vegetables” product category.

3. The Farmer’s Market Customer Appreciation Committee wants to give a bumper sticker to everyone who has ever spent more than $50 at the market. Write a query that generates a list of customers for them to give stickers to, sorted by last name, then first name. (HINT: This query requires you to join two tables, use an aggregate function, and use the HAVING keyword.)

Q&A

Questions and answers

Thanks!