Filtering contd. and Subqueries

1. Problem Statement
2. DISTINCT
3. Null vs. Blank
4. IS NULL, IS NOT NULL
5. TRIM()
6. IFNULL()
7. Subqueries
8. CASE statement
9. IF() function

Please note that any topics that are not covered in today's lecture will be covered in the next lecture.

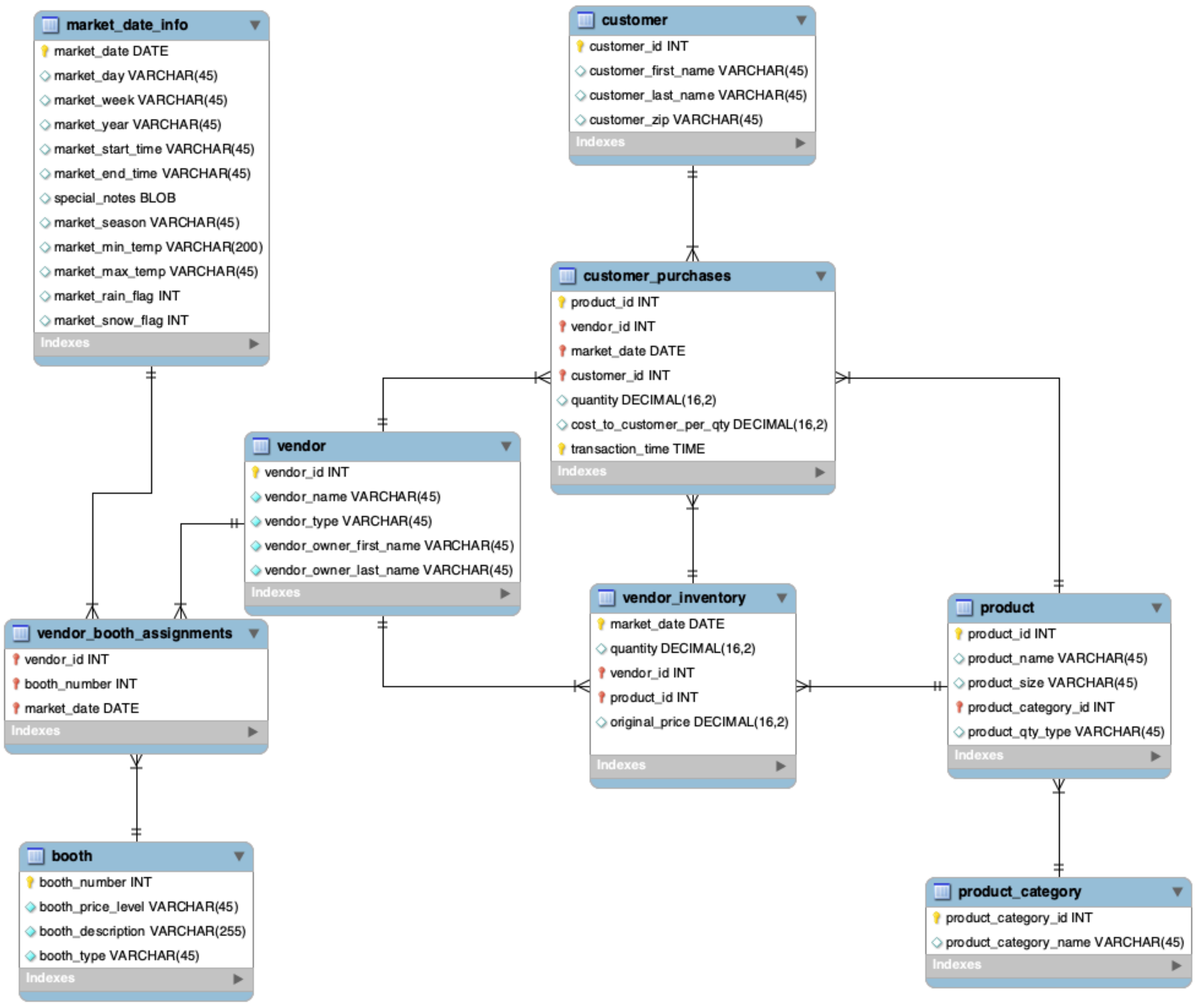
Problem Statement:

- Amazon Farmers Market (AFM), a beloved community market, faces a growing threat from large grocery chains that are expanding their local produce offerings. To maintain its customer base and thrive in this competitive environment, AFM needs to leverage data analytics for a deeper understanding of:
    - **Customer Behavior:** Analyze purchase history, spending habits, and visit frequency to identify customer segments, preferences, and potential churn risks.
    - **Vendor Performance:** Analyze sales data and customer attraction metrics to identify top-performing vendors, evaluate product popularity, and optimize vendor selection.

    - **Market Operations**: Analyze traffic trends and seasonal sales patterns to optimize inventory management, staffing levels, and marketing campaigns.
    - **Inventory management:** Analyze future demand for specific products based on historical sales data to optimize the right amount of inventory to meet customer needs while minimizing waste and storage costs.

- As you delve further, you encounter additional scenarios requiring more advanced filtering and subqueries to extract meaningful insights from the data.
- AFM aims to leverage the data analytics skills that you have achieved till now to gain a competitive advantage, optimize inventory management, tailor marketing efforts, improve the customer experience for long-term success and solidify AFM's position as a vital community hub for fresh, local produce

Dataset: Farmer’s Market database



Formulating questions to be explored based on the data provided:

- **Customer Behavior:**
  - Your manager wants to see all the unique customer IDs present in the `customer_purchases` table. How would you get this data?

- **Vendor Performance:**
  - Find out which vendors primarily sell fresh products and which don't.

    - What if we want to add 1 for vendors who sell fresh products and 0 for those who don’t?

- **Market Operations**:
    - Analyze purchases made at the market on days when it rained.
    - Put the total cost of customer purchases into different bins.
        - under $5.00,
        - $5.00–$9.99,
        - $10.00–$19.99, or
        - $20.00 and over.

- **Inventory management:**
    - Find all of the products from the `product` table that don’t have their sizes mentioned.
    - What if you’re asked to fetch all the product IDs from the `products` table and set a default value “medium” in rows where the product size is NULL?
    - List down all the product details where product_category_name contains “Fresh” in it.

While exploring the dataset there can be situations when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only the unique ones.

In that case, you can use the **DISTINCT** keyword in conjunction with the SELECT statement to **eliminate all the duplicate rows**.

Syntax:

```
SELECT DISTINCT col1, col2, …
FROM DB.table
```

Even if you want to fetch the **unique** values from a single column, you can do so using the DISTINCT keyword.

Let's have a look at an example:

Questions: Your manager wants to see all the **unique** customer IDs present in the `customer_purchases` table. How would you get this data?

```
SELECT
DISTINCT customer_id
FROM farmers_market.customer_purchases
```

Order of Execution of a SQL query :

- **FROM** - The database gets the data from tables in the FROM clause.
- **WHERE** - The data is filtered based on the conditions specified in the WHERE clause. Rows that do not meet the criteria are excluded.
- **SELECT** - After filtering is done, the SELECT statement determines which columns to include in the final result set.
- **DISTINCT** - The DISTINCT keyword is applied within the SELECT clause to ensure that only unique values are returned for the specified columns.
- **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.

Now as you explore further, you come across some fields that have some missing values in them and you don't know how to deal with and extract these missing values from the data.

Let’s understand it with the help of an example:

Question: Find all of the products from the `product` table that don’t have their sizes mentioned.

The absence of any value in a cell is represented by the **NULL** keyword.

You can use the **IS NULL** operators in a WHERE clause to filter rows that have missing or null values in a specific column.

```
SELECT *
FROM farmers_market.product
WHERE product_size IS NULL
```

NOTE: Keep in mind that **“blank”** and **NULL** are not the same thing in database terms.

- If someone asked you to find all products that didn’t have product sizes, you might also want to check for blank strings, which would equal “”, or rows where someone entered a space or any number of spaces into that field.
- The **TRIM()** function **removes excess spaces from the beginning or end of a string value**, so if you use a combination of the TRIM() function and blank string comparison, you can find any row that is blank or contains only spaces.

```
SELECT *
FROM farmers_market.product
WHERE
    product_size IS NULL
    OR TRIM(product_size) = “”
```

- You might wonder why the comparison operator IS NULL is used instead of equals NULL just like numbers.
- **NULL is not actually a value, it’s the absence of a value, so it can’t be compared to any existing value.**
- If you wanted to return all records that don’t have NULL values in a field, you could use the condition “[field name] **IS NOT NULL**” in the WHERE clause.

```
SELECT *
FROM farmers_market.product
WHERE
    product_size IS NOT NULL
```

Now let’s say your manager wants to replace the null values with some other values then how will you achieve it? Let’s understand it with the help of an example.

Question: What if you’re asked to fetch all the product IDs from the `products` table and set a default value “medium” in rows where the product size is NULL?

We can use the **IFNULL()** function to check for null values in a column and if found, replace them with some other value.

```
SELECT
   product_id,
   IFNULL(product_size,  “medium”)
FROM farmers_market.product
```

Subqueries

- A Subquery is a SQL query embedded within the WHERE clause of another SQL query.
- A subquery is also called an **inner/child** query, while the statement containing a subquery is also called an **outer/parent** query.
- A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.
- Note that the **inner query executes first before its parent query** so that the results of an inner query can be passed to the outer query.

Question: Analyze purchases made at the market on days when it rained.

There is a value in the *market_date_info* **table** called *market_rain_flag* that has a value of 0 if it didn't rain while the market was open and a value of 1 if it did.

- 0 - it didn't rain
- 1 - it did

First, let's write a query that gets a list of market dates when it rained, using this query:

```
SELECT market_date, market_rain_flag
 FROM farmers_market.market_date_info
 WHERE market_rain_flag = 1
```

Now let's use the list of dates generated by that query to return purchases made on those dates.

```
SELECT *
```

```
        FROM `farmers_market.customer_purchases`
        WHERE market_date IN (SELECT market_date
                                FROM `farmers_market.market_date_info`
                                WHERE market_rain_flag = 1 )
```

Question: List down all the product details where product_category_name contains “Fresh” in it.

- Capture the list category_ids where the category_name contains “Fresh”
- Get the products that belong to the shortlisted category_ids

```
SELECT *
FROM `farmers_market.product`
WHERE product_category_id IN ( SELECT product_category_id
                                FROM
                                `farmers_market.product_category`
                                WHERE LOWER(product_category_name)
                                LIKE "%fresh%")
```

Similarly, you could use the **NOT IN** operator if you want to get the product details where the product_category_name does not contain “Fresh”.

CASE statement

What if you want to take some action based on a certain condition?

For example -

“If [one condition] is true, then [take this action].
Otherwise, [take this other action].”

That’s where CASE statements come into play.

Syntax:

```
SELECT cols,
CASE
        WHEN [first conditional statement]
         THEN [value or calculation]
        WHEN [second conditional statement]
         THEN [value or calculation]
        ELSE [value or calculation]
END AS alias
FROM table
```

IF() function

Syntax:
IF(condition, true_value, false_value)

Parameters used:

- **condition** – It is used to specify the condition to be evaluated.
- **true_value** – It is an optional parameter that is used to specify the value to be returned if the condition evaluates to be true.
- **false_value** – It is an optional parameter that is used to specify the value to be returned if the condition evaluates to be false.

Let’s have a look at a few examples:

Question: Find out which vendors primarily sell fresh products and which don’t.

You have to add **“Fresh Produce”** for vendors who sell fresh products and **“Other”** for those who don’t.

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

We can do the same using the **IF() function** as well -

```
SELECT *,
IF(LOWER(vendor_type) LIKE “%fresh%”, “Fresh”, “Not Fresh”) AS
type FROM `farmers_market.vendor`
```

What if we want to add **1** for vendors who sell fresh products and **0** for those who don’t?

```
SELECT vendor_id, vendor_name, vendor_type,
CASE
WHEN LOWER(vendor_type) LIKE "%fresh%" THEN 1
ELSE 0
END AS category
FROM `farmers_market.vendor`
```

We can do the same using the **IF() function** as well -

```
        SELECT *,
        IF(LOWER(vendor_type) LIKE “%fresh%”, 1, 0) AS type FROM
        `farmers_market.vendor`
```

Now what if we want to categorize the customer based on their total purchases, we can also achieve this by using the CASE function.

Question: Put the total cost of customer purchases into bins of -
- under $5.00,
- $5.00–$9.99,
- $10.00–$19.99, or
- $20.00 and over.

```
SELECT
        market_date, customer_id,
        quantity, cost_to_customer_per_qty,
CASE
  WHEN quantity * cost_to_customer_per_qty < 5
  THEN "Under $5"
  WHEN quantity * cost_to_customer_per_qty BETWEEN 5 AND 9.99
  THEN "$5 - $9.99"
  WHEN quantity * cost_to_customer_per_qty BETWEEN 10 AND 19.99
  THEN "$10 - $19.99"
  ELSE "Above $20"
END AS price_bin
FROM `farmers_market.customer_purchases`
```

Insights and Recommendations:

1. **Missing Records analysis:**

Missing size information can lead to operational inefficiencies and customer dissatisfaction.

- Recommendation: Update product records with missing size data promptly to ensure accurate inventory management and customer expectations.

2. **Setting Default Value in place of NULL values:**

   Setting default values ensures consistency in data analysis and operations.

   - Recommendation: Regularly review and update data integrity protocols to minimize errors in product information.

3. **Fresh Product Focus:**

   Fresh products are likely to be high-demand items in a market focused on local produce.

   - Recommendation: Highlight these products prominently in marketing campaigns to appeal to health-conscious and environmentally-aware customers.

4. **Rainy Day Purchase Analysis:**

   Weather impacts consumer behaviour, affecting attendance and purchase patterns.

   - Recommendation: Use weather data to optimize staffing, inventory levels, and promotional strategies during inclement weather to maintain customer satisfaction.

5. **Customer Spending Segmentation:**

   Binning purchase amounts helps in understanding spending patterns and customer segments.

   - Recommendation: Tailor marketing offers and product promotions based on spending behaviours observed in each price category

6. **Vendor Product Differentiation:**

   Differentiating vendors based on product offerings informs market strategy and vendor relations.

   - Recommendation: Encourage vendors with fresh offerings to highlight their products prominently, possibly with incentives, to attract health-conscious consumers.

7. **Customer ID Analysis:**

   Understanding unique customer IDs supports customer segmentation and analytics.

   - Recommendation: Regularly update and maintain customer databases to ensure accurate reporting and targeted marketing efforts.

Overall Summary:

- **Data Analytics for Competitive Advantage:**
  - Leveraging data analytics across customer behaviour, vendor performance, market operations, and inventory management enables AFM to optimize operations, enhance customer experience, and stay competitive against larger grocery chains.

- **Recommendations:**

    - Foster vendor partnerships based on performance metrics, and personalize customer interactions to solidify AFM’s position as a community-centric market for fresh produce. Regular data audits and updates ensure data integrity and effectiveness of business strategies.
