More Advanced Query Structures

25/08/2024

UNIONs

Using a UNION, you can combine any two queries that result in the same number of columns with the same data types. The columns must be in the same order in both queries.

The syntax is simple: write two queries with the same number and type of fields, and put a UNION keyword between them:

```
SELECT market_year, MIN(market_date) AS first_market_date
FROM farmers_market.market_date_info
WHERE market_year = '2019'

UNION

SELECT market_year, MIN(market_date) AS first_market_date
FROM farmers_market.market_date_info
WHERE market_year = '2020'
```

UNIONs

You could just write one query, GROUP BY market_year, and filter to WHERE market_year IN ('2019','2020') and get the same output.

For a more complex example using CTEs (Common Table Expressions) and UNIONs, we'll create a report showing the products with the largest quantities at each market: the bulk product with the highest weight and the unit product with the highest count.

union_slide_3.sql

The with CTE aggregates the quantity of products available at the market by market_date and product_id and outputs a table with the total quantity of each product available on each market date.

UNIONs

The main Query: ranked_products ranks products by quantity for each market_date and product_qty_type (separately for unit and lbs)

RANK() Window Function: Ranks products within each market_date by their total_quantity_available. The highest quantity receives a rank of 1.

UNION: Combines results for both quantity types (unit and lbs) into a single dataset.

Filtering: The outer query filters out only the top-ranked products (those with quantity_rank = 1).

market_date	product_id	product_name	total_quantity_available	product_qty_type	quantity_rank
2019-08-03	16	Sweet Corn	300.00	unit	1
2019-08-03	2	Jalapeno Peppers - Organic	32.23	lbs	1
2019-08-07	16	Sweet Corn	300.00	unit	1
2019-08-07	2	Jalapeno Peppers - Organic	29.28	lbs	1
2019-08-10	16	Sweet Corn	250.00	unit	1
2019-08-10	2	Jalapeno Peppers - Organic	27.18	lbs	1
2019-08-14	16	Sweet Corn	200.00	unit	1
2019-08-14	2	Jalapeno Peppers - Organic	33.35	lbs	1
2019-08-17	16	Sweet Corn	300.00	unit	1
2019-08-17	2	Jalapeno Peppers - Organic	25.58	lbs	1
2019-08-21	16	Sweet Corn	250.00	unit	1
2019-08-21	2	Jalapeno Peppers - Organic	32.02	lbs	1
2019-08-24	16	Sweet Corn	250.00	unit	1
2019-08-24	2	Jalapeno Peppers - Organic	17.29	lbs	1
2019-08-28	16	Sweet Corn	250.00	unit	1
2019-08-28	2	Jalapeno Peppers - Organic	26.20	lbs	1
2019-08-31	16	Sweet Corn	300.00	unit	1
2019-08-31	2	Jalapeno Peppers - Organic	27.87	lbs	1

UNIONs

There is at least one other way to get the preceding output that doesn't require a UNION.

union_slide_5.sql

First with CTE (product_quantity_by_date): Aggregates the total quantity available for each product on each market date.

Second with CTE (rank_by_qty_type): Ranks products based on their quantity within each market_date and product_qty_type.

Final Selection: Filters out only the products with the highest quantity for each market_date and product_qty_type.

We were able to accomplish the same result without the UNION by partitioning by both the market_date and product_qty_type in the RANK() function, resulting in a ranking for each date and quantity type.

UNIONs

Because I have shown two examples of UNION queries that don't actually require UNIONs, I wanted to mention one case when a UNION is definitely required:

When you have separate tables with the same columns representing different time periods—like event logs split across multiple files or static snapshots of a dynamic dataset from different times—or when data is migrated from different systems and needs to be combined into a single view to see the complete record history.

Self-Join to Determine To-Date Maximum

A self-join in SQL is when a table is joined to itself (you can think of it like two copies of the table joined together) in order to compare rows to one another.

```
SELECT t1.id1, t1.field2, t2.field2, t2.field3
FROM mytable AS t1
  LEFT JOIN mytable AS t2
    ON t1.id1 = t2.id1
```

For example, we want to determine whether there is any previous date that has a higher sales total than the “current” row we’re looking at, and we can use a self-join to do that comparison.

Self-Join to Determine To-Date Maximum

First, we'll need to summarize the sales by market_date. We'll put this query into a CTE (WITH clause), and alias it sales_per_market_date.

```
WITH
sales_per_market_date AS
(
    SELECT
        market_date,
        ROUND(SUM(quantity * cost_to_customer_per_qty),2) AS sales
    FROM farmers_market.customer_purchases
    GROUP BY market_date
    ORDER BY market_date
)

SELECT *
FROM sales_per_market_date
LIMIT 10
```

market_date	sales
2019-04-03	439.00
2019-04-06	557.50
2019-04-10	483.43
2019-04-13	384.62
2019-04-17	507.50
2019-04-20	433.73
2019-04-24	346.42
2019-04-27	433.58
2019-05-01	488.92
2019-05-04	496.74

Self-Join to Determine To-Date Maximum

To compare each row with all prior dates, join the table to itself using the market_date field with a less-than sign (<) instead of an equal sign.

So, we're joining every row to every other row in the database that has a lower market_date value than it does.

```
WITH
sales_per_market_date AS
(
    SELECT
        market_date,
        ROUND(SUM(quantity * cost_to_customer_per_qty),2) AS sales
    FROM farmers_market.customer_purchases
    GROUP BY market_date
    ORDER BY market_date
)

SELECT *
FROM sales_per_market_date AS cm
    LEFT JOIN sales_per_market_date AS pm
        ON pm.market_date < cm.market_date
WHERE cm.market_date = '2019-04-13'
```

market_date	sales	market_date	sales
2019-04-13	384.62	2019-04-03	439.00
2019-04-13	384.62	2019-04-06	557.50
2019-04-13	384.62	2019-04-10	483.43

Self-Join to Determine To-Date Maximum

Now we'll use a MAX() function on the pm.sales field and GROUP BY cm.market_date to get the previous highest sales value.

```
WITH
sales_per_market_date AS
(
    SELECT
        market_date,
        ROUND(SUM(quantity * cost_to_customer_per_qty),2) AS sales
    FROM farmers_market.customer_purchases
    GROUP BY market_date
    ORDER BY market_date
)

SELECT
    cm.market_date,
    cm.sales,
    MAX(pm.sales) AS previous_max_sales
FROM sales_per_market_date AS cm
    LEFT JOIN sales_per_market_date AS pm
        ON pm.market_date < cm.market_date
WHERE cm.market_date = '2019-04-13'
GROUP BY cm.market_date, cm.sales
```

market_date	sales	previous_max_sales
2019-04-13	384.62	557.50

Self-Join to Determine To-Date Maximum

Remove the date filter to get previous_max_sales for each date. Use a CASE statement to add a flag indicating if the current sales are higher than the previous maximum, showing if each date set a sales record.

```
WITH
sales_per_market_date AS
(
    SELECT
        market_date,
        ROUND(SUM(quantity * cost_to_customer_per_qty),2) AS sales
    FROM farmers_market.customer_purchases
    GROUP BY market_date
    ORDER BY market_date
)

SELECT
    cm.market_date,
    cm.sales,
    MAX(pm.sales) AS previous_max_sales,
    CASE WHEN cm.sales > MAX(pm.sales)
        THEN "YES"
        ELSE "NO"
    END sales_record_set
FROM sales_per_market_date AS cm
    LEFT JOIN sales_per_market_date AS pm
        ON pm.market_date < cm.market_date
GROUP BY cm.market_date, cm.sales
```

market_date	sales	previous_max_sales	sales_record_set
2019-04-06	557.50	439.00	YES
2019-04-10	483.43	557.50	NO
2019-04-13	384.62	557.50	NO
2019-04-17	507.50	557.50	NO
2019-04-20	433.73	557.50	NO
2019-04-24	346.42	557.50	NO
2019-04-27	433.58	557.50	NO
2019-05-01	488.92	557.50	NO
2019-05-04	496.74	557.50	NO
2019-05-08	490.86	557.50	NO
2019-05-11	446.50	557.50	NO
2019-05-15	426.00	557.50	NO
2019-05-18	465.93	557.50	NO
2019-05-22	531.40	557.50	NO
2019-05-25	376.31	557.50	NO
2019-05-29	576.30	557.50	YES
2019-06-01	472.02	576.30	NO
2019-06-05	377.54	576.30	NO
2019-06-08	470.85	576.30	NO

Counting New vs. Returning Customers by Week

The manager of the farmer's market might want to monitor how many customers are visiting the market per week, and how many of those are new, making a purchase for the first time.

To determine if a customer is new, compare their purchase date to their earliest purchase date. If the minimum purchase date is today, the customer made their first purchase today and is therefore new

Summarize each market date attended by every customer and determine their first purchase date using MIN() as a window function partitioned by customer_id.

```
SELECT DISTINCT
    customer_id,
    market_date,
    MIN(market_date) OVER(PARTITION BY cp.customer_id) AS first_purchase_
date
FROM farmers_market.customer_purchases cp
```

Counting New vs. Returning Customers by Week

The output is:

customer_id	market_date	first_purchase_date
2	2020-08-15	2019-04-06
2	2020-09-19	2019-04-06
2	2020-10-07	2019-04-06
2	2019-06-05	2019-04-06
2	2019-07-27	2019-04-06
3	2019-07-10	2019-04-03
3	2019-07-31	2019-04-03
3	2019-09-25	2019-04-03
3	2019-09-28	2019-04-03
3	2020-09-16	2019-04-03
3	2020-09-26	2019-04-03
3	2019-07-06	2019-04-03
3	2019-07-20	2019-04-03

Counting New vs. Returning Customers by Week

Place the query in a WITH clause, then join it with market_date_info to get year and week information. Group by week so customers with purchases at multiple markets within the same week will have rows grouped by each year-week combination.

```
WITH
customer_markets_attended AS
(
    SELECT DISTINCT
        customer_id,
        market_date,
        MIN(market_date) OVER(PARTITION BY cp.customer_id) AS first_purchase_
date
FROM farmers_market.customer_purchases cp
)

SELECT
    md.market_year,
    md.market_week,
    COUNT(customer_id) AS customer_visit_count,
    COUNT(DISTINCT customer_id) AS distinct_customer_count
FROM customer_markets_attended AS cma
    LEFT JOIN farmers_market.market_date_info AS md
        ON cma.market_date = md.market_date
GROUP BY md.market_year, md.market_week
ORDER BY md.market_year, md.market_week
```

market_year	market_week	customer_visit_count	distinct_customer_count
2019	14	25	19
2019	15	23	16
2019	16	27	18
2019	17	29	20
2019	18	27	21
2019	19	25	18
2019	20	23	19
2019	21	24	18
2019	22	27	19
2019	23	28	20
2019	24	30	22

Counting New vs. Returning Customers by Week

We also want to get a count of new customers per week, so let's add a column displaying what percent of each week's customers are new. This requires adding two more fields to the query. The first looks like this:

```
COUNT(
    DISTINCT
    CASE WHEN cma.market_date = cma.first_purchase_date
        THEN customer_id
        ELSE NULL
    END
    ) AS new_customer_count
```

The COUNT() function includes a CASE statement that counts rows where the market_date matches the customer's first purchase date.
If the values match, the CASE statement returns a customer_id to count; otherwise, it returns NULL. This results in a distinct count of customers making their first purchase that week.

Counting New vs. Returning Customers by Week

The second field, which is the last listed in the following full query, then divides that same value by the total distinct count of customer IDs, giving us a percentage.

slide_15.sql

market_year	market_week	customer_visit_count	distinct_customer_count	new_customer_count	new_customer_percent
2019	14	25	19	19	1.0000
2019	15	23	16	2	0.1250
2019	16	27	18	3	0.1667
2019	17	29	20	1	0.0500
2019	18	27	21	1	0.0476
2019	19	25	18	0	0.0000
2019	20	23	19	0	0.0000
2019	21	24	18	0	0.0000
2019	22	27	19	0	0.0000
2019	23	28	20	0	0.0000
2019	24	30	22	0	0.0000

Exercises

1. Starting with the query associated with Slide 11, put the larger SELECT statement in a second CTE, and write a query that queries from its results to display the current record sales and associated market date. Can you think of another way to generate the same results?

2. Modify the “New vs. Returning Customers Per Week” report to summarize the counts by vendor by week.

3. Using a UNION, write a query that displays the market dates with the highest and lowest total sales.

Storing and Modifying Data

Storing and Modifying Data

In this part, we'll cover some types of SQL queries beyond SELECT statements, such as INSERT statements, which allow you to store the results of your query in a new table in the database.

Storing SQL Datasets as Tables and Views

In most databases, you can store query results as a table or a view. A table saves a snapshot of the results at the time of the query, while a view stores the SQL and generates results on-demand based on the current state of the referenced data.

One way to store the results of a query is to use a CREATE TABLE statement. The syntax is

```
CREATE TABLE [schema_name].[new_table_name] AS
(
   [your query here]
)
```

After creating a table, you can query it like any other table or view.

Storing SQL Datasets as Tables and Views

If needed, you can DROP the table to delete or re-create it with a different name or definition. The syntax for dropping a table is simply:

```
DROP TABLE [schema_name].[table_name]
```

Storing SQL Datasets as Tables and Views

To create, select from, and drop a table with a snapshot of products from the Farmer's Market database where the quantity type is 'unit,' run these three queries in sequence:

```
CREATE TABLE farmers_market.product_units AS
(
    SELECT *
    FROM farmers_market.product
    WHERE product_qty_type = "unit"
)
;

SELECT * FROM farmers_market.product_units
;

DROP TABLE farmers_market.product_units
;
```

Storing SQL Datasets as Tables and Views

Database views are like shortcuts to queries. They don't store data but define a query to be run when accessed. Dropping a view removes this shortcut, not the underlying data.

```
CREATE VIEW farmers_market.product_units_vw AS
(
    SELECT *
    FROM farmers_market.product
    WHERE product_qty_type = "unit"
)
;

SELECT * FROM farmers_market.product_units_vw
;

DROP VIEW farmers_market.product_units_vw
;
```

Adding a Timestamp Column

To track row creation/modification times, add a timestamp column to your table definition.

We can modify the preceding CREATE TABLE example to include a timestamp column as follows:

```
CREATE TABLE farmers_market.product_units AS
(
    SELECT p.*,
        CURRENT_TIMESTAMP AS snapshot_timestamp
    FROM farmers_market.product AS p
    WHERE product_qty_type = "unit"
)
```

product_id	product_name	product_size	product_category_id	product_qty_type	snapshot_timestamp
3	Poblano Peppers - Organic	large	1	unit	2021-04-18 00:49:24
4	Banana Peppers - Jar	8 oz	3	unit	2021-04-18 00:49:24
5	Whole Wheat Bread	1.5 lbs	3	unit	2021-04-18 00:49:24
6	Cut Zinnias Bouquet	medium	5	unit	2021-04-18 00:49:24
7	Apple Pie	10"	3	unit	2021-04-18 00:49:24
8	Cherry Pie	10"	3	unit	2021-04-18 00:49:24
10	Eggs	1 dozen	6	unit	2021-04-18 00:49:24
12	Baby Salad Lettuce Mix - Bag	1/2 lb	1	unit	2021-04-18 00:49:24
16	Sweet Corn	Ear	1	unit	2021-04-18 00:49:24
18	Carrots - Organic	bunch	1	unit	2021-04-18 00:49:24
19	Farmer's Market Resuable Shopping Bag	medium	7	unit	2021-04-18 00:49:24

Inserting Rows & Updating Values in Tables

If you want to modify data in an existing database table, you can use an INSERT statement to add a new row or an UPDATE statement to modify an existing row of data in a table.

This part covers INSERT INTO SELECT, which inserts query results into a table. Syntax:

```
INSERT INTO [schema_name].[table_name] ([comma-separated list of column
    names])
[your SELECT query here]
```

Inserting Rows & Updating Values in Tables

So, if we wanted to add rows to our product_units table created earlier, we would write:

```
INSERT INTO farmers_market.product_units (product_id, product_name,
product_size, product_category_id, product_qty_type, snapshot_timestamp)
    SELECT
        product_id,
        product_name,
        product_size,
        product_category_id,
        product_qty_type,
        CURRENT_TIMESTAMP
    FROM farmers_market.product AS p
    WHERE product_id = 23
```

product_id	product_name	product_size	product_category_id	product_qty_type	snapshot_timestamp
23	Maple Syrup - Jar	8 oz	2	unit	2021-04-11 23:41:41
23	Maple Syrup - Jar	8 oz	2	unit	2021-04-18 00:49:24

Inserting Rows & Updating Values in Tables

If you make a mistake when inserting a row and want to delete it, the syntax is simply:

```
DELETE FROM [schema_name].[table_name]
WHERE [set of conditions that uniquely identifies the row]
```

You may want to start with SELECT * instead of DELETE so you can see what rows will be deleted before running the DELETE statement!

The product_id & snapshot_timestamp uniquely identify rows in the product_units table, so we can run the following statement to delete the row added by our previous INSERT INTO:

```
DELETE FROM farmers_market.product_units
WHERE product_id = 23
    AND snapshot_timestamp = '2021-04-18 00:49:24'
```

Inserting Rows & Updating Values in Tables

Sometimes you want to update a value in an existing row instead of inserting a totally new row. The syntax for an UPDATE statement is as follows:

```
UPDATE [schema_name].[table_name]
SET [column_name] = [new value]
WHERE [set of conditions that uniquely identifies the rows you want to
    change]
```

Suppose you've scheduled vendor booths for months, but vendor 4 can't attend on October 10. You decide to move vendor 8 to vendor 4's larger, more accessible booth for that day.

Inserting Rows & Updating Values in Tables

Before making any changes, let's snapshot the existing vendor booth assignments, along with the vendor name and booth type, into a new table:

```
CREATE TABLE farmers_market.vendor_booth_log AS
(
    SELECT vba.*,
        b.booth_type,
        v.vendor_name,
        CURRENT_TIMESTAMP AS snapshot_timestamp
    FROM farmers_market.vendor_booth_assignments vba
        INNER JOIN farmers_market.vendor v
            ON vba.vendor_id = v.vendor_id
        INNER JOIN farmers_market.booth b
            ON vba.booth_number = b.booth_number
    WHERE market_date >= '2020-10-01'
)
```

vendor_id	booth_number	market_date	booth_type	vendor_name	snapshot_timestamp
1	2	2020-10-07	Standard	Chris's Sustainable Eggs & Meats	2021-04-18 01:23:24
3	1	2020-10-07	Standard	Mountain View Vegetables	2021-04-18 01:23:24
4	7	2020-10-07	Standard	Fields of Corn	2021-04-18 01:23:24
7	11	2020-10-07	Large	Marco's Peppers	2021-04-18 01:23:24
8	6	2020-10-07	Small	Annie's Pies	2021-04-18 01:23:24
9	8	2020-10-07	Small	Mediterranean Bakery	2021-04-18 01:23:24
1	2	2020-10-10	Standard	Chris's Sustainable Eggs & Meats	2021-04-18 01:23:24
3	1	2020-10-10	Standard	Mountain View Vegetables	2021-04-18 01:23:24
4	7	2020-10-10	Standard	Fields of Corn	2021-04-18 01:23:24
7	11	2020-10-10	Large	Marco's Peppers	2021-04-18 01:23:24
8	6	2020-10-10	Small	Annie's Pies	2021-04-18 01:23:24
9	8	2020-10-10	Small	Mediterranean Bakery	2021-04-18 01:23:24

Inserting Rows & Updating Values in Tables

To update vendor 8's booth assignment, we can run the following SQL:

```
UPDATE farmers_market.vendor_booth_assignments
SET booth_number = 7
WHERE vendor_id = 8 and market_date = '2020-10-10'
```

And we can delete vendor 4's booth assignment with the following SQL:

```
DELETE FROM farmers_market.vendor_booth_assignments
WHERE vendor_id = 4 and market_date = '2020-10-10'
```

Inserting Rows & Updating Values in Tables

After updating booth assignments, the vendor_booth_assignments table shows no record of the previous assignments. However, the vendor_booth_log retains these records. We can insert new records into the log table to capture the latest changes.

```
INSERT INTO farmers_market.vendor_booth_log (vendor_id, booth_number,
market_date, booth_type, vendor_name, snapshot_timestamp)
    SELECT
        vba.vendor_id,
        vba.booth_number,
        vba.market_date,
        b.booth_type,
        v.vendor_name,
        CURRENT_TIMESTAMP AS snapshot_timestamp
    FROM farmers_market.vendor_booth_assignments vba
        INNER JOIN farmers_market.vendor v
            ON vba.vendor_id = v.vendor_id
        INNER JOIN farmers_market.booth b
            ON vba.booth_number = b.booth_number
    WHERE market_date >= '2020-10-01'
```

vendor_id	booth_number	market_date	snapshot_timestamp
4	7	2020-10-03	2021-04-18 01:23:24
4	7	2020-10-07	2021-04-18 01:23:24
4	7	2020-10-10	2021-04-18 01:23:24
8	6	2020-10-03	2021-04-18 01:23:24
8	6	2020-10-07	2021-04-18 01:23:24
8	6	2020-10-10	2021-04-18 01:23:24
4	7	2020-10-03	2021-04-18 01:35:14
4	7	2020-10-07	2021-04-18 01:35:14
8	6	2020-10-03	2021-04-18 01:35:14
8	6	2020-10-07	2021-04-18 01:35:14
8	7	2020-10-10	2021-04-18 01:35:14

Q&A

Questions and answers

Thanks!