


this code snippet returns,

ID_VAR	NULL_VAR	COALESCE_NULL_VAR
19017	(null)	MISSING
19017	(null)	MISSING
19017	(null)	MISSING
19017	(null)	MISSING
19064	(null)	MISSING
19064	(null)	MISSING
19064	(null)	MISSING
19064	(null)	MISSING
19064	(null)	MISSING
19228	(null)	MISSING
19228	(null)	MISSING
19228	(null)	MISSING
19228	(null)	MISSING
19272	(null)	MISSING

COALESCE() to recode NULL

One important note, however, is that in databases, *missing values* can be encoded in various ways besides NULL. For instance, they could be empty string/blank space (e.g., EMPTY\_STR\_VAR in our table), or a character string 'NA' (e.g., NA\_STR\_VAR in our table). In these cases, COALESCE() would not work, but they can be handled with the CASE WHEN statement,

coalesce casewhen.sql hosted with  by GitHub

```
1 --- However, COALESCE() NOT WORK for Empty or NA string, instead, use CASE WHEN
2 SELECT
3 ID_VAR,
4 EMPTY_STR_VAR,
5 COALESCE(EMPTY_STR_VAR, 'MISSING') AS COALESCE_EMPTY_STR_VAR,
6 CASE WHEN EMPTY_STR_VAR = ' ' THEN 'EMPTY_MISSING' END AS CASEWHEN_EMPTY_STR_VAR,
7 NA_STR_VAR,
8 CASE WHEN NA_STR_VAR = 'NA' THEN 'NA_MISSING' END AS CASEWHEN_NA_STR_VAR
9 FROM
10 CURRENT_TABLE
11 ORDER BY ID_VAR
```

view raw

CASE WHEN to re-code empty or NA

ID_VAR	EMPTY_STR_VAR	COALESCE_EMPTY_STR_VAR	CASEWHEN_EMPTY_STR_VAR	NA_STR_VAR	CASEWHEN_NA_STR_VAR
19228	S	S	S	NA	NA_MISSING
19228	S	S	S	NA	NA_MISSING
19228	S	S	S	NA	NA_MISSING
19272			EMPTY_MISSING	NA	NA_MISSING

## 2. Compute running total and cumulative frequency

Output from CASE WHEN

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,

```
1 --- 2) Running total/frequency
2 SELECT
3   DAT.NUM_VAR,
4   SUM(NUM_VAR) OVER (PARTITION BY JOIN_ID) AS TOTAL_SUM,
5   ROUND((CUM_SUM / SUM(NUM_VAR) OVER (PARTITION BY JOIN_ID), 4) AS CUM_FREQ
6 FROM
7   (
8     SELECT
9       T.*,
10      SUM(NUM_VAR) OVER (ORDER BY NUM_VAR ROWS UNBOUNDED PRECEDING) AS CUM_SUM,
11      CASE WHEN ID_VAR IS NOT NULL THEN '1' END AS JOIN_ID
12 FROM CURRENT_TABLE T
13 ) DAT
14 ORDER BY CUM_FREQ
```

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,

```
1 --- 3) Find the record having a number calculated by analytic functions (e.g., MAX) without self
2 SELECT *
3 FROM
4 (
5     SELECT
6         DAT.*,
7         CASE WHEN (NUM_VAR = MAX(NUM_VAR) OVER (PARTITION BY ID_VAR)) THEN 'Y' ELSE 'N' END AS MAX
8     FROM
9         CURRENT_TABLE DAT
10 ) DAT2
11 WHERE MAX_NUM_IND = 'Y'
```

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,

```
1  -- 4) Conditional where clause
2  SELECT
3    DAT.ID_VAR,
4    DAT.SEQ_VAR,
5    DAT.NUM_VAR,
6    DATE_VAR1,
7    DATE_VAR2,
8    TRUNC(DATE_VAR2) - TRUNC(DATE_VAR1) AS LAG_IN_DATES
9  FROM
10   CURRENT_TABLE DAT
11 WHERE
12   (TRUNC(DATE_VAR2) - TRUNC(DATE_VAR1)) >= CASE WHEN SEQ_VAR IN (1,2,3) THEN 0 WHEN SEQ_VAR IN
13   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),

```
1 --- 5) LAG() or LEAD() function
2 SELECT
3   DAT.ID_VAR,
4   DAT.SEQ_VAR,
5   DAT.NUM_VAR,
6   NUM_VAR - PREV_NUM AS NUM_DIFF
7 FROM
8   (
9     SELECT
10      T.*,
11      LAG(NUM_VAR, 1, 0) OVER (PARTITION BY ID_VAR ORDER BY SEQ_VAR) AS PREV_NUM
12 FROM
13   CURRENT_TABLE T
14 ) DAT
15 ORDER BY ID_VAR, SEQ_VAR
```

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



