GROUP BY Syntax

```
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]
```

- `GROUP BY` is followed by column names to summarize query results.
- Without grouping, a query listing customer IDs by market date shows duplicates per purchase.

```
SELECT
    market_date, customer_id
FROM farmers_market.customer_purchases
ORDER BY market_date, customer_id
LIMIT 5
```

Table 6.1

market_date	customer_id
2019-04-03	3
2019-04-03	4
2019-04-03	5
2019-04-03	5
2019-04-03	6

- Use `GROUP BY` to get one row per customer per market date.
  - Group by `customer_id` and `market_date`.

```
SELECT
    market_date, customer_id
FROM farmers_market.customer_purchases
GROUP BY market_date, customer_id
ORDER BY market_date, customer_id
LIMIT 5
```

Table 6.3

market_date	customer_id
2019-04-03	3
2019-04-03	4
2019-04-03	5
2019-04-03	6
2019-04-03	7

- You could also use `SELECT DISTINCT` to remove duplicates. This query provides the same result:

```sql
SELECT DISTINCT
    market_date, customer_id
FROM farmers_market.customer_purchases
ORDER BY market_date, customer_id
LIMIT 5
```

Displaying Group Summaries

- After grouping, add aggregate functions like `SUM` and `COUNT` to summarize data.
- Use `COUNT()` to count rows in `customer_purchases` per market date per customer.

```sql
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 5
```

Table 6.4

market_date	customer_id	items_purchased
2019-04-03	3	1
2019-04-03	4	1
2019-04-03	5	2
2019-04-03	6	2
2019-04-03	7	1

- The granularity of `customer_purchases` affects `items_purchased`.
    - Three identical items bought at once show as 1 row with quantity 3.
- Separate purchases generate new rows.
- To count all items, sum the quantity column.

```sql
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 5
```

Table 6.5

market_date	customer_id	items_purchased
2019-04-03	3	1.00
2019-04-03	4	1.00
2019-04-03	5	4.00
2019-04-03	6	5.00
2019-04-03	7	5.00

- To find how many different items each customer bought, count unique `product_id` values

- Use `DISTINCT` on `product_id` instead of `COUNT(*)` or `SUM(quantity)`.
- This shows the variety of products each customer bought per market date.

```
SELECT
    market_date, customer_id,
    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 5
```

Table 6.6

market_date	customer_id	different_products_purchased
2019-04-03	3	1
2019-04-03	4	1
2019-04-03	5	2
2019-04-03	6	2
2019-04-03	7	1

- We can 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 5
```

Table 6.7

market_date	customer_id	items_purchased	different_products_purchased
2019-04-03	3	1.00	1
2019-04-03	4	1.00	1
2019-04-03	5	4.00	2
2019-04-03	6	5.00	2
2019-04-03	7	5.00	1

Performing Calculations Inside Aggregate Functions

- Aggregate functions can include mathematical operations, calculated per row before summarizing.

```
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
LIMIT 5
```

Table 6.8

market_date	customer_id	vendor_id	price
2019-04-03	3	3	4.0000
2019-04-13	3	2	18.0000
2019-04-13	3	4	18.0000
2019-04-13	3	4	16.0000
2019-04-13	3	4	4.0000

- To find total spending per customer on each `market_date`, group by `market_date` and use `SUM` to add item prices.

```
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
LIMIT 5
```

Table 6.9

customer_id	market_date	total_spent
3	2019-04-03	4.0000
3	2019-04-13	56.0000
3	2019-04-24	20.0000
3	2019-04-27	72.0000
3	2019-05-01	52.0000

- `vendor_id` is removed from display and `ORDER BY` to get one row per customer per date.
  - Including it prevents aggregation, as customers can buy from multiple vendors on the same date.
- Add `customer_id` to `GROUP BY` to ensure the query works without errors. This query will provide the same result:

```
...
GROUP BY market_date, customer_id
...
```

- To find total spending per vendor, 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
```

Table 6.10

customer_id	vendor_id	total_spent
3	1	291.9536
3	2	332.3101
3	3	713.0967
3	4	599.0258
3	5	310.5352
3	6	438.3000
3	7	242.4963
3	8	558.4940
3	9	345.9458

- Again, this query provides the same result:

```
...
GROUP BY vendor_id
ORDER BY vendor_id
```

- Remove the `WHERE` clause to remove the `customer_id` filter.
- `GROUP BY` `customer_id` to list each customer and their total spending.

```
SELECT
    customer_id, vendor_id,
    SUM(quantity * cost_to_customer_per_qty) AS total_spent
FROM farmers_market.customer_purchases
GROUP BY customer_id, vendor_id
ORDER BY customer_id, vendor_id
LIMIT 5
```

Table 6.11

customer_id	vendor_id	total_spent
1	1	303.7412
1	2	371.8064
1	3	374.9069
1	4	425.1200
1	5	419.9807

- Aggregation can be done on joined tables too.
  - Join tables first, ensuring no duplicated fields.
  - Then, add the `GROUP BY`.
- To add customer and vendor details:
  - Join the three tables.
  - 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
LIMIT 5
```

Table 6.12

customer_first_name	customer_last_name	customer_id	vendor_name	vendor_id	price
Bob	Wilson	3	Chris's Sustainable Eggs & Meats	1	18.4536
Bob	Wilson	3	Chris's Sustainable Eggs & Meats	1	1.0000
Bob	Wilson	3	Chris's Sustainable Eggs & Meats	1	4.0000
Bob	Wilson	3	Chris's Sustainable Eggs & Meats	1	8.0000
Bob	Wilson	3	Chris's Sustainable Eggs & Meats	1	20.0000

- To summarize at the level of one row per vendor for customer id 3:

```
SELECT
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id,
    SUM(quantity * cost_to_customer_per_qty) 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
```

Table 6.13

customer_first_name	customer_last_name	customer_id	vendor_name	vendor_id	total_spent
Bob	Wilson	3	Chris's Sustainable Eggs & Meats	1	291.9536
Bob	Wilson	3	Hernández Salsa & Veggies	2	332.3101
Bob	Wilson	3	Mountain View Vegetables	3	713.0967
Bob	Wilson	3	Fields of Corn	4	599.0258
Bob	Wilson	3	Seashell Clay Shop	5	310.5352
Bob	Wilson	3	Mother's Garlic & Greens	6	438.3000
Bob	Wilson	3	Marco's Peppers	7	242.4963
Bob	Wilson	3	Annie's Pies	8	558.4940
Bob	Wilson	3	Mediterranean Bakery	9	345.9458

- Filter by vendor instead of customer to get a list of customers for a vendor.
  - Change the `WHERE` clause.
  - Grouping and output fields stay the same.
  - `customer_id` now varies, and `vendor_id` is limited to vendor 8.

```
SELECT
    c.customer_first_name,
    c.customer_last_name,
    cp.customer_id,
    v.vendor_name,
    cp.vendor_id,
    SUM(quantity * cost_to_customer_per_qty) 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 = 8
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
LIMIT 5
```

Table 6.14

customer_first_name	customer_last_name	customer_id	vendor_name	vendor_id	total_spent
Jane	Connor	1	Annie's Pies	8	506.1768
Manuel	Diaz	2	Annie's Pies	8	614.0794
Bob	Wilson	3	Annie's Pies	8	558.4940
Deanna	Washington	4	Annie's Pies	8	456.9701
Abigail	Harris	5	Annie's Pies	8	598.3710

- Removing the `WHERE` clause gives a row for every customer-vendor pair.
  - Useful for front-end filtering in tools like `Tableau`.
  - This query lists each customer and vendor, showing the total spent.
  - Users can dynamically filter by customer or vendor in the reporting tool.

MIN and MAX

- To find the most and least expensive items per product category in the `vendor_inventory` table.
  - Vendors set and adjust prices per customer.
  - The `customer_purchases` table has a `cost_to_customer_per_qty` field for price overrides (selling price).
  - The `vendor_inventory` table has original prices per item on each market date.
- We can find the least and most expensive item prices in the entire table by `MIN()` and `MAX()` functions.

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

Table 6.15

minimum_price	maximum_price
0.50	18.00

- To get the lowest and highest prices within each product category:
  - Group by `product_category_id`, which appears in the table `Product`.
  - Display `product_category_name`, which appears in the table `Product Category`
  - Table aliases are used to distinguish fields from multiple tables.

```
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
```

Table 6.16

product_category_name	product_category_id	minimum_price	maximum_price
Fresh Fruits & Vegetables	1	0.50	18.00
Packaged Prepared Food	3	0.50	18.00

- Adding `MIN(product_name)` and `MAX(product_name)` columns will not give the product names with the lowest and highest prices, but the alphabetically first and last product names.
- To find the products with the min and max prices per category, use window functions, which will be explained in the next chapter.

COUNT and COUNT DISTINCT

- To count products for sale on each market date or how many different products each vendor offered, use `COUNT` and `COUNT DISTINCT`.
- `COUNT` counts rows within a group when used with `GROUP BY`.
- `COUNT DISTINCT` counts unique values in the specified field within the group.
- To find the number of products offered each market date:
  - Count rows in the `vendor_inventory` table, grouped by date.
  - This gives the number of products available, as each row represents a product for each vendor on each market date.

```
SELECT
    market_date,
    COUNT(product_id) AS product_count
FROM farmers_market.vendor_inventory
GROUP BY market_date
ORDER BY market_date
LIMIT 5
```

Table 6.17

market_date	product_count
2019-04-03	4
2019-04-06	4
2019-04-10	4
2019-04-13	4
2019-04-17	4

- If we wanted to know how many different products with unique `product_id`s each vendor brought to market during a date range:
  - Use `COUNT DISTINCT` on the `product_id` field.

```
SELECT
    vendor_id,
    COUNT(DISTINCT product_id) AS different_products_offered
FROM farmers_market.vendor_inventory
WHERE market_date BETWEEN '2019-04-03' AND '2019-04-20'
GROUP BY vendor_id
ORDER BY vendor_id
```

- Note that the `DISTINCT` goes inside the parentheses for the `COUNT()` aggregate function.

Table 6.18

vendor_id	different_products_offered
2	3
3	2
4	3
5	2
6	3
7	1
8	2
9	2

Average

- What if we also want the average original price of a product per vendor, in addition to the count of different products per vendor?
  - We can add a line to the previous 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 '2019-04-03' AND '2019-04-20'
GROUP BY vendor_id
ORDER BY vendor_id
```

Table 6.19

vendor_id	different_products_offered	average_product_price
2	3	9.100000
3	2	8.625000
4	3	11.500000
5	2	9.750000
6	3	5.100000
7	1	16.500000
8	2	9.500000
9	2	1.500000

- We need to consider the meaning of "average product price."
  - The table has one row per product type, so each price is included only once.
  - For example, if a vendor sells 100 tomatoes and bouquets at $20, both prices count only once.
  - This calculation gives the average of one tomato and one bouquet.
- To get a true average price:
  - Multiply each item's quantity by its price.
  - Sum these values and divide by the total quantity of items.
  - Let's perform a calculation using these summary values.

```sql
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 '2019-04-03' AND '2019-04-20'
GROUP BY vendor_id
ORDER BY vendor_id
```

- Multiply `quantity * original_price` per row, then calculate the aggregate `SUM`s.
- Divide one `SUM` by the other to get the "average item price."
- This involves operations both before and after `GROUP BY` summarization.

Table 6.20

vendor_id	different_products_offered	value_of_inventory	inventory_item_count	average_item_price
2	3	700.0000	70.00	10.00000000
3	2	921.0000	110.00	8.37272727
4	3	419.0000	46.00	9.10869565
5	2	474.0000	63.00	7.52380952
6	3	253.0000	86.00	2.94186047
7	1	115.5000	7.00	16.50000000
8	2	488.0000	46.00	10.60869565
9	2	24.0000	16.00	1.50000000

Filtering with HAVING

- Filtering can occur after summarization with the `HAVING` clause.
  - `WHERE` filters rows before grouping, as shown previously.
- To filter after aggregation:
  - Use the `HAVING` clause to filter groups based on summary values.
- For example, to filter vendors with more 220 items over a period:
  - Add a `HAVING` clause to the query.

```sql
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 '2019-04-03' AND '2019-04-20'
GROUP BY vendor_id
HAVING inventory_item_count > 50
ORDER BY vendor_id
```

Table 6.21

vendor_id	different_products_offered	value_of_inventory	inventory_item_count	average_item_price
2	3	700.0000	70.00	10.00000000
3	2	921.0000	110.00	8.37272727
5	2	474.0000	63.00	7.52380952
6	3	253.0000	86.00	2.94186047

- Use `GROUP BY` on all distinct fields, then add a `HAVING COUNT(*) > 1` clause.
- Any returned results indicate unwanted duplicates in your dataset.

CASE Statements Inside Aggregate Functions

- Earlier in this chapter, in the query that generated the output in Table 6.7, we added up the quantity value in the `customer_purchases` table.
    - This included discrete items sold individually as well as bulk items sold by ounce or pound.
    - It was awkward to add those quantities together.
- In Chapter 4, "Conditionals / CASE Statements," you learned about conditional `CASE` statements.
    - Here, 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.
    - This column currently only contains the values "unit" and "lbs."
- In Table 6.7, we added quantity values from `customer_purchases`, mixing individual and bulk items, which was awkward.
- Using `CASE` statements, as learned in Chapter 4, we can specify which item quantities to sum.
- First, `JOIN` the `customer_purchases` table with the `product` table to get the `product_qty_type` column, which contains "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
ORDER BY cp.market_date
LIMIT 5
```

Table 6.22

market_date	vendor_id	customer_id	product_id	quantity	product_name	product_size	produ
2019-04-03	3	3	4	1.00	Banana Peppers - Jar	8 oz	unit
2019-04-03	3	4	4	1.00	Banana Peppers - Jar	8 oz	unit
2019-04-03	3	5	4	3.00	Banana Peppers - Jar	8 oz	unit
2019-04-03	3	6	4	4.00	Banana Peppers - Jar	8 oz	unit
2019-04-03	3	7	4	5.00	Banana Peppers - Jar	8 oz	unit

- Create columns to add quantities sold by unit, by pound, and a third for other units (like bulk ounces).
- Review results with `CASE` statements before grouping:
  - The `CASE` statements separate quantities into three columns by `product_qty_type`.
  - These values will be summed per group next.

```
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
ORDER BY cp.market_date
LIMIT 5
```

Table 6.23

market_date	vendor_id	customer_id	product_id	quantity_units	quantity_lbs	quantity_other
2019-04-03	3	3	4	1.00	0	0
2019-04-03	3	4	4	1.00	0	0
2019-04-03	3	5	4	3.00	0	0
2019-04-03	3	6	4	4.00	0	0
2019-04-03	3	7	4	5.00	0	0

- Add `SUM` functions around each `CASE` statement to sum values per market date per customer, as defined in the `GROUP BY` clause.

```sql
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
LIMIT 5
```

Table 6.25

market_date	customer_id	qty_units_purchased	qty_lbs_purchased	qty_other_purchased
2019-04-03	3	1.00	0.00	0.00
2019-04-03	4	1.00	0.00	0.00
2019-04-03	5	4.00	0.00	0.00
2019-04-03	6	5.00	0.00	0.00
2019-04-03	7	5.00	0.00	0.00

Exercises Using the Included Database

1. Write a query that determines how many times each vendor has rented a booth at the farmer's market.
    - Count the vendor booth assignments per `vendor_id`.
2. In Chapter 5, "SQL Joins," Exercise 3, we asked, "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.
    - Sort by last name, then first name.
    - (Hint: This query requires you to join two tables, use an aggregate function, and use the `HAVING` keyword.)