


coalesce.sql hosted with  by GitHub

view raw

```
1  ---- 1) COALESCE() to recode the NULL value to the character string MISSING
2  SELECT
3  ID_VAR,
4  NULL_VAR,
5  COALESCE(NULL_VAR, 'MISSING') AS RECODE_NULL_VAR
6  FROM
7  CURRENT_TABLE
8  ORDER BY ID_VAR
```

When it comes to re-coding missing values, the COALESCE() function is our secret sauce, which, under this circumstance, re-codes the NULL to whatever value specified in the second argument. For our example, we can re-code the NULL_VAR to a character value 'MISSING',

1. COALESCE() to recode NULL / missing data

Toy data table (with variable definitions)

ID_VAR	SEQ_VAR	EMPTY_STR_VAR	NULL_VAR	NA_STR_VAR	NUM_VAR	DATE_VAR1	DATE_VAR2
19017	1	coded as empty)	string (missing value	string (missing value	float number	date1	date2
19017	2	coded as NULL)	coded as 'NA')				
19017	3						
19017	4						
19017	5						
19064	1						
19064	2						
19064	3						
19064	4						
19064	5						
19228	1						
19228	2						
19228	3						
19272	1						

Hence rather than transferring huge datasets into Python or R, the first step of analytics should be using SQL to gain informative insights from our data.

Working in real-world relational databases, SQL is way more than just SELECT, JOIN, ORDER BY statements. In this blog, I will discuss 6 tips (and one Bonus tip) to make your analytics work more efficient with SQL and its integrating with other programming languages like Python and R.

For this exercise, we will work with Oracle SQL on the toy data table below, which consists of multiple types of data elements,


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,

```
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
```

coalesce casewhen.sql hosted with  by GitHub

[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

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,

running total.sql hosted with  by GitHub

```
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
```

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,

```
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

