Output from CASE WHEN

2. Compute running total and cumulative frequency

Running total can be useful when we are interested in the total sum (but not individual value) at a given point for potential analysis population segmentation and outlier identification.

The following showcases how to calculate the running total and cumulative frequency for the variable NUM_VAR,

```sql
--- 2) Running total/frequency
SELECT
  DAT.NUM_VAR,
  SUM(NUM_VAR) OVER (PARTITION BY JOIN_ID) AS TOTAL_SUM,
  ROUND(CUM_SUM / SUM(NUM_VAR) OVER (PARTITION BY JOIN_ID), 4) AS CUM_FREQ
FROM
(
  SELECT
        T.*,
        SUM(NUM_VAR) OVER (ORDER BY NUM_VAR ROWS UNBOUNDED PRECEDING) AS CUM_SUM,
        CASE WHEN ID_VAR IS NOT NULL THEN '1' END AS JOIN_ID
  FROM CURRENT_TABLE    T
) DAT
ORDER BY CUM_FREQ
```

running_total.sql hosted with ♡ by GitHub view raw

NUM_VAR	TOTAL_SUM	CUM_FREQ
62.71	24458.96	0.0026
62.71	24458.96	0.0051
62.74	24458.96	0.0077
83.05	24458.96	0.0111
99.66	24458.96	0.0152
135.13	24458.96	0.0207
135.13	24458.96	0.0262
135.13	24458.96	0.0317
198.2	24458.96	0.0398
212.35	24458.96	0.0485
302.44	24458.96	0.0609
318.53	24458.96	0.0739
424.99	24458.96	0.0913
22226.19	24458.96	1

Output for cumulative frequency

Here is our output (on the left).

Two tricks here, (1) SUM over ROWS UNBOUNDED PRECEDING will calculate the sum of all prior values to this point; (2) create a JOIN_ID to calculate the total sum.

We use the window function for this calculation, and from the cumulative frequency, it is not hard to spot the last record as an outlier.

3. Find the record(s) with extreme values without self joining

So our task is to return the row(s) with the largest NUM_VAR value for each unique ID. An intuitive query is to first find the max value for each ID using group by, and then self join on ID and the max value. Yet a more concise way would be,

```
--- 3) Find the record having a number calculated by analytic functions (e.g., MAX) without sel
SELECT *
FROM
(
  SELECT
    DAT.*,
    CASE WHEN (NUM_VAR = MAX(NUM_VAR) OVER (PARTITION BY ID_VAR)) THEN 'Y' ELSE 'N' END AS MAX_
  FROM
    CURRENT_TABLE     DAT
) DAT2
WHERE MAX_NUM_IND = 'Y'
```

Records with the max value

this query should give us the following output, showing rows having the max NUM_VAR grouped by ID,

ID_VAR	SEQ_VAR	EMPTY_STR_VAR	NULL_VAR	NA_STR_VAR	NUM_VAR	DATE_VAR1	DATE_VAR2	MAX_NUM_IND
19017	3		(null)	NA	424.99	11/2/2018	11/30/2018	Y
19064	1		(null)	NA	135.13	1/28/2019	1/28/2019	Y
19064	2		(null)	NA	135.13	1/30/2019	1/30/2019	Y
19064	3		(null)	NA	135.13	1/26/2019	1/26/2019	Y
19228	3	S	(null)	NA	62.74	3/29/2019	3/29/2019	Y
19272	1		(null)	NA	22226.19	2/24/2019	3/22/2019	Y

Output for records with the max NUM_VAR value

4. Conditional WHERE clause

Everyone knows the WHERE clause in SQL for subsetting. In fact, I find myself using conditional WHERE clause more often. With the toy table, for instance, we want only

to keep the rows satisfying the following logic,

— if SEQ_VAR in (1, 2, 3) & diff(DATE_VAR2, DATE_VAR1)≥ 0

— elif SEQ_VAR in (4, 5, 6) & diff(DATE_VAR2, DATE_VAR1) ≥1

— else diff(DATE_VAR2, DATE_VAR1) ≥2

Now the conditional WHERE clause comes in handy,

```sql
-- 4) Conditional where clause
SELECT
  DAT.ID_VAR,
  DAT.SEQ_VAR,
  DAT.NUM_VAR,
  DATE_VAR1,
  DATE_VAR2,
  TRUNC(DATE_VAR2) - TRUNC(DATE_VAR1) AS LAG_IN_DATES
FROM
  CURRENT_TABLE      DAT
WHERE
  (TRUNC(DATE_VAR2) - TRUNC(DATE_VAR1)) >= CASE WHEN SEQ_VAR IN (1,2,3) THEN 0 WHEN SEQ_VAR IN
ORDER BY ID_VAR, SEQ_VAR
```

ID_VAR	SEQ_VAR	NUM_VAR	DATE_VAR1	DATE_VAR2	LAG_IN_DATES
19017	1	198.2	11/2/2018	11/30/2018	28
19017	2	212.35	11/2/2018	11/30/2018	28
19017	3	424.99	11/2/2018	11/30/2018	28
19017	4	318.53	11/2/2018	11/30/2018	28
19017	5	302.44	11/2/2018	11/30/2018	28
19064	1	135.13	1/28/2019	1/28/2019	0
19064	2	135.13	1/30/2019	1/30/2019	0
19064	3	135.13	1/26/2019	1/26/2019	0
19228	1	62.71	3/25/2019	3/25/2019	0
19228	2	62.71	3/27/2019	3/27/2019	0
19228	3	62.74	3/29/2019	3/29/2019	0
19272	1	22226.19	2/24/2019	3/22/2019	26

Output for conditional where clause

The logic aforementioned should eliminate the sequences 4, 5 of ID = 19064 because the difference between date2 and date1 = 0, and this is exactly what the query returns

above.

5. Lag() and Lead() to work with consecutive rows

Lag (looking at the previous row) and Lead (looking at the next row) probably are two of the most used analytic functions in my day-to-day work. In a nutshell, these two functions allow users to query more than one row at a time without self-joining. More detailed explanations can be found here.

Let's say, we want to compute the difference in NUM_VAR between two consecutive rows (sorted by sequences),

```sql
--- 5) LAG() or LEAD() function
SELECT
 DAT.ID_VAR,
 DAT.SEQ_VAR,
 DAT.NUM_VAR,
 NUM_VAR - PREV_NUM AS NUM_DIFF
FROM
(
        SELECT
         T.*,
         LAG(NUM_VAR, 1, 0) OVER (PARTITION BY ID_VAR ORDER BY SEQ_VAR) AS PREV_NUM
        FROM
         CURRENT_TABLE     T
)  DAT
ORDER BY ID_VAR, SEQ_VAR
```

lag.sql hosted with ♡ by GitHub view raw

The LAG() function returns the prior row, and if there is none (i.e., the first row of each ID), the PREV_NUM is coded as 0 to compute the difference shown as NUM_DIFF below,

ID_VAR	SEQ_VAR	NUM_VAR	PREV_NUM	NUM_DIFF
19017	1	198.2	0	198.2
19017	2	212.35	198.2	14.15
19017	3	424.99	212.35	212.64
19017	4	318.53	424.99	-106.46
19017	5	302.44	318.53	-16.09
19064	1	135.13	0	135.13
19064	2	135.13	135.13	0
19064	3	135.13	135.13	0
19064	4	83.05	135.13	-52.08
19064	5	99.66	83.05	16.61
19228	1	62.71	0	62.71

2