MSDS 597

Week 6

UNION and UNION ALL

- UNION and UNION ALL stack data vertically
  - Combine the result of 2 or more SELECT statements
  - This is in contrast to JOINs which put the data side to side
  - Note: When using UNION/UNION ALL ensure that your columns are aligned
- UNION
  - Returns a deduplicated version of the data
  - i.e. any rows that are identical between the two data sets are deduplicated so that only one of those rows remain
- UNION ALL
  - Does not deduplicate the data

UNION

chemistry

student_id	name	grade
1	Albert	95
2	Brenda	90
3	Chris	85

physics

student_id	name	grade
1	Albert	93
2	Brenda	91
4	Dotty	75

```
SELECT
    student_id,
    name
FROM chemistry

UNION

SELECT
    student_id,
    name
FROM physics
```

student_id	name
1	Albert
2	Brenda
3	Chris
4	Dotty

```
SELECT *
FROM chemistry

UNION

SELECT *
FROM physics
```

student_id	name	grade
1	Albert	95
2	Brenda	90
3	Chris	85
1	Albert	93
2	Brenda	91
4	Dotty	75

UNION ALL

chemistry

student_id	name	grade
1	Albert	95
2	Brenda	90
3	Chris	85

physics

student_id	name	grade
1	Albert	93
2	Brenda	91
4	Dotty	75

```
SELECT
    student_id,
    name
FROM chemistry

UNION ALL

SELECT
    student_id,
    name
FROM physics
```

student_id	name
1	Albert
2	Brenda
3	Chris
1	Albert
2	Brenda
4	Dotty

```
SELECT *
FROM chemistry

UNION ALL

SELECT *
FROM physics
```

student_id	name	grade
1	Albert	95
2	Brenda	90
3	Chris	85
1	Albert	93
2	Brenda	91
4	Dotty	75

Subqueries

- SQL operates on tabular data.
  - Tables in a database (physically stored)
  - Subqueries
- Subqueries
  - Query inside of another query (nested queries)
  - Wrapped in parentheses and aliased
  - Returns tabular data which can be used in queries anywhere you would have used a table
  - Allows for advanced logic
    - Filters in the where clause using IN
    - Multi-step queries

```
SELECT
    by_country_product."ShipCountry",
    by_country_product."ProductID",
    by_country_product.quantity,
    by_country_product.quantity * 100.0 /
by_country.total_quantity AS pct_total
FROM (
     SELECT
         "ShipCountry",
         "ProductID",
         SUM("Quantity") AS quantity
     FROM orders
     INNER JOIN order_details od on
orders."OrderID" = od."OrderID"
    GROUP BY 1,2
) by_country_product
LEFT JOIN (
    SELECT
        "ShipCountry",
        SUM("Quantity") AS total_quantity
    FROM orders
    INNER JOIN order_details od2 ON
orders."OrderID" = od2."OrderID"
    GROUP BY 1
) by_country
ON by_country_product."ShipCountry" =
by_country."ShipCountry"
ORDER BY 4 DESC;
```

Subqueries to filter data in WHERE

- Use in conjunction with IN or NOT IN to test against multiple values produced by a subquery

```
-- Top 3 products
SELECT
    *
FROM products
WHERE "ProductID" IN (
    SELECT
        od."ProductID"
    FROM orders
    JOIN order_details od on
orders."OrderID" = od."OrderID"
    GROUP BY 1
    ORDER BY
SUM(od."UnitPrice"*"Quantity"*(1-"Disc
ount")) DESC
    LIMIT 3
    );
```

Subqueries to filter data with JOIN

- Use a subquery in conjunction with an INNER JOIN to filter data.
- This is similar to putting the subquery in a WHERE clause, but is more flexible as it allows you to filter on multiple columns or to multiple values very easily

```
-- Products
SELECT
    *
FROM products p
JOIN (
    SELECT
        od."ProductID",
        SUM(
            od."UnitPrice" *
            "Quantity" *
            (1-"Discount")
        ) AS sales
    FROM orders
    JOIN order_details od on
orders."OrderID" = od."OrderID"
    GROUP BY 1
    ORDER BY 2 DESC
    LIMIT 3
    ) s
ON p."ProductID" = s."ProductID"
```

Subqueries to aggregate in multiple stages

- If you need to perform an aggregation on another aggregation, you can perform the first aggregation in a subquery (the inner query) and then perform the second aggregation in the outer query

```
SELECT
    DATE_PART('month',
"OrderDate"::TIMESTAMP) AS month,
    dow,
    AVG(num_orders) AS avg_orders
FROM (
    SELECT
        "OrderDate",
        DATE_PART('dow',
"OrderDate"::TIMESTAMP) AS dow,
        COUNT(1) AS num_orders
    FROM orders
    GROUP BY 1,2
) a
GROUP BY 1,2
ORDER BY 1,2
```

Common Table Expressions (CTEs)

- Also known as WITH clause
- Essentially a named subquery that is defined earlier in the query
- Useful for pre-processing data to be used/combined in later stages of the query
- Helps to make the code more modular and readable by others
- Can be reused multiple times in the same query
- Multiple CTEs can be created at the start of the query
- CTEs can read/access other CTEs that were defined ahead of it

```
WITH first_cte AS (
    SELECT …
),
second_cte AS (
    SELECT …
),
third_cte AS (
    SELECT *
    FROM first_cte
    UNION
    SELECT *
    FROM second_cte
)
SELECT *
FROM third_cte
```

Analytic Functions

- Sometimes referred to as window functions
- Computes values over a group of rows and returns a single result **for each row**
    - This is different from standard aggregation, where we return a single result **for a group of rows**
- Analytic functions can be aggregate, numbering, or navigation functions
    - Aggregate functions
        - Aggregate functions are computed over the group of rows and the same value is attached to each row
        - SUM, COUNT, MAX, MIN, AVG
        - COUNT(DISTINCT) cannot be used
    - Numbering functions
        - Assign a number to each row based on the row’s position within the group of rows
        - ROW_NUMBER, RANK, DENSE_RANK
    - Navigation functions
        - Can return a value from a different row in the group of rows depending on the function used
        - FIRST_VALUE, LAST_VALUE, LAG, LEAD

Analytics Functions

General Syntax

```
<analytic function> ()
    OVER (
        PARTITION BY
            <partition columns>
        ORDER BY
            <order columns>
        <WINDOW FRAME CLAUSE>
    )
```

Break up the input rows into separate partitions (groups) over each of which the analytic function is independently evaluated

Specify how the rows should be ordered within each partition

The window frame clause defines the window frame around the current row within a partition, over which the analytic function is evaluated. <u>Only aggregate analytic functions can use a window frame clause.</u>

Most commonly written as:

```
ROWS BETWEEN _______ AND _______
```

Where the ______ are filled in with

```
UNBOUNDED PRECEDING
UNBOUNDED FOLLOWING
XX PRECEDING
XX FOLLOWING
CURRENT_ROW
```

Aggregate Analytic Functions

monthly_product_sales

month	product_name	sales
2020-01	Apples	300
2020-01	Bananas	500
2020-02	Apples	200
2020-02	Bananas	300
2020-03	Apples	150

```
SELECT
    month,
    product_name,
    sales,
    -- SUMs all sales and attaches to each row
    SUM(sales) OVER () AS overall_sales,
    -- SUMs all sales per product and attaches to each row
    SUM(sales) OVER (PARTITION BY product_name) AS
total_product_sales,
    -- SUMs all sales per product over all preceding months
    SUM(sales) OVER (
        PARTITION BY product_name
        ORDER BY month
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
cumulative_sum
FROM monthly_product_sales;
```

Aggregate Analytic Functions

Results from previous slide's query

month	product_name	sales	overall_sales	total_product_sales	cumulative_sum
2020-01	Apples	300	1450	650	300
2020-02	Apples	200	1450	650	500
2020-03	Apples	150	1450	650	650
2020-01	Bananas	500	1450	800	500
2020-02	Bananas	300	1450	800	800

Numbering Analytic Functions

ROW_NUMBER: Returns the row number of the row within the window
RANK: Returns the rank of the row within the window, can have ties but the next row in sequence will increment from the number of rows preceding it, not from the previous value
DENSE_RANK: Returns the dense rank of the row within the window, can also have ties but will not skip numbers

x
1
2
2
5
8
10
10

```
SELECT
    x,
    ROW_NUMBER() OVER (ORDER BY x)
AS row_num,
    RANK() OVER (ORDER BY x) AS
rank,
    DENSE_RANK() OVER (ORDER BY x)
AS dense_rank
FROM numbers;
```

x	row_num	rank	dense_rank
1	1	1	1
2	2	2	2
2	3	2	2
5	4	4	3
8	5	5	4
10	6	6	5
10	7	6	5

Navigation Analytic Functions

FIRST_VALUE(col): Returns the value of col on the first row within the window and attaches it to every row
LAST_VALUE(col): Returns the value of col on the last row within the window and attaches it to every row
LAG(col, X): Returns the value of col on the row X rows prior to the current row (default X=1)
LEAD(col, X): Returns the value of col on the row X rows after the current row (default X=1)

monthly_product_sales

month	product_name	sales
2020-01	Apples	300
2020-01	Bananas	500
2020-02	Apples	200
2020-02	Bananas	300
2020-03	Apples	150

```
SELECT
    month,
    product_name,
    sales,
    FIRST_VALUE(sales) OVER(
          PARTITION BY product_name
          ORDER BY month
     ) AS sales_first_month,
    LAG(sales, 1) OVER(
          PARTITION BY product_name
          ORDER BY month
     ) AS prev_month_sales,
    LEAD(sales, 1) OVER(
          PARTITION BY product_name
          ORDER BY month
     ) AS next_month_sales
FROM monthly_product_sales;
```

Navigation Analytic Functions

Results from previous slide's query

month	product_name	sales	sales_first_month	prev_month_sales	next_month_sales
2020-01	Apples	300	300	NULL	200
2020-02	Apples	200	300	300	150
2020-03	Apples	150	300	200	NULL
2020-01	Bananas	500	500	NULL	300
2020-02	Bananas	300	500	500	NULL