6 SQL Tricks Every Data Scientist Should Know

Part 1 of SQL tricks to make your analytics work more efficient

Yi Li

Mar 12 · 6 min read ★

Data scientists/analysts should know SQL, in fact, all professionals working with data and analytics should know SQL. To some extent, SQL is an under-rated skill for data science because it has been taken for granted as a necessary yet uncool way of extracting data out from the database to feed into pandas and {tidyverse} — fancier ways to wrangle your data.

Photo Source

However, with massive data being collected and churned out every day in the industries, as long as the data reside in a SQL compliant database, SQL is still the most proficient tool to help you investigate, filter and aggregate to get a thorough understanding of your data. By slicing and dicing with SQL, analysts are allowed to possibly identify patterns that worth further looking into, which oftentimes leads to redefining the analysis population and variables to be considerably smaller (than the initial scope).

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,

ID_VAR	SEQ_VAR	EMPTY_STR_VAR	NULL_VAR	NA_STR_VAR	NUM_VAR	DATE_VAR1	DATE_VAR2
ID, used as the join key	sequence number	string (missing value coded as empty)	string (missing value coded as NULL)	string (missing value coded as 'NA')	float number	date1	date2
19017	1		(null)	NA	198.2	11/2/2018	11/30/2018
19017	2		(null)	NA	212.35	11/2/2018	11/30/2018
19017	3		(null)	NA	424.99	11/2/2018	11/30/2018
19017	4		(null)	NA	318.53	11/2/2018	11/30/2018
19017	5		(null)	NA	302.44	11/2/2018	11/30/2018
19064	1		(null)	NA	135.13	1/28/2019	1/28/2019
19064	2		(null)	NA	135.13	1/30/2019	1/30/2019
19064	3		(null)	NA	135.13	1/26/2019	1/26/2019
19064	4		(null)	NA	83.05	1/26/2019	1/26/2019
19064	5		(null)	NA	99.66	1/31/2019	1/31/2019
19228	1	S	(null)	NA	62.71	3/25/2019	3/25/2019
19228	2	S	(null)	NA	62.71	3/27/2019	3/27/2019
19228	3	S	(null)	NA	62.74	3/29/2019	3/29/2019
19272	1		(null)	NA	22226.19	2/24/2019	3/22/2019

Toy data table (with variable definitions)

1. COALESCE() to recode NULL / missing data

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 the NULL value to the character string MISSING
SELECT
    ID_VAR,
    NULL_VAR,
    COALESCE(NULL_VAR, 'MISSING') AS RECODE_NULL_VAR
FROM
  CURRENT_TABLE
ORDER BY ID_VAR
```

coalesce.sql hosted with ♡ by GitHub view raw

this code snippet returns,

ID_VAR	NULL_VAR	COALESCE_NULL_VAR
19017	(null)	MISSING
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
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,

```sql
--- However, COALESCE() NOT WORK for Empty or NA string, instead, use CASE WHEN
SELECT
  ID_VAR,
  EMPTY_STR_VAR,
  COALESCE(EMPTY_STR_VAR, 'MISSING') AS COALESCE_EMPTY_STR_VAR,
  CASE WHEN EMPTY_STR_VAR = ' ' THEN 'EMPTY_MISSING' END AS CASEWHEN_EMPTY_STR_VAR,

  NA_STR_VAR,
  CASE WHEN NA_STR_VAR = 'NA' THEN 'NA_MISSING' END AS CASEWHEN_NA_STR_VAR
FROM
  CURRENT_TABLE
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		NA	NA_MISSING
19228	S	S		NA	NA_MISSING
19228	S	S		NA	NA_MISSING
19272			EMPTY_MISSING	NA	NA_MISSING

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

19228	2	62.71	62.71	0
19228	3	62.74	62.71	0.03
19272	1	22226.19	0	22226.19

Output from LAG()

6. Integrate SQL query with Python and R

The prerequisite of integrating SQL queries into Python and R is to establish the database connections via ODBC or JDBC. Since this is beyond the scope of this blog, I will not discuss it here, however, more details regarding how to (create ODBC or JDBC connections) can be found here.

Now, assuming that we already connected Python and R to our database, the most straightforward way of using query in, say Python, is to copy and paste it as a string, then call pandas.read_sql(),

```
my_query = "SELECT * FROM CURRENT_TABLE"
sql_data = pandas.read_sql(my_query, connection)
```

Well, as long as our queries are short and finalized with no further changes, this method works well. However, what if our query has 1000 lines, or we need to constantly update it? For these scenarios, we would want to read *.sql* files directly into Python or R. The following demonstrates how to implement a getSQL function in Python, and the idea is the same in R,

```
import pandas as pd
def getSQL(sql_query,
           place_holder_str,
           replace_place_holder_with,
           database_con):
    '''
    Args:
        sql_query: sql query file
        place_holder_str: string in the original sql query that is to be replaced
        replace_place_holder_with: real values that should be put in
        database_con: connection to the database
    '''

    sqlFile = open(sql_query, 'r')
    sqlQuery = sqlFile.read()

```

```
    sqlQuery = sqlQuery.replace(place_holder_str, replace_place_holder_with)

    df = pd.read_sql_query(sqlQuery, database_con)

    database_con.close()

    return df
```

Here, the first arg sql_query takes in a separate standalone *.sql* file that can be easily maintained, like this,

```
SELECT
  *
FROM
  CURRENT_TABLE   DAT
WHERE
  ID_VAR IN ('ID_LIST')
ORDER BY ID_VAR, SEQ_VAR
```

SQL_FILE.sql hosted with ♡ by GitHub view raw

The "ID_LIST" is a placeholder string for the values we are about to put in, and the getSQL( ) can be called using the following code,

```
seq12_df = getSQL('SQL_FILE.sql', 'ID_LIST', "','".join(['19228', '19272']), database_con=conn)
```

Bonus tip, regular expression in SQL

Even though I do not use regular expression in SQL all the time, it sometimes can be convenient for text extraction. For instance, the following code shows a simple example of how to use REGEXP_INSTR( ) to find and extract numbers (see here for more details),

```
-- Find and extract numbers between 0 - 9 that consecutively happens 5 times
SELECT
  SUBSTRING(LONG_TEXT, REG_IDX, REG_IDX+5) AS NUMBER_LIST_FOUND
FROM
(
  SELECT
    REGEXP_INSTR(LONG_TEXT, '[0-9]{5}') AS REG_IDX,
    LONG_TEXT
```

```
    FROM
      BONUS
) DAT
```

regularExpression.sql hosted with ♡ by GitHub view raw

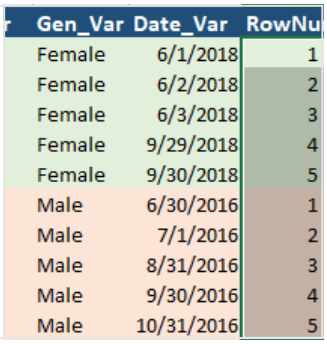
I hope you find this blog helpful, and the full code along with the toy dataset is available in my github. 😀

P.S. head over to Part 2 of this mini-series for more SQL analytics tips.

Extra 4 SQL Tricks Every Data Scientist Should Know

Getting more out of SQL to step up your analytics work

towardsdatascience.com



Gen_Var	Date_Var	RowNu
Female	6/1/2018	1
Female	6/2/2018	2
Female	6/3/2018	3
Female	9/29/2018	4
Female	9/30/2018	5
Male	6/30/2016	1
Male	7/1/2016	2
Male	8/31/2016	3
Male	9/30/2016	4
Male	10/31/2016	5

Sql Data Science R Python Analytics