Common table expressions

INTRODUCTION TO BIGQUERY

Matthew Forrest
Field CTO

What is a CTE?

```
-- The WITH keywords start the CTE
WITH our_cte AS (
  SELECT column1, column2
  FROM table
)

/* The CTE is contained in parentheses and can be used as a
table in the main query below */
SELECT *
FROM our_cte
```

Subqueries vs. CTEs

Subquery

```
SELECT *
FROM (
  SELECT order_id, status
  FROM ecommerce.ecomm_order_status
) AS subquery
```

CTE

```
WITH cte_name AS (
  SELECT order_id, status
  FROM ecommerce.ecomm_order_status
)
SELECT *
FROM cte_name
```

Writing CTEs

```sql
-- All CTEs begin with the WITH keyword
WITH cte_name AS

-- The CTE is contained in parentheses
(SELECT col_1, col_2
 FROM tablename WHERE col_1 > 10
)

-- The new CTE 'cte_name' is only accessible within the scope of this query

SELECT * FROM cte_name LIMIT 10
```

Using multiple CTEs

```
WITH orders AS (
  SELECT order_id, status
  FROM ecommerce.ecomm_order_status
),
-- Additional CTEs do not reuse the WTIH keyword and are separated by commas
order_details AS (
SELECT orders.*, details.order_items
FROM ecommerce.ecomm_orders details
```
```
```

Using CTEs to filter data

```sql
WITH filtered_data AS (
  SELECT *
  FROM ecommerce.ecomm_orders
  -- Here, we filter the data in our CTE
  WHERE order_status = 'delivered'
)
SELECT order_id, order_status
FROM filtered_data
```

```
| order_id | order_status |
|----------|--------------|
| 1        | delivered    |
| 2        | delivered    |
```

Using CTEs to optimize queries

```sql
WITH precomputed_data AS (
  SELECT order_id, SUM(payment_value) AS total
  FROM ecommerce.ecomm_payments
  GROUP BY order_id
)
SELECT *
FROM precomputed_data
WHERE total > 1000
```

```
| order_id | total    |
|----------|---------|
| 1        | 50000    |
| 2        | 8000     |
```

Using CTEs to join data

```sql
WITH cte1 AS (
  SELECT order_id, status
  FROM ecommerce.ecomm_order_status
),
cte2 AS (
  SELECT order_id, order_items
  FROM ecommerce.ecomm_orders
)
SELECT *
FROM cte1
JOIN cte2 USING (order_id)
```

Let's practice!

INTRODUCTION TO BIGQUERY

Aggregations

INTRODUCTION TO BIGQUERY

Matthew Forrest
Field CTO

Aggregations in BigQuery

```
SELECT
  SUM(sales) AS total_sales,
  AVG(quantity) AS avg_quantity,
  MAX(price) AS max_price,
  MIN(price) AS min_price
FROM sales_data;
```

```
| total_sales | avg_quantity | max_price
|-------------|--------------|----------
| 50          | 8            | 100
```

- Summarize larger datasets
- Identify trends and patterns
- BigQuery is optimized to handle aggregations

GROUP BY and ORDER BY

```sql
SELECT
  -- Later we will group by the 'order_id' column
  order_id,
  SUM(sales) AS order_total
FROM total_sales
-- Here we can GROUP and ORDER our query
GROUP BY order_id
ORDER BY order_total DESC;
```

```
| order_id | order_total |
|----------|-------------|
| 1        | 500         |
| 2        | 850         |
```

COUNT

```sql
SELECT
  category,
  -- COUNT the number of rows returned
  COUNT(order_id) AS record_count
FROM total_sales
GROUP BY category;
```

```
| category    | record_count |
|-------------|--------------|
| shoes       | 238          |
| electronics | 183          |
```

SUM and AVG

```sql
SELECT
  category,
  SUM(cost) AS total_cost,
  AVG(cost) AS average_payment
FROM total_sales
GROUP BY category;
```

```
| category    | total_cost | avg_cost |
|-------------|------------|----------|
| shoes       | 10345      | 54       |
| electronics | 9340       | 34       |
```

MIN and MAX

```sql
SELECT
    MIN(product_photos_qty) as min_photo_count,
    MAX(product_photos_qty) as max_photo_count
FROM ecommerce.ecomm_products;
```

```
| min__photo_count | max__photo_count |
|----------|----------|
| 1    | 20 |
```

COUNTIF

```sql
SELECT
  category,
  -- Counts only if the cost is over $500 grams
  COUNTIF(cost > 500) AS large_items
FROM total_sales
GROUP BY category;
```

```
| category    | large_items |
|-------------|-------------|
| shoes       | 2           |
| electronics | 35          |
```

HAVING

```sql
SELECT
category,
COUNT(order_id) as orders
FROM total_sales
-- Here we add the condition for item categories with an average cost of over $75
HAVING AVG(cost) > 75;
```

```
| category    | orders |
|-------------|--------|
| shoes       | 25     |
| electronics | 98     |
```

ANY_VALUE

```sql
SELECT
  order_id,
  -- Will return an arbitrary category
  ANY_VALUE(category) as cat
  -- Returns the category with the highest cost
  ANY_VALUE(category HAVING MAX cost) AS max_cat
FROM total_sales
GROUP BY order_id;
```

```
| order_id | cat       | max_cat     |
|----------|-----------|-------------|
| 1        | shoes     | electronics |
| 2        | household | exercise    |
```

Let's practice!

INTRODUCTION TO BIGQUERY

Special aggregations in BigQuery

INTRODUCTION TO BIGQUERY

Matthew Forrest
Field CTO

Introduction to special aggregates

Function	Category	Description
`APPROX_COUNT_DISTINCT` , `APPROX_QUANTILES` , `APPROX_TOP_COUNT` , `APPROX_TOP_SUM`	Approximate Aggregation	Provide estimates for certain calculations, reducing processing time and resource consumption.
`ARRAY_CONCAT_AGG` , `STRING_AGG`	Array and String Manipulation	Collect, concatenate, and manipulate arrays and strings.
`LOGICAL_AND` , `LOGICAL_OR`	Logical Operations	Evaluate logical AND and OR operations on a set of boolean expressions.

ARRAY_CONCAT_AGG

- `ARRAY_CONCAT_AGG()` addresses the limitations of `ARRAY_AGG()`

```
SELECT
  order_id,
  ARRAY_CONCAT_AGG(order_items) AS all_items
FROM sales_data
GROUP BY order_id;
```

```
| order_id | all_items                          |
|----------|------------------------------------|
| 1        | [shoes, electronics]               |
| 2        | [electronics, household, clothing] |
```

STRING_AGG

- `STRING_AGG()` concatenates strings into a single string

```
SELECT STRING_AGG(customer_id, ', ') AS all_customers
FROM sales_data
WHERE delivery_date
BETWEEN CURRENT_TIMESTAMP()
AND TIMESTAMP_ADD(CURRENT_TIMESTAMP(), INTERVAL 3 DAY) ;
```

```
| all_customers   |
|-----------------|
| 123, 456, 789   |
```

APPROX_COUNT_DISTINCT

- `APPROX_COUNT_DISTINCT()` gives quick estimation of the count of distinct values

```
SELECT
  customer_id,
  APPROX_COUNT_DISTINCT(order_id) AS unique_orders
FROM sales_data
GROUP BY customer_id;
```

```
| customer_id | unique_orders |
|-------------|---------------|
| 1           | 5             |
| 2           | 2             |
```

APPROX_QUANTILES

- `APPROX_QUANTILES()` gives approximate quantile values

```
SELECT
  category,
  APPROX_QUANTILES(value, 4) AS quartiles
FROM sales_data
GROUP BY category;
```

```
| category    | quartiles          |
|-------------|--------------------|
| shoes       | [25, 40, 55, 100] |
| electronics | [10, 40, 95, 300] |
```

APPROX_TOP_COUNT

- `APPROX_TOP_COUNT()` identifies the top K elements based on their occurrence

```
SELECT
  category,
  APPROX_TOP_COUNT(customer_id, 3) AS customers
FROM sales_data
GROUP BY category;
```

```
| category    | customers    |
|-------------|-------------|
| shoes       | [1, 7, 19]  |
| electronics | [8, 19, 22] |
```

APPROX_TOP_SUM

- `APPROX_TOP_SUM(el, weight, K)` finds the top `K` elements `el` based on `weight`

```
SELECT
  seller_id,
  APPROX_TOP_SUM(item_id, cost, 3) AS top_items
FROM sales_data
GROUP BY seller_id;
```

```
| seller_id | top_items |
|-----------|-----------|
| 1         | 9203      |
| 2         | 8002      |
```

LOGICAL_AND and LOGICAL_OR

```
SELECT
  customer_id,
  -- true if ALL are true
  LOGICAL_AND(order_status = 'shipped') AS all_shipped,
  -- true if at least one is true
  LOGICAL_OR(order_status = 'shipped') AS one_shipped
FROM sales_data
GROUP BY customer_id;
```

```
| customer_id | all_shipped | one_shipped |
|-------------|-------------|-------------|
| 1           | false       | true        |
| 2           | true        | true        |
```

Let's practice!

INTRODUCTION TO BIGQUERY

WINDOW Functions

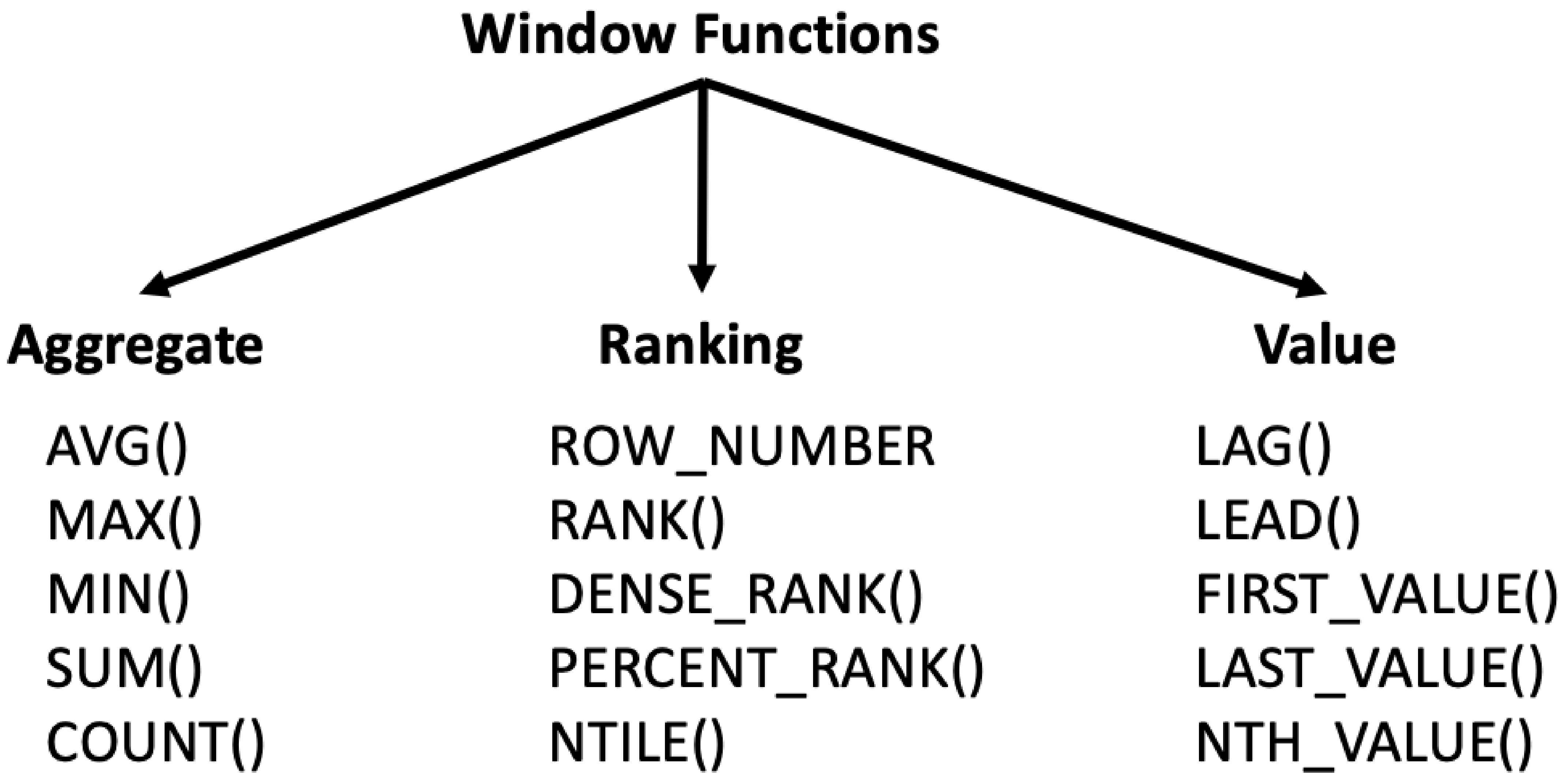
INTRODUCTION TO BIGQUERY

Matthew Forrest
Field CTO

What are WINDOW functions

Row	order_id	order_purchase_timestamp	price	rolling_avg
1	2e7a8482f6fb09756ca50c10d...	2016-09-04 21:15:19	32.9	32.9
2	2e7a8482f6fb09756ca50c10d...	2016-09-04 21:15:19	39.99	72.89
3	e5fa5a7210941f7d56d0208e4...	2016-09-05 00:15:34	59.5	132.39
4	bfbd0f9bdef84302105ad712d...	2016-09-15 12:16:38	44.99	144.48
5	bfbd0f9bdef84302105ad712d...	2016-09-15 12:16:38	44.99	149.48
6	bfbd0f9bdef84302105ad712d...	2016-09-15 12:16:38	44.99	134.97
7	71303d7e93b399f5bcd537d12...	2016-10-02 22:07:52	100	189.98
8	3b697a20d9e427646d925679...	2016-10-03 09:44:50	29.9	174.89
9	be5bc2f0da14d8071e2d45451...	2016-10-03 16:56:50	21.9	151.8
10	65d1e226dfaeb8cdc42f66542...	2016-10-03 21:01:41	21.5	73.3
11	a41c8759fbe7aab36ea07e038...	2016-10-03 21:13:36	36.49	79.89

When to use WINDOW functions



[1] https://towardsdatascience.com/a-guide-to-advanced-sql-window-functions-f63f2642cbf9

WINDOW structure, PARTITION and ORDER BY

```
SELECT
  customer_id,
  order_date,
  order_total,
  ROW_NUMBER() OVER(
    PARTITION BY customer_id
    ORDER BY order_date
  ) AS order_sequence
FROM orders;
```

- `ROW_NUMBER()` : The window function that returns the row number
- `OVER()` : Defines the window frame
- `PARTITION BY customer_id` : Groups the data by customer
- `ORDER BY order_date` : Orders data **within** each partition
- `order_sequence` : Result of the window function operation

RANK and PERCENT_RANK

```sql
SELECT
  product_id,
  product_photos_qty,
  -- Ordinal rank for each row
  RANK() OVER(
    ORDER BY product_photos_qty DESC
  ) as rank,
  -- Percentile rank for each row
  PERCENT_RANK() OVER(
    ORDER BY product_photos_qty
  ) as percent
FROM ecommerce.ecomm_products
ORDER BY product_photos_qty DESC;
```

product_photos_qty	rank	percent
20	1	1.0
19	2	0.99969066105...
18	3	0.999907198317...
18	3	0.999907198317...
17	5	0.999690661057...
17	5	0.999690661057...
17	5	0.999690661057...
17	5	0.999690661057...
17	5	0.999690661057...
17	5	0.999690661057...
17	5	0.999690661057...
[illegible]	[illegible]	[illegible]

LAG and LEAD

```
SELECT
  product_id,
  -- Returns value from previous row
  LAG(product_photos_qty) OVER(
    ORDER BY product_photos_qty
  ) as lag,
  product_photos_qty,
  -- Returns value from next row
  LEAD(product_photos_qty) OVER(
    ORDER BY product_photos_qty
  ) as lead
FROM ecommerce.ecomm_products
ORDER BY product_photos_qty DESC;
```

lag	product_photos_qty	lead
19	20	null
18	19	20
18	18	19
17	18	18
17	17	17
17	17	17
15	17	17
17	17	18
17	17	17
17	17	17
17	17	17

RANGE BETWEEN and CURRENT ROW

```
SELECT
  order_id,
  order_timestamp,
  SUM(cost) OVER(
    ORDER BY order_timestamp
    ROWS BETWEEN 2 PRECEDING
    AND CURRENT ROW) as rolling_avg
FROM sales_data
ORDER BY order_timestamp
```

Row-based bounding options:

- `UNBOUNDED PRECEDING` : All rows before
- `UNBOUNDED FOLLOWING` : All rows after
- `[INT] ROWS PRECEDING` : Specific number of rows before
- `[INT] ROWS FOLLOWING` : Specific number of rows after

QUALIFY

```
SELECT
  product_id,
  product_photos_qty,
  RANK() OVER(
    ORDER BY product_photos_qty DESC
  ) as rank
FROM ecommerce.ecomm_products
-- Filter using QUALIFY
QUALIFY rank < 4
ORDER BY product_photos_qty DESC;
```

product_photos_qty	rank
20	1
19	2
18	3
18	3

- Can't use `HAVING` as we are not aggregating

Let's practice!

INTRODUCTION TO BIGQUERY