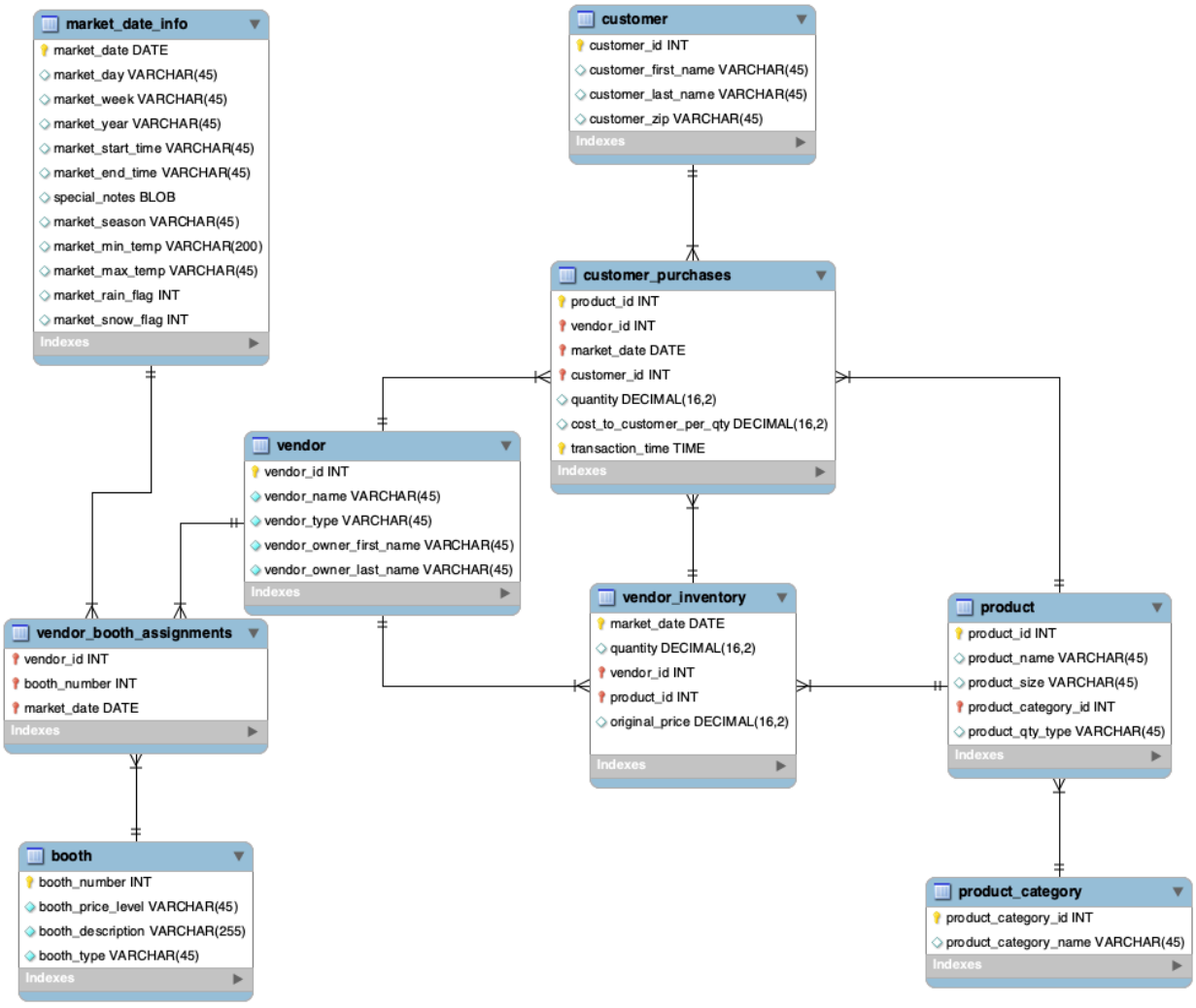
Group By and Aggregation contd.

Problem Statement:

You are a Data Analyst at Amazon Fresh. You have been tasked to study the Farmer's Market.

Dataset: Farmer's Market database



Question: Calculate the total price paid by customer_id 3 per market_date.

We can perform calculations inside the Aggregation Functions

Query:

```
SELECT
        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;
```

- The price will be calculated per row of the table, and then the results will be summed up per group.

Important:

- Notice that **vendor_id has been removed from the list of columns** to be displayed and from the ORDER BY clause.
- That's because if we want the aggregation level of one row per customer per date, we **can't also include vendor_id** in the output, **because the customer can purchase from multiple vendors on a single date**, so the results wouldn't be aggregated at the level we wanted.

Question: What if we wanted to determine how much this customer had spent at each vendor, regardless of date?

Then we can group by customer_id and vendor_id.

Query:

```
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;
```

Question: 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.

- To determine how many products are offered for sale each market date, we can count up the rows in the **vendor_inventory** table, grouped by date.
- This doesn't tell us what quantity of each product was offered or sold, but counts the number of products available, because there is a row in this table for each product for each vendor for each market date.

Query:

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

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:

Query:

```
SELECT
    vendor_id,
    COUNT(DISTINCT product_id) AS different_products_offered
FROM farmers_market.vendor_inventory
WHERE
        market_date BETWEEN '2019-04-03' AND '2019-05-16'
GROUP BY vendor_id
ORDER BY vendor_id;
```

Question: In addition to the count of different products per vendor, 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.

Query:

```
SELECT
    vendor_id,
    COUNT(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 '2019-05-16'
GROUP BY vendor_id, product_id
ORDER BY vendor_id;
```

But is this average product price?

- Is it fair to call it “average product price” when the underlying table has one row per type of product?

- If the vendor brought 100 tomatoes to market, those would all be in one line of the underlying vendor inventory table, so the price of a tomato would only be included in the average once.
- If you calculated the “average product price” for the vendor this way, you would just get the average of the price of one tomato and one bouquet.

How to calculate the price per item?

To get an actual average price of items in each vendor’s inventory between the specified dates, it might make more sense to multiply the quantity of each type of item by the price of that item, which is a calculation that would occur per row, then sum that up and divide by the total quantity of items, which is a calculation that would occur per vendor.

Let’s try a calculation that includes these two summary values.

Query:

```
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_inventory_price
FROM farmers_market.vendor_inventory
WHERE
        market_date BETWEEN '2019-04-03' AND '2019-05-16'
GROUP BY vendor_id
ORDER BY vendor_id;
```

Filtering with Having

Filtering is another thing that can be done in the query after summarization occurs.

Using the **HAVING** clause allows you to filter the results of a query after the aggregate functions are applied, to grouped data.

This filters the groups based on the summary values.

Syntax:

```
SELECT aggregate_function(col)
FROM table
GROUP BY col
HAVING condition;
```

Recall the **Order of Execution** of a SQL query (as discussed earlier):

- **FROM** - The database gets the data from tables in FROM clause and if necessary, performs the JOINs.
- **WHERE** - The data is filtered based on the conditions specified in the WHERE clause. Rows that do not meet the criteria are excluded.
- **GROUP BY** - After filtering the rows using the WHERE clause, the rows that remain are grouped together based on the columns specified in the GROUP BY clause.
- **Aggregate functions** - The aggregate functions are applied to the groups created in the GROUP BY clause.
- **HAVING** - The HAVING clause filters the groups of rows based on aggregate functions applied to the grouped data.
- **SELECT** - After grouping and filtering, the SELECT clause specifies which columns and aggregate functions should be included in the result set.
- **ORDER BY** - It allows you to sort the result set based on one or more columns, either in ascending or descending order.
- **OFFSET** - The specified number of rows are skipped from the beginning of the result set.

- **LIMIT** - After skipping the rows, the LIMIT clause is applied to restrict the number of rows returned.

The HAVING clause is executed after the WHERE and Group By clauses.

Question: Filter out vendors who brought at least 100 items from the farmer's market over the period - 2019-05-02 and 2019-05-16.

Query:

```
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 '2019-05-16'
 GROUP BY vendor_id
 HAVING inventory_item_count >= 100
 ORDER BY vendor_id;
```

HAVING and WHERE

The main difference between the WHERE & HAVING clause is that

- the WHERE clause is used to specify a condition for filtering records before any groupings are made,
- while the HAVING clause is used to specify a condition for filtering values from a group.

Comparison Basis	WHERE Clause	HAVING Clause
Definition	It is used to perform filtration on individual rows.	It is used to perform filtration on groups.
Basic	It is implemented in row operations.	It is implemented in column operations.
Data fetching	The WHERE clause fetches the specific data from particular rows based on the specified condition	The HAVING clause first fetches the complete data. It then separates them according to the given condition.
Aggregate Functions	The WHERE clause does not allow to work with aggregate functions.	The HAVING clause can work with aggregate functions.
Act as	The WHERE clause acts as a pre-filter.	The HAVING clause acts as a post-filter.
Used with	We can use the WHERE clause with the SELECT, UPDATE, and DELETE statements.	The HAVING clause can only use with the SELECT statement.
GROUP BY	The GROUP BY clause comes after the WHERE clause.	The GROUP BY clause comes before the HAVING clause.

Question: Find the average amount spent on each market day. We want to consider only those days where the number of purchases were more than 3 and every single transaction amount must be greater than 5.

Query:

```
SELECT market_date,
  ROUND(AVG(quantity * cost_to_customer_per_qty), 2) AS avg_spent
FROM farmers_market.customer_purchases
WHERE quantity * cost_to_customer_per_qty > 5
GROUP BY market_date
HAVING COUNT(*) > 3
ORDER BY market_date;
```