Databases and SQL for Data Science with Python

19/08/2024

Window Functions and Subqueries

Window Functions

All previously covered functions, like ROUND(), return one value per row in the result set. When using GROUP BY, functions like AVG() operate on multiple values within an aggregated group of records, summarizing across multiple rows but returning one value per row in the results.

Window functions also work across multiple records but do not require grouping in the output. They provide context by comparing a row's values to a group of rows (a partition), enabling queries to answer questions like the row's position if sorted, or comparing values between rows.

Window functions return group aggregate calculations alongside individual row-level data within that group or partition and can rank or sort values within each partition. In data science, window functions can include information from past records alongside current records.

For example, they can retrieve the date of a customer's first purchase to determine how long they have been a customer at each subsequent purchase.

Window Functions

Based on previous knowledge; to find the most expensive product sold by each vendor, you can group the records in the `vendor_inventory` table by `vendor_id` and return the maximum `original_price` value using the following query:

```
SELECT
    vendor_id,
    MAX(original_price) AS highest_price
FROM farmers_market.vendor_inventory
GROUP BY vendor_id
ORDER BY vendor_id
```

vendor_id	highest_price
4	0.50
7	6.99
8	18.00

However, this only provides the price of the most expensive item per vendor. To determine which product is the most expensive, how would you identify the `product_id` associated with the `MAX(original_price)` per vendor?

Row Number

There is a window function that enables you to rank rows by a value—in this case, ranking products per vendor by price—called ROW_NUMBER().

```
SELECT
    vendor_id,
    market_date,
    product_id,
    original_price,
    ROW_NUMBER() OVER (PARTITION BY vendor_id ORDER BY original_price DESC) AS
price_rank
FROM farmers_market.vendor_inventoryORDER BY vendor_id, original_price DESC
```

number the rows of inventory per vendor, sorted by original price, in descending order

like a GROUP BY without actually combining the rows, so we're telling it how to split the rows into groups, without aggregating

Row Number

For each vendor, the products are sorted by original_price, high to low, and the row numbering column is called price_rank. The row numbering starts over when you get to the next vendor_id, so the most expensive item per vendor has a price_rank of 1.

vendor_id	market_date	product_id	original_price	price_rank
1	2019-03-20	11	13.00	1
1	2019-03-02	11	12.00	2
1	2019-03-09	11	12.00	3
1	2019-03-02	10	6.00	4
1	2019-03-09	10	6.00	5
4	2019-03-16	9	2.00	4
4	2019-03-02	9	2.00	1
4	2019-03-09	9	2.00	2
4	2019-03-13	9	2.00	3
7	2019-03-09	13	6.00	1
7	2019-03-20	13	6.00	2
7	2019-03-23	13	6.00	3
7	2019-03-30	13	6.00	4
7	2019-03-20	12	3.00	6
7	2019-03-23	12	3.00	7
7	2019-03-30	12	3.00	8
7	2019-03-09	12	3.00	5

روّاد مصر الرقمية

وزارة الاتصـــالات وتكنولوجيا المعلومات

Row Number

To return only the record of the highest-priced item per vendor, you can query the results of the previous query (which is called a subquery) and limit the output to the #1 ranked item per vendor_id.

```
SELECT * FROM
(
    SELECT
        vendor_id,
        market_date,
        product_id,
        original_price,
        ROW_NUMBER() OVER (PARTITION BY vendor_id ORDER BY original_price DESC) AS
price_rank
    FROM farmers_market.vendor_inventory
    ORDER BY vendor_id) x
WHERE x.price_rank = 1
```

vendor_id	market_date	product_id	original_price	price_rank
1	2019-03-20	11	13.00	1
4	2019-03-02	9	2.00	1
7	2019-03-09	13	6.00	1
8	2019-03-02	4	4.00	1
9	2019-03-09	7	18.00	1

RANK and DENSE RANK

Two other window functions are very similar to ROW_NUMBER and have the same syntax but provide slightly different results.

The RANK function numbers the results just like ROW_NUMBER does but gives rows with the same value the same ranking.

```
SELECT
    vendor_id,
    market_date,
    product_id,
    original_price,
    RANK() OVER (PARTITION BY vendor_id ORDER BY original_price DESC) AS
price_rank
    FROM farmers_market.vendor_inventory
ORDER BY vendor_id, original_price DESC
```

If you don't want to skip numbers in your ranking when there is a tie use the DENSE_RANK function.

NTILE

But what if you were asked to return the “top tenth” of the inventory, when sorted by price?

With NTILE, you specify a number inside the parentheses, NTILE(n), to indicate that you want the results broken up into n blocks.

```
SELECT
    vendor_id,
    market_date,
    product_id,
    original_price,
    NTILE(10) OVER (ORDER BY original_price DESC) AS price_ntile
    FROM farmers_market.vendor_inventory
ORDER BY original_price DESC
```

Aggregate Window Functions

We can use the AVG() function as a window function, partitioned by market_date, and compare each product's price to that value.

vendor_id	market_date	product_id	original_price	average_cost_product_by_market_date
8	2019-03-02	4	4.00	6.000000
4	2019-03-02	9	2.00	6.000000
1	2019-03-02	10	6.00	6.000000
1	2019-03-02	11	12.00	6.000000
8	2019-03-09	4	4.00	8.222222
9	2019-03-09	5	5.00	8.222222
9	2019-03-09	7	18.00	8.222222
9	2019-03-09	8	18.00	8.222222
4	2019-03-09	9	2.00	8.222222
1	2019-03-09	10	6.00	8.222222
1	2019-03-09	11	12.00	8.222222
7	2019-03-09	12	3.00	8.222222
7	2019-03-09	13	6.00	8.222222
8	2019-03-13	4	4.00	3.000000
4	2019-03-13	9	2.00	3.000000

```
    SELECT
        vendor_id,
        market_date,
        product_id,
        original_price,
        AVG(original_price) OVER (PARTITION BY market_date ORDER BY
market_date)
            AS average_cost_product_by_market_date
    FROM farmers_market.vendor_inventory
```

Aggregate Window Functions

Now, let's wrap that query inside another query (use it as a subquery) so we can compare the original price per item to the average cost of products on each market date that has been calculated by the window function.

Using a subquery, we can filter the results to a single vendor, with vendor_id 1, and only display products that have prices above the market date's average product cost.

```
SELECT * FROM
(
    SELECT
        vendor_id,
        market_date,
        product_id,
        original_price,
        ROUND(AVG(original_price) OVER (PARTITION BY market_date ORDER BY
market_date), 2)
            AS average_cost_product_by_market_date
    FROM farmers_market.vendor_inventory
) x
WHERE x.vendor_id = 1
    AND x.original_price > x.average_cost_product_by_market_date
ORDER BY x.market_date, x.original_price DESC
```

vendor_id	market_date	product_id	original_price	average_cost_product_by_market_date
1	2019-03-02	11	12.00	6.00
1	2019-03-09	11	12.00	8.22
1	2019-03-20	11	13.00	7.33

Aggregate Window Functions

Another use of an aggregate window function is to count how many items are in each partition.

vendor_id	market_date	product_id	original_price	vendor_product_count_per_market_date
1	2019-03-02	11	12.00	2
1	2019-03-02	10	6.00	2
1	2019-03-09	11	12.00	2
1	2019-03-09	10	6.00	2
1	2019-03-20	11	13.00	1
4	2019-03-02	9	2.00	1
4	2019-03-09	9	2.00	1
4	2019-03-13	9	2.00	1
4	2019-03-16	9	2.00	1

```
SELECT
    vendor_id,
    market_date,
    product_id,
    original_price,
    COUNT(product_id) OVER (PARTITION BY market_date, vendor_id)
vendor_product_count_per_market_date
    FROM farmers_market.vendor_inventory
ORDER BY vendor_id, market_date, original_price DESC
```

Aggregate Window Functions

You can also use aggregate window functions to calculate running totals. In the query shown, we're not using a PARTITION BY clause, so the running total of the price is calculated across the entire results set:

customer_id	market_date	vendor_id	product_id	price	running_total_purchases
9	2019-04-03	8	8	36.0000	36.0000
9	2019-04-03	8	7	18.0000	54.0000
9	2019-04-03	8	5	6.5000	60.5000
9	2019-04-03	8	7	36.0000	96.5000
6	2019-04-03	8	5	6.5000	103.0000
9	2019-04-03	8	5	6.5000	109.5000
23	2019-04-03	8	7	18.0000	127.5000

```
SELECT customer_id,
    market_date,
    vendor_id,
    product_id,
    quantity * cost_to_customer_per_qty AS price,
    SUM(quantity * cost_to_customer_per_qty) OVER (ORDER BY market_date,
transaction_time, customer_id, product_id) AS running_total_purchases
FROM farmers_market.customer_purchases
```

Aggregate Window Functions

In this next query, we are calculating the same running total, but it is partitioned by customer_id. That means that each time we get to a new customer_id, the running total resets.

customer_id	market_date	vendor_id	product_id	price	customer_spend_running_total
1	2019-03-09	9	8	18.0000	58.9000
1	2019-03-09	1	10	11.0000	69.9000
1	2019-03-09	7	13	12.6500	82.5500
1	2019-03-09	7	13	NULL	82.5500
1	2019-03-20	7	13	17.8250	100.3750
2	2019-03-02	4	9	9.2000	9.2000
2	2019-03-13	4	9	8.2000	17.4000
2	2019-03-13	8	4	8.0000	25.4000
3	2019-03-02	4	9	16.8000	16.8000
3	2019-03-16	9	8	18.0000	34.8000
3	2019-03-16	4	9	11.0000	45.8000

```
SELECT customer_id,
    market_date,
    vendor_id,
    product_id,
    quantity * cost_to_customer_per_qty AS price,
    SUM(quantity * cost_to_customer_per_qty) OVER (PARTITION BY
customer_id ORDER BY market_date, transaction_time, product_id) AS
customer_spend_running_total
FROM farmers_market.customer_purchases
```

Aggregate Window Functions

What do you expect to happen when there is only a PARTITION BY clause (and no ORDER BY clause)?

customer_id	market_date	vendor_id	product_id	price	customer_spend_total
1	2019-03-09	9	8	18.0000	100.3750
1	2019-03-02	1	10	5.5000	100.3750
1	2019-03-02	1	10	15.0000	100.3750
1	2019-03-09	1	10	11.0000	100.3750
1	2019-03-02	1	11	20.4000	100.3750
1	2019-03-09	7	13	12.6500	100.3750
1	2019-03-09	7	13	NULL	100.3750
1	2019-03-20	7	13	17.8250	100.3750
2	2019-03-13	8	4	8.0000	25.4000
2	2019-03-02	4	9	9.2000	25.4000
2	2019-03-13	4	9	8.2000	25.4000
3	2019-03-16	9	8	18.0000	45.8000
3	2019-03-02	4	9	16.8000	45.8000
3	2019-03-16	4	9	11.0000	45.8000

```
SELECT customer_id,
    market_date,
    vendor_id,
    product_id,
    ROUND(quantity * cost_to_customer_per_qty, 2) AS price,
    ROUND(SUM(quantity * cost_to_customer_per_qty) OVER (PARTITION BY
customer_id), 2) AS customer_spend_total
FROM farmers_market.customer_purchases
```

Without ORDER BY, the SUM is calculated across the entire partition, instead of as a per-row running total.

LAG and LEAD

Using the vendor_booth_assignments table in the Farmer's Market database, we can display each vendor's booth assignment for each market_date alongside their previous booth assignments using the LAG() function.

LAG retrieves data from a row that is a selected number of rows back in the dataset. You can set the number of rows (offset) to any integer value x to count x rows backwards, following the sort order specified in the ORDER BY section of the window function:

```
SELECT
    market_date,
    vendor_id,
    booth_number,
    LAG(booth_number,1) OVER (PARTITION BY vendor_id ORDER BY market_date,
vendor_id) AS previous_booth_number
FROM farmers_market.vendor_booth_assignments
ORDER BY market_date, vendor_id, booth_number
```

LAG and LEAD

In this case, for each vendor_id for each market_date, we're pulling the booth_number the vendor had 1 market date in the past.

market_date	vendor_id	booth_number	previous_booth_number
2019-04-03	3	1	NULL
2019-04-03	4	7	NULL
2019-04-03	7	11	NULL
2019-04-03	8	6	NULL
2019-04-03	9	8	NULL
2019-04-06	1	2	2
2019-04-06	3	1	1
2019-04-06	4	7	7
2019-04-06	7	11	11
2019-04-06	8	6	6
2019-04-06	9	8	8
2019-04-10	1	7	2
2019-04-10	3	1	1
2019-04-10	4	2	7
2019-04-10	7	11	11
2019-04-10	8	6	6
2019-04-10	9	8	8

LAG and LEAD

We will create this report by wrapping the query with the LAG function in another query, which we can use to filter the results to a market_date and vendors whose current booth_number is different from their previous_booth_number:

```
SELECT * FROM
(
    SELECT
        market_date,
        vendor_id,
        booth_number,
        LAG(booth_number,1) OVER (PARTITION BY vendor_id ORDER BY market_
date, vendor_id) AS previous_booth_number
        FROM farmers_market.vendor_booth_assignments
        ORDER BY market_date, vendor_id, booth_number
) x
WHERE x.market_date = '2019-04-10'
        AND (x.booth_number <> x.previous_booth_number OR x.previous_
booth_number IS NULL)
```

market_date	vendor_id	booth_number	previous_booth_number
2019-04-10	4	2	7
2019-04-10	1	7	2

LAG and LEAD

To show another example use case, let’s say we want to find out if the total sales on each market date are higher or lower than they were on the previous market date.

In this example we will add in a GROUP BY function. The window functions are calculated after the grouping and aggregation occurs.

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

market_date	market_date_total_sales
2019-03-02	81.7000
2019-03-09	171.4750
2019-03-13	16.2000
2019-03-16	51.0000
2019-03-20	26.8250
2019-03-23	22.8000
2019-03-30	3.0000

LAG and LEAD

Then, we can add the LAG() window function to output the previous market_date's calculated sum on each row.

```
SELECT
    market_date,
    SUM(quantity * cost_to_customer_per_qty) AS market_date_total_sales,
    LAG(SUM(quantity * cost_to_customer_per_qty), 1) OVER (ORDER BY
market_date) AS previous_market_date_total_sales
FROM farmers_market.customer_purchases
GROUP BY market_date
ORDER BY market_date
```

market_date	market_date_total_sales	previous_market_date_total_sales
2019-03-02	81.7000	NULL
2019-03-09	171.4750	81.7000
2019-03-13	16.2000	171.4750
2019-03-16	51.0000	16.2000
2019-03-20	26.8250	51.0000
2019-03-23	22.8000	26.8250
2019-03-30	3.0000	22.8000

LAG and LEAD

LEAD works the same way as LAG, but it gets the value from the next row instead of the previous row (assuming the offset integer is 1).

You can set the offset integer to any value x to count x rows forward, following the sort order specified in the ORDER BY section of the window function.

If the rows are sorted by a time value, LAG would be retrieving data from the past, and LEAD would be retrieving data from the future (relative to the current row).

These values can also now be used in calculations; for example, to determine the change in sales week to week.

Exercises

1. Do the following two steps:

   a. Write a query that selects from the customer_purchases table and numbers each customer's visits to the farmer's market (labeling each market date with a different number). Each customer's first visit is labeled 1, second visit is labeled 2, etc. (We are of course not counting visits where no purchases are made, because we have no record of those.) You can either display all rows in the customer_purchases table, with the counter changing on each new market date for each customer, or select only the unique market dates per customer (without purchase details) and number those visits.
   HINT: One of these approaches uses ROW_NUMBER() and one uses DENSE_RANK().

   b. Reverse the numbering of the query from a part so each customer's most recent visit is labeled 1, then write another query that uses this one as a subquery and filters the results to only the customer's most recent visit.

Exercises

2. Using a COUNT() window function, include a value along with each row of the customer_purchases table that indicates how many different times that customer has purchased that product_id.

3. In the last query we used LAG and sorted by market_date. Can you think of a way to use LEAD in place of LAG, but get the exact same output?

Date and Time Functions

Setting datetime Field Values

The market_date_info table lacks datetime fields, so to demonstrate date and time functions without repeatedly combining fields, I'll create a demo table by merging the market_date and market_start_time fields using this query:

```
CREATE TABLE farmers_market.datetime_demo AS
(
    SELECT market_date,
        market_start_time,
        market_end_time,
        STR_TO_DATE(CONCAT(market_date, ' ', market_start_time), '%Y-%m-%d
%h:%i %p')
            AS market_start_datetime,
        STR_TO_DATE(CONCAT(market_date, ' ', market_end_time), '%Y-%m-%d
%h:%i %p')
            AS market_end_datetime
    FROM farmers_market.market_date_info
)
```

Setting datetime Field Values

Data will look like:

market_date	market_start_time	market_end_time	market_start_datetime	market_end_datetime
2019-03-02	8:00 AM	2:00 PM	2019-03-02 08:00:00	2019-03-02 14:00:00
2019-03-09	9:00 AM	2:00 PM	2019-03-09 09:00:00	2019-03-09 14:00:00
2019-03-13	4:00 PM	7:00 PM	2019-03-13 16:00:00	2019-03-13 19:00:00
2019-03-16	8:00 AM	2:00 PM	2019-03-16 08:00:00	2019-03-16 14:00:00
2019-03-20	4:00 PM	7:00 PM	2019-03-20 16:00:00	2019-03-20 19:00:00
2019-03-23	8:00 AM	2:00 PM	2019-03-23 08:00:00	2019-03-23 14:00:00
2019-03-27	4:00 PM	7:00 PM	2019-03-27 16:00:00	2019-03-27 19:00:00
2019-03-30	8:00 AM	2:00 PM	2019-03-30 08:00:00	2019-03-30 14:00:00

EXTRACT and DATE_PART

Using datetime values established in the datetime_demo table created in the previous section, we can EXTRACT date and time parts from the fields.

The following query demonstrates five different “date parts” that can be extracted from the datetime:

```
SELECT market_start_datetime,
    EXTRACT(DAY FROM market_start_datetime) AS mktsrt_day,
    EXTRACT(MONTH FROM market_start_datetime) AS mktsrt_month,
    EXTRACT(YEAR FROM market_start_datetime) AS mktsrt_year,
    EXTRACT(HOUR FROM market_start_datetime) AS mktsrt_hour,
    EXTRACT(MINUTE FROM market_start_datetime) AS mktsrt_minute
 FROM farmers_market.datetime_demo
 WHERE market_start_datetime = '2019-03-02 08:00:00'
```

market_start_datetime	mktsrt_day	mktsrt_month	mktsrt_year	mktsrt_hour	mktsrt_minute
2019-03-02 08:00:00	2	3	2019	8	0

EXTRACT and DATE_PART

There are also shortcuts for extracting the entire date and entire time from the datetime field, so you don't have to extract each part and re-concatenate it together.

```
SELECT market_start_datetime,
    DATE(market_start_datetime) AS mktsrt_date,
    TIME(market_start_datetime) AS mktsrt_time
FROM farmers_market.datetime_demo
WHERE market_start_datetime = '2019-03-02 08:00:00'
```

market_start_datetime	mktsrt_date	mktsrt_time
2019-03-02 08:00:00	2019-03-02	08:00:00

DATE_ADD and DATE_SUB

To find how many sales occurred within the first 30 minutes of the market opening, how would you dynamically calculate the cutoff time for each market date? The DATE_ADD function is key here.

```
SELECT market_start_datetime,
    DATE_ADD(market_start_datetime, INTERVAL 30 MINUTE) AS mktstrt_date_
plus_30min
FROM farmers_market.datetime_demo
WHERE market_start_datetime = '2019-03-02 08:00:00'
```

market_start_datetime	mktstrt_date_plus_30min
2019-03-02 08:00:00	2019-03-02 08:30:00

DATE_ADD and DATE_SUB

If we instead wanted to do a calculation that required looking 30 days past a date:

```
SELECT market_start_datetime,
    DATE_ADD(market_start_datetime, INTERVAL 30 DAY) AS mktstrt_date_
plus_30days
FROM farmers_market.datetime_demo
WHERE market_start_datetime = '2019-03-02 08:00:00'
```

market_start_datetime	mktstrt_date_plus_30days
2019-03-02 08:00:00	2019-04-01 08:00:00

DATE_ADD and DATE_SUB

There is also a related function called DATE_SUB() that subtracts intervals from datetimes. However, instead of switching to DATE_SUB(), you could also just add a negative number to the datetime if you prefer.

```
SELECT market_start_datetime,
    DATE_ADD(market_start_datetime, INTERVAL -30 DAY) AS mktstrt_date_
plus_neg30days,
    DATE_SUB(market_start_datetime, INTERVAL 30 DAY) AS mktstrt_date_
minus_30days
FROM farmers_market.datetime_demo
WHERE market_start_datetime = '2019-03-02 08:00:00'
```

market_start_datetime	mktstrt_date_plus_neg30days	mktstrt_date_minus_30days
2019-03-02 08:00:00	2019-01-31 08:00:00	2019-01-31 08:00:00

DATEDIFF

DATEDIFF is a SQL function available in most database systems that accepts two dates or datetime values, and returns the difference between them in days.

Here, the inner query returns the first and last market dates from the datetime_demo table, and the outer query (which is selecting from “x”) calculates the difference between those two dates using DATEDIFF.

```
SELECT
    x.first_market,
    x.last_market,
    DATEDIFF(x.last_market, x.first_market) days_first_to_last
FROM
(
    SELECT
        min(market_start_datetime) first_market,
        max(market_start_datetime) last_market
    FROM farmers_market.datetime_demo
) x
```

first_market	last_market	days_first_to_last
2019-03-02 08:00:00	2019-03-30 08:00:00	28

TIMESTAMPDIFF

The DATEDIFF function returns the difference in days, but there is also a function in MySQL called TIMESTAMPDIFF that returns the difference between two datetimes in any chosen interval.

Here, we calculate the hours and minutes between the market start and end times on each market date.

```
SELECT market_start_datetime, market_end_datetime,
    TIMESTAMPDIFF(HOUR, market_start_datetime, market_end_datetime)
        AS market_duration_hours,
    TIMESTAMPDIFF(MINUTE, market_start_datetime, market_end_datetime)
        AS market_duration_mins
 FROM farmers_market.datetime_demo
```

market_start_datetime	market_end_datetime	market_duration_hours	market_duration_mins
2019-03-02 08:00:00	2019-03-02 14:00:00	6	360
2019-03-09 09:00:00	2019-03-09 14:00:00	5	300
2019-03-13 16:00:00	2019-03-13 19:00:00	3	180
2019-03-16 08:00:00	2019-03-16 14:00:00	6	360
2019-03-20 16:00:00	2019-03-20 19:00:00	3	180
2019-03-23 08:00:00	2019-03-23 14:00:00	6	360
2019-03-27 16:00:00	2019-03-27 19:00:00	3	180
2019-03-30 08:00:00	2019-03-30 14:00:00	6	360

Date Functions in Aggregate Summaries and Window Functions

To profile each farmer's market customer's habits over time, we'll group results by customer and include date-related summaries.

Let's start by retrieving the purchases, focusing on the dates for customer_id 1:

```
SELECT customer_id, market_date
FROM farmers_market.customer_purchases
WHERE customer_id = 1
```

customer_id	market_date
1	2019-03-09
1	2019-03-02
1	2019-03-02
1	2019-03-09
1	2019-03-02
1	2019-03-09
1	2019-03-09
1	2019-03-20

Date Functions in Aggregate Summaries and Window Functions

Let's summarize this data and get their earliest purchase date, latest purchase date, and number of different days on which they made a purchase.

```
SELECT customer_id,
    MIN(market_date) AS first_purchase,
    MAX(market_date) AS last_purchase,
    COUNT(DISTINCT market_date) AS count_of_purchase_dates
FROM farmers_market.customer_purchases
WHERE customer_id = 1
GROUP BY customer_id
```

customer_id	first_purchase	last_purchase	count_of_purchase_dates
1	2019-03-02	2019-03-20	3

Date Functions in Aggregate Summaries and Window Functions

To find how long this person has been a customer, we can calculate the difference between their first and last purchase.

```
SELECT customer_id,
    MIN(market_date) AS first_purchase,
    MAX(market_date) AS last_purchase,
    COUNT(DISTINCT market_date) AS count_of_purchase_dates,
    DATEDIFF(MAX(market_date), MIN(market_date)) AS days_between_first_
last_purchase
FROM farmers_market.customer_purchases
GROUP BY customer_id
```

customer_id	first_purchase	last_purchase	count_of_purchase_dates	days_between_first_last_purchase
1	2019-03-02	2019-03-20	3	18
2	2019-03-02	2019-03-13	2	11
3	2019-03-02	2019-03-16	2	14
4	2019-03-02	2019-03-23	3	21
5	2019-03-09	2019-03-09	1	0
7	2019-03-09	2019-03-20	2	11
10	2019-03-02	2019-03-16	2	14
12	2019-03-09	2019-03-30	3	21

Date Functions in Aggregate Summaries and Window Functions

If we wanted to also know how long it's been since the customer last made a purchase, we can use the CURDATE() function:

```
SELECT customer_id,
    MIN(market_date) AS first_purchase,
    MAX(market_date) AS last_purchase,
    COUNT(DISTINCT market_date) AS count_of_purchase_dates,
    DATEDIFF(MAX(market_date), MIN(market_date)) AS days_between_first_
last_purchase,
    DATEDIFF(CURDATE(), MAX(market_date)) AS days_since_last_purchase
FROM farmers_market.customer_purchases
GROUP BY customer_id
```

Date Functions in Aggregate Summaries and Window Functions

We can also write a query that gives us the days between each purchase a customer makes.

Let's go back to customer 1's detailed purchases and use both the RANK and LEAD window functions to retrieve each purchase date, along with the next purchase date:

```
SELECT customer_id, market_date,
    RANK() OVER (PARTITION BY customer_id ORDER BY market_date) AS
purchase_number,
    LEAD(market_date,1) OVER (PARTITION BY customer_id ORDER BY market_
date) AS next_purchase
FROM farmers_market.customer_purchases
WHERE customer_id = 1
```

customer_id	market_date	purchase_number	next_purchase
1	2019-03-02	1	2019-03-02
1	2019-03-02	1	2019-03-02
1	2019-03-02	1	2019-03-09
1	2019-03-09	4	2019-03-09
1	2019-03-09	4	2019-03-09
1	2019-03-09	4	2019-03-09
1	2019-03-09	4	2019-03-20
1	2019-03-20	8	NULL

Date Functions in Aggregate Summaries and Window Functions

We didn't achieve the goal of showing time between each purchase date because multiple rows exist for the same date when a customer bought multiple items.

The solution is to remove duplicates, use a subquery to get date differences, and move window functions to the outer query to correctly count purchase dates instead of each item.

```
SELECT
    x.customer_id,
    x.market_date,
    RANK() OVER (PARTITION BY x.customer_id ORDER BY x.market_date) AS
purchase_number,
    LEAD(x.market_date,1) OVER (PARTITION BY x.customer_id ORDER BY
x.market_date) AS next_purchase
FROM
   (
      SELECT DISTINCT customer_id, market_date
      FROM farmers_market.customer_purchases
      WHERE customer_id = 1
   ) x
```

Date Functions in Aggregate Summaries and Window Functions

And we can now add a line to the query to use that next_purchase date in a DATEDIFF calculation:

```
SELECT
    x.customer_id,
    x.market_date,
    RANK() OVER (PARTITION BY x.customer_id ORDER BY x.market_date)
        AS purchase_number,
    LEAD(x.market_date,1) OVER (PARTITION BY x.customer_id ORDER BY
x.market_date) AS next_purchase,
    DATEDIFF(
        LEAD(x.market_date,1) OVER
        (PARTITION BY x.customer_id ORDER BY x.market_date),
        x.market_date
        ) AS days_between_purchases
FROM
(
    SELECT DISTINCT customer_id, market_date
    FROM farmers_market.customer_purchases
    WHERE customer_id = 1
) x
```

customer_id	market_date	purchase_number	next_purchase	days_between_purchases
1	2019-03-02	1	2019-03-09	7
1	2019-03-09	2	2019-03-20	11
1	2019-03-20	3	NULL	NULL

Date Functions in Aggregate Summaries and Window Functions

If we wanted to use the next_purchase field name inside the DATEDIFF() function to avoid inserting that LEAD() calculation twice, we could use another query layer and have a query of a query of a query, as shown in the following code.

```
SELECT
    a.customer_id,
    a.market_date AS first_purchase,
    a.next_purchase AS second_purchase,
    DATEDIFF(a.next_purchase, a.market_date) AS time_between_1st_2nd_
purchase
FROM
(
    SELECT
        x.customer_id,
        x.market_date,
        RANK() OVER (PARTITION BY x.customer_id ORDER BY x.market_date)
AS purchase_number,
        LEAD(x.market_date,1) OVER (PARTITION BY x.customer_id ORDER BY
x.market_date) AS next_purchase
        FROM
        (
            SELECT DISTINCT customer_id, market_date
            FROM farmers_market.customer_purchases
        ) x
) a
WHERE a.purchase_number = 1
```

We removed the customer_id filter to return all customers, then filter to each customer's first purchase by adding a filter on the calculated purchase_number.

This query answers "How many days pass between each customer's first and second purchase?"

customer_id	first_purchase	second_purchase	time_between_1st_2nd_purchase
1	2019-03-02	2019-03-09	7
2	2019-03-02	2019-03-13	11
3	2019-03-02	2019-03-16	14
4	2019-03-02	2019-03-09	7
5	2019-03-09	NULL	NULL
7	2019-03-09	2019-03-20	11
10	2019-03-02	2019-03-16	14
12	2019-03-09	2019-03-16	7

Date Functions in Aggregate Summaries and Window Functions

The director needs a list of customers who made only one purchase at a market event last month to send them a discount coupon for April. How would you generate that list?

Well, first we have to find everyone who made a purchase in the 31 days prior to March 31, 2019. Then, we need to filter that list to those who only made a purchase on a single market date during that time.

```
SELECT DISTINCT customer_id, market_date
FROM farmers_market.customer_purchases
WHERE DATEDIFF('2019-03-31', market_date) <= 31
```

Date Functions in Aggregate Summaries and Window Functions

Then, we could query the results of that query, count the distinct market_date values per customer during that time, and filter to those with exactly one market date, using the HAVING clause

```
SELECT x.customer_id,
    COUNT(DISTINCT x.market_date) AS market_count
FROM
(
    SELECT DISTINCT customer_id, market_date
    FROM farmers_market.customer_purchases
    WHERE DATEDIFF('2019-03-31', market_date) <= 31
) x
GROUP BY x.customer_id
HAVING COUNT(DISTINCT market_date) = 1
```

customer_id	market_count
5	1

Exercises

1. Get the customer_id, month, and year (in separate columns) of every purchase in the farmers_market.customer_purchases table.

2. Write a query that filters to purchases made in the past two weeks, returns the earliest market_date in that range as a field called sales_since_date, and a sum of the sales (quantity * cost_to_customer_per_qty) during that date range.

   - Your final answer should use the CURDATE() function, but if you want to test it out on the Farmer's Market database, you can replace your CURDATE() with the value '2019 03-31' to get the report for the two weeks prior to March 31, 2019 (otherwise your query will not return any data, because none of the dates in the database will have occurred within two weeks of you writing the query).

Exploratory Data Analysis with SQL

Demonstrating EDA with SQL

Here's a real-world scenario for this Exploratory Data Analysis:

The Director of the Farmer’s Market asks us to build reports using the database mentioned in this module.

While they haven't provided specific requirements, they've mentioned interest in general product availability and purchase trends and shared the E-R diagram.

Given this, we should explore the product, vendor_inventory, and customer_purchases tables.

Demonstrating EDA with SQL

Some sensible questions to ask via query are:

- How large are the tables, and how far back in time does the data go?
- What kind of information is available about each product and each purchase?
- What is the granularity of each of these tables; what makes a row unique?
- Since we'll be looking at trends over time, what kind of date and time dimensions are available, and how do the different values look when summarized over time?
- How is the data in each table related to the other tables? How might we join them together to summarize the details for reporting?

Exploring the Products Table

Some databases (like MySQL) offer a function called DESCRIBE [table name] or DESC [table name], or have a special schema to select from to list the columns, data types, and other settings for fields in tables.

But this function isn't available in every database system and doesn't show a preview of the data, so we'll take a more universal approach here to preview data in a table.

Let's start with the product table first:

```
SELECT * FROM farmers_market.product
LIMIT 10
```

Exploring the Products Table

The table includes product_id, product_name, product_size, product_category_id, and product_qty_type.

It doesn't show individual items for sale or purchased, which would be typical in a transactional table. The product_category_id is likely a foreign key, which we should verify later.

The table shows various product_name and product_size values but only two product_qty_type values: “lbs” and “unit.”

product_id	product_name	product_size	product_category_id	product_qty_type
1	Habanero Peppers - Organic	medium	1	lbs
2	Jalapeno Peppers - Organic	small	1	lbs
3	Poblano Peppers - Organic	large	1	unit
4	Banana Peppers - Jar	8 oz	3	unit
5	Whole Wheat Bread	1.5 lbs	3	unit
6	Cut Zinnias Bouquet	medium	5	unit
7	Apple Pie	10"	3	unit
8	Cherry Pie	10"	3	unit
9	Sweet Potatoes	medium	1	lbs
10	Eggs	1 dozen	6	unit

Exploring the Products Table

The product_id appears to be the primary key. What if we didn't know whether it was a unique identifier?

To see if any two rows share the same product_id, we can group by product_id and return groups with more than one record. This checks if product_id is currently unique per record, though it doesn't guarantee it's the primary key.

```
SELECT product_id, count(*)
FROM farmers_market.product
GROUP BY product_id
HAVING count(*) > 1
```

No results were returned, indicating that each product_id is unique and the table has a granularity of one row per product.

Exploring the Products Table

What are the different product categories and their details? Let's check the product_category table.

```
SELECT * FROM farmers_market.product_category
```

product_category_id	product_category_name
1	Fresh Fruits & Vegetables
2	Packaged Pantry Goods
3	Packaged Prepared Food
4	Freshly Prepared Food
5	Plants & Flowers
6	Eggs & Meat (Fresh or Frozen)
7	Non-Edible Products

How many different products are there in the catalog-like product metadata table?

```
SELECT count(*) FROM farmers_market.product
```

count(*)
23

Exploring the Products Table

We might next ask, "How many products are there per product category?"

We'll quickly join the product table and the product_category table to pull in the category names that we think go with the IDs here, and count up the products in each category:

```
SELECT pc.product_category_id, pc.product_category_name,
    count(product_id) AS count_of_products
FROM farmers_market.product_category AS pc
LEFT JOIN farmers_market.product AS p
    ON pc.product_category_id = p.product_category_id
GROUP BY pc.product_category_id
```

product_category_id	product_category_name	count_of_products
1	Fresh Fruits & Vegetables	13
2	Packaged Pantry Goods	1
3	Packaged Prepared Food	4
4	Freshly Prepared Food	0
5	Plants & Flowers	1
6	Eggs & Meat (Fresh or Frozen)	2
7	Non-Edible Products	2

All IDs match the categories, with "Fresh Fruits & Vegetables" being the most common.

The "Freshly Prepared Food" category currently has no products.

Exploring Possible Column Values

To explore the values in a column with string categories, we can ask, **'What is in the product_qty_type field, and how many different types are there?'** This can be answered with a DISTINCT query:

```
SELECT DISTINCT product_qty_type
FROM farmers_market.product
```

product_qty_type
lbs
unit
NULL

Let's take a look at some of the data in the vendor_inventory table next:

```
SELECT * FROM farmers_market.vendor_inventory
LIMIT 10
```

market_date	quantity	vendor_id	product_id	original_price
2019-07-03	7.38	7	1	6.99
2019-07-06	10.96	7	1	6.99
2019-07-10	13.08	7	1	6.99
2019-07-13	10.22	7	1	6.99
2019-07-17	10.59	7	1	6.99
2019-07-20	9.04	7	1	6.99
2019-07-24	10.66	7	1	6.99
2019-07-27	6.76	7	1	6.99
2019-07-31	11.23	7	1	6.99
2019-08-03	10.72	7	1	6.99

Exploring Possible Column Values

We should confirm the primary key by grouping the expected unique fields and using HAVING to check for multiple records with the same combination:

```
SELECT market_date, vendor_id, product_id, count(*)
FROM farmers_market.vendor_inventory
GROUP BY market_date, vendor_id, product_id
HAVING count(*) > 1
```

No combinations of these three values appear in more than one row, confirming that the vendor_inventory table has a unique record for each market date, vendor, and product.

Exploring Possible Column Values

This table includes dates in the market_date field. We can ask, **'How far back does the data go, and when was the most recent market tracked?'**

```
SELECT min(market_date), max(market_date)
FROM farmers_market.vendor_inventory
```

min(market_date)	max(market_date)
2019-04-03	2020-10-10

We have about 1.5 years of records, so for any forecasting involving annual seasonality, we'll need to explain the limited training data due to incomplete years of trends.

Exploring Possible Column Values

We might ask: **How many different vendors are there? When did they start selling, and which are still active at the most recent market date?**

We can do that by grouping the previous query by vendor_id to get the earliest and latest dates for which each vendor had inventory.

```
SELECT vendor_id, min(market_date), max(market_date)
FROM farmers_market.vendor_inventory
GROUP BY vendor_id
ORDER BY min(market_date), max(market_date)
```

vendor_id	min(market_date)	max(market_date)
7	2019-04-03	2020-10-10
8	2019-04-03	2020-10-10
4	2019-06-01	2020-09-30

Exploring Changes Over Time

After seeing the output, we might ask: **Do most vendors sell year-round, or do vendor numbers vary by season?** We'll extract the month and year from each market date and count the vendors each month:

market_year	market_month	vendors_with_inventory
2019	4	2
2019	5	2
2019	6	3
2019	7	3
2019	8	3
2019	9	3
2019	10	2
2019	11	2
2019	12	2
2020	3	2
2020	4	2
2020	5	2
2020	6	3
2020	7	3
2020	8	3
2020	9	3
2020	10	2

```
SELECT
EXTRACT(YEAR FROM market_date) AS market_year,
EXTRACT(MONTH FROM market_date) AS market_month,
COUNT(DISTINCT vendor_id) AS vendors_with_inventory
FROM farmers_market5.vendor_inventory
GROUP BY EXTRACT(YEAR FROM market_date), EXTRACT(MONTH FROM market_date)
ORDER BY EXTRACT(YEAR FROM market_date), EXTRACT(MONTH FROM market_date)
```

Exploring Changes Over Time

There are 3 vendors from June to September and 2 per month the rest of the year, suggesting that one vendor (likely vendor 4) may be seasonal.

Interestingly, output shows no data for months 1 and 2, indicating the market is closed in January and February.

Exploring Changes Over Time

Next, let's look at the details of what a particular vendor's inventory looks like.

```
SELECT * FROM farmers_market.vendor_inventory
WHERE vendor_id = 7
ORDER BY market_date, product_id
```

This vendor sold only one product for most of May and June, with additional products (IDs 1–3) appearing in July.

Product 4's price remained unchanged at $4.00 throughout the period.

market_date	quantity	vendor_id	product_id	original_price
2019-05-15	30.00	7	4	4.00
2019-05-18	30.00	7	4	4.00
2019-05-22	40.00	7	4	4.00
2019-05-25	30.00	7	4	4.00
2019-05-29	40.00	7	4	4.00
2019-06-01	30.00	7	4	4.00
2019-06-05	40.00	7	4	4.00
2019-06-08	30.00	7	4	4.00
2019-06-12	30.00	7	4	4.00
2019-06-15	40.00	7	4	4.00
2019-06-19	40.00	7	4	4.00
2019-06-22	40.00	7	4	4.00
2019-06-26	40.00	7	4	4.00
2019-06-29	30.00	7	4	4.00
2019-07-03	7.38	7	1	6.99
2019-07-03	33.63	7	2	3.49
2019-07-03	70.00	7	3	0.50
2019-07-03	40.00	7	4	4.00

رواد مصر الرقمية

وزارة الاتصالات وتكنولوجيا المعلومات

Exploring Multiple Tables Simultaneously

Some products have round quantities, while others seem to be sold by weight. This vendor always brings either 30 or 40 of product 4 to each market. It would be useful to compare how many are sold each time.

Let's check the customer_purchases table to compare product purchases with the vendor's inventory. First, we'll review the available data in that table.

Customer flow throughout the day or how long each customer spends at the market

```
SELECT * FROM farmers_market.customer_purchases
LIMIT 10
```

product_id	vendor_id	market_date	customer_id	quantity	cost_to_customer_per_qty	transaction_time
1	7	2019-07-03	2	2.77	6.99	18:11:00
1	7	2019-07-03	14	0.99	6.99	17:32:00
1	7	2019-07-03	14	2.18	6.99	18:23:00
1	7	2019-07-03	15	1.53	6.99	18:41:00
1	7	2019-07-03	16	2.02	6.99	18:18:00
1	7	2019-07-03	17	4.75	6.99	17:27:00
1	7	2019-07-06	4	0.27	6.99	12:20:00
1	7	2019-07-06	12	3.60	6.99	09:33:00
1	7	2019-07-06	14	3.04	6.99	13:05:00
1	7	2019-07-10	3	2.48	6.99	18:40:00

Exploring Multiple Tables Simultaneously

Since we see vendor_id and product_id are both included here, we can look closer at purchases of vendor 7's product #4

```
SELECT * FROM farmers_market.customer_purchases
WHERE vendor_id = 7 AND product_id = 4
ORDER BY market_date, transaction_time
```

product_id	vendor_id	market_date	customer_id	quantity	cost_to_customer_per_qty	transaction_time
4	7	2019-04-03	7	5.00	4.00	17:59:00
4	7	2019-04-03	4	1.00	4.00	18:09:00
4	7	2019-04-03	12	3.00	4.00	18:35:00
4	7	2019-04-03	3	1.00	4.00	18:44:00
4	7	2019-04-03	6	4.00	4.00	18:49:00
4	7	2019-04-03	5	3.00	4.00	18:54:00
4	7	2019-04-03	16	2.00	4.00	18:58:00
4	7	2019-04-06	12	5.00	4.00	08:12:00
4	7	2019-04-06	12	5.00	4.00	08:41:00
4	7	2019-04-06	2	5.00	4.00	09:34:00
4	7	2019-04-06	5	1.00	4.00	11:51:00
4	7	2019-04-06	16	2.00	4.00	13:08:00
4	7	2019-04-06	16	5.00	4.00	13:12:00
4	7	2019-04-06	14	2.00	4.00	13:16:00

We can see that for the two market dates that are visible, most customers are buying between 1 and 5 of these items at a time and spending $4 per item.

Exploring Multiple Tables Simultaneously

Examining the customer_purchases data, we observe multiple sales per vendor, product, to compare daily sales with vendor inventory, we'll aggregate sales by market date, vendor_id, and product_id, summing quantities sold and calculating total sales by multiplying each row's quantity by its cost.

```
SELECT market_date,
    vendor_id,
    product_id,
    SUM(quantity) quantity_sold,
    SUM(quantity * cost_to_customer_per_qty) total_sales
FROM farmers_market.customer_purchases
WHERE vendor_id = 7 and product_id = 4
GROUP BY market_date, vendor_id, product_id
ORDER BY market_date, vendor_id, product_id
```

market_date	vendor_id	product_id	quantity_sold	total_sales
2019-04-03	7	4	19.00	76.0000
2019-04-06	7	4	30.00	120.0000
2019-04-10	7	4	23.00	92.0000
2019-04-13	7	4	30.00	120.0000
2019-04-17	7	4	39.00	156.0000
2019-04-20	7	4	20.00	80.0000
2019-04-24	7	4	27.00	108.0000
2019-04-27	7	4	29.00	116.0000
2019-05-01	7	4	22.00	88.0000
2019-05-04	7	4	25.00	100.0000
2019-05-08	7	4	22.00	88.0000
2019-05-11	7	4	30.00	120.0000
2019-05-15	7	4	35.00	140.0000
2019-05-18	7	4	30.00	120.0000

Exploring Inventory vs. Sales

We now have everything needed to compare product sales to inventory, except for the inventory counts. After exploring the data, we can start joining tables to better understand the relationships between entities.

For instance, with customer_purchases aggregated to match the granularity of the vendor_inventory table, we can join them to view inventory alongside sales.

First, we'll join the two tables (detail and summary) and display all columns to verify the join. We'll alias the customer_purchases summary table as 'sales' and limit the output to 10 rows since we haven't filtered by vendor or product yet, which would otherwise return many rows.

Exploring Inventory vs. Sales

Here is the query:

```
SELECT * FROM farmers_market.vendor_inventory AS vi
    LEFT JOIN
        (
        SELECT market_date,
            vendor_id,
            product_id,
            SUM(quantity) AS quantity_sold,
            SUM(quantity * cost_to_customer_per_qty) AS total_sales
        FROM farmers_market.customer_purchases
        GROUP BY market_date, vendor_id, product_id
        ) AS sales
        ON vi.market_date = sales.market_date
            AND vi.vendor_id = sales.vendor_id
            AND vi.product_id = sales.product_id
ORDER BY vi.market_date, vi.vendor_id, vi.product_id
LIMIT 10
```

Exploring Inventory vs. Sales

Vendor_id, product_id, and market_date match on every row, and the summary values for vendor_id 8 and product_id 4 align with the customer_purchases table. This confirms the join is correct, so we can remove redundant columns and specify which ones to display from the vendor_inventory table, the 'left' side of the JOIN, as customers can't buy non-existent inventory.

market_date	quantity	vendor_id	product_id	original_price	market_date	vendor_id	product_id	quantity_sold	total_sales
2019-04-03	40.00	7	4	4.00	2019-04-03	7	4	19.00	76.0000
2019-04-03	16.00	8	5	6.50	2019-04-03	8	5	20.00	130.0000
2019-04-03	8.00	8	7	18.00	2019-04-03	8	7	8.00	144.0000
2019-04-03	10.00	8	8	18.00	2019-04-03	8	8	8.00	144.0000
2019-04-06	40.00	7	4	4.00	2019-04-06	7	4	30.00	120.0000
2019-04-06	23.00	8	5	6.50	2019-04-06	8	5	20.00	130.0000
2019-04-06	8.00	8	7	18.00	2019-04-06	8	7	7.00	126.0000
2019-04-06	8.00	8	8	18.00	2019-04-06	8	8	6.00	108.0000
2019-04-10	30.00	7	4	4.00	2019-04-10	7	4	23.00	92.0000
2019-04-10	23.00	8	5	6.50	2019-04-10	8	5	25.00	162.5000

Exploring Inventory vs. Sales

We can join additional lookup tables to convert IDs to readable names, bringing in vendor and product names. Then, we can filter for vendor 7 and product 4 to compare this vendor's inventory with sales at each market.

```
SELECT vi.market_date,
    vi.vendor_id,
    v.vendor_name,
    vi.product_id,
    p.product_name,
    vi.quantity AS quantity_available,
    sales.quantity_sold,
    vi.original_price,
    sales.total_sales
FROM farmers_market.vendor_inventory AS vi
```

```
    LEFT JOIN
        (
        SELECT market_date,
            vendor_id,
            product_id,
            SUM(quantity) AS quantity_sold,
            SUM(quantity * cost_to_customer_per_qty) AS total_sales
        FROM farmers_market.customer_purchases
        GROUP BY market_date, vendor_id, product_id
        ) AS sales
    ON vi.market_date = sales.market_date
        AND vi.vendor_id = sales.vendor_id
        AND vi.product_id = sales.product_id
    LEFT JOIN farmers_market.vendor v
        ON vi.vendor_id = v.vendor_id
    LEFT JOIN farmers_market.product p
        ON vi.product_id = p.product_id
WHERE vi.vendor_id = 7
    AND vi.product_id = 4
ORDER BY vi.market_date, vi.vendor_id, vi.product_id
```

Exploring Inventory vs. Sales

This vendor is called Marco's Peppers, and the product we were looking at is jars of Banana Peppers. He brings 30–40 jars each time and sells between 1 and 40 jars per market. We quickly found this by sorting the output by quantity_sold in ascending and descending order in the SQL editor.

Alternatively, we could have added quantity_sold to the ORDER BY clause or used a query to calculate the MIN and MAX values

market_date	vendor_id	vendor_name	product_id	product_name	quantity_available	quantity_sold	original_price	total_sales
2019-04-03	7	Marco's Peppers	4	Banana Peppers - Jar	40.00	19.00	4.00	76.0000
2019-04-06	7	Marco's Peppers	4	Banana Peppers - Jar	40.00	30.00	4.00	120.0000
2019-04-10	7	Marco's Peppers	4	Banana Peppers - Jar	30.00	23.00	4.00	92.0000
2019-04-13	7	Marco's Peppers	4	Banana Peppers - Jar	30.00	30.00	4.00	120.0000
2019-04-17	7	Marco's Peppers	4	Banana Peppers - Jar	40.00	39.00	4.00	156.0000
2019-04-20	7	Marco's Peppers	4	Banana Peppers - Jar	40.00	20.00	4.00	80.0000
2019-04-24	7	Marco's Peppers	4	Banana Peppers - Jar	40.00	27.00	4.00	108.0000
2019-04-27	7	Marco's Peppers	4	Banana Peppers - Jar	30.00	29.00	4.00	116.0000
2019-05-01	7	Marco's Peppers	4	Banana Peppers - Jar	40.00	22.00	4.00	88.0000
2019-05-04	7	Marco's Peppers	4	Banana Peppers - Jar	30.00	25.00	4.00	100.0000
2019-05-08	7	Marco's Peppers	4	Banana Peppers - Jar	40.00	22.00	4.00	88.0000
2019-05-11	7	Marco's Peppers	4	Banana Peppers - Jar	40.00	30.00	4.00	120.0000

Q&A

Questions and answers

وزارة الاتصالات وتكنولوجيا المعلومات

رواد مصر الرقمية

Thanks!