Intermediate SQL

BA770 Lab Session

Questrom School of Business, Boston University

August 5, 2019

Topics

- **CASE** statement
- Subquery
- Correlated queries, nested queries, common table expressions
- Window function

CASE

- **CASE** in **SELECT** - categorizing data
- **CASE** in **WHERE** - filtering data
- **CASE** with aggregation functions - aggregating data

- Treat a **CASE** statement as a column in the query.

CASE in SELECT

- **CASE WHEN** condition1 **THEN** value1
  WHEN condition2 **THEN** value2
  .....
  (**ELSE** default_value) **END AS** new_col_name

- **CASE WHEN...THEN...END** clause helps do multiple if-then-else statements in a simplified way.
- If you do not include an **ELSE** statement, then values that not satisfy any condition will be marked as **NULL**. It's the same as specifying **ELSE NULL**.
- Make sure you add specific filters in the **WHERE** clause that exclude all scenarios you do not want to show in your result.

CASE in WHERE

- **CASE WHEN** condition1 **THEN** value1
  WHEN condition2 **THEN** value2
  .....
  (**ELSE** default_value) **END** condition

- A **CASE** statement in **WHERE** clause is treated as a column to be filtered.
- Include an entire **CASE** statement in **WHERE** clause. Using the alias of a **CASE** statement instead will lead to error.
- Do not alias the statement in **WHERE**.
- A commonly used condition is **IS (NOT) NULL**.

CASE with COUNT

- **COUNT** (**CASE WHEN** condition
  THEN value **END**) **AS** new_col_name

- Aggregate data based on the result of a logical test ('condition').
- 'value' can be anything you like as SQL is counting the number of rows returned by **CASE** statement.
- **ELSE** statement is omitted and assumed to be **NULL**.

CASE with SUM/AVG

- **SUM/AVG** (**CASE WHEN** condition
  THEN value **END**) **AS** new_col_name

- Different from working with **COUNT**, 'value' here should be the exact value that you are going to add up or take average of.
- But similarly, **ELSE** statement is omitted and assumed to be **NULL**.
- You may use **ROUND**(statement, decimal) function to make your results more readable.

CASE with AVG to calculate percent

- **AVG** (**CASE WHEN** condition_is_met **THEN** 1
  WHEN condition_is_not_met **THEN** 0 **END**)
  AS new_col_name

- With this approach, it's important to accurately specify which records count as 1 and which as 0.

Subquery

- Subquery in **WHERE** - filtering results based on information calculated separately beforehand
- Subquery in **FROM** - 1) restructuring and transforming data 2) calculating aggregates of aggregates
- Subquery in **SELECT** - performing complex mathematical calculations

- A subquery is a query nested inside another query that can be run on its own.
- A subquery can be placed in any part of a query.
- All subqueries are processed before the main query.

Subquery in WHERE

- **SELECT** column
  FROM table
  WHERE column >/</=/etc. (subquery)
- **SELECT** column
  FROM table
  WHERE column (**NOT**) **IN** (subquery)

- The first query treats the subquery as a single, aggregate value.
- The second query uses a filtering list generated by the subquery for filtering results.
- Remember to wrap the subquery with parentheses.
- Feel free to add extra filtering conditions.

Subquery in FROM

- **SELECT** column
  FROM (subquery) **AS** subquery_alias
- **SELECT** column
  FROM table **AS** table_alias
  INNER JOIN (subquery) **AS** subquery_alias
  ON table_alias.column1 = subquery_alias.column2

- You can take a subquery in a **FROM** clause as a new table that you are going to retrieve information from.
- You can create multiple subqueries in one **FROM** statement, but make sure to use alias and join them properly.
- You can join (any type of join - **INNER**, **LEFT**, **FULL**, etc.) a subquery to any existing table.
- Remember to wrap the subquery with parentheses.

Subquery in SELECT

- **SELECT** columns
  (subquery) **AS** subquery_alias
  FROM table

- A subquery in **SELECT** needs to return a single value.
- Place filters (**WHERE** statements) correctly in both the main query and the subquery.
- Remember to wrap the subquery with parentheses.

Correlated Subquery

- Use values from the outer query to generate the final result.
- Dependent on the main query; cannot be executed on its own.
- Re-executed for every row generated in the final result.
- Slow down query performance.

Nested Subquery

- Subquery inside another subquery.
- Can be correlated or uncorrelated, or a combination of the two.

Common Table Expression (CTE)

- **WITH** table_name1 **AS** (
  SELECT columns
  FROM table1
  ……),
  WITH table_name2 **AS** (
  SELECT columns
  FROM table2
  ……),
  ……

  SELECT columns
  FROM table_name1
  ……

Common Table Expression (CTE)

- Table declared before the main query.
- Name it using **WITH** statement and reference it by name later.
- Advantages:
  1) Improve readability and information accessibility.
  2) Save running time: Run once then stored in memory.
  3) Able to reference CTEs declared earlier.
  4) Able to reference itself.

Comparison and Summary

Join	**Correlated Subquery**	**Multiple/Nested Subquery**	**Common Table Expression**
Combine multiple tables	Match subqueries and tables	Multi-step transformations	Organize subqueries
Simple operations	Avoid limits of joins; high processing time	Improve accuracy and reproducibility	Can reference other CTEs

Table: Differentiating Techniques

Window Function

- Window functions perform calculations on a result set that already been generated, which is referred to as a "window".
- Window functions are especially useful when generating time-series results, e.g. moving average and running total.
- Window functions work with the **OVER** clause. The **OVER** clause tells SQL to pass an aggregation value over the existing result set.

OVER with AVG

- **SELECT** ..., **AVG**(expression) **OVER**() **AS** alias
  FROM table

- This query generates an overall average and saves the result in each row in the final dataset.
- Notice the difference between '**SELECT AVG**(column)' and the above query. '**SELECT AVG**(column)' generates a single value, whereas the above query creates a new column and saves the aggregation result in each row.
- Selecting aggregation results along with other columns using '**SELECT** columns, **AVG**(column)' will lead to error. This is because other columns must appear in a **GROUP BY** clause or be used in an aggregate function.

OVER with RANK

- **SELECT** ..., **RANK**() **OVER**(**ORDER BY** expression (**DESC**)) **AS** alias
  FROM table

- This query generates a rank ordered by a specified column, but does not sort the final dataset. If you want to sort the final result, add an **ORDER BY** clause.
- **RANK** creates a column numbering the dataset from highest to lowest or lowest to highest, based on the column specified.
- **RANK** function automatically ties identical values and skips other values in the rank.
- Window functions are processed after the entire query except the final **ORDER BY** statement.

OVER with PARTITION BY

- **SELECT** ..., **AVG/SUM/etc.**(expression) **OVER**(**PARTITION BY** column) **AS** alias
  FROM table

- A partition calculates separate values for different categories established in a partition.
- You may include multiple columns in the **PARTITION BY** clause if necessary.
- **PARTITION BY** can also work with other window functions, like **RANK**.

Sliding Window

- Sliding windows perform calculations relative to the current row of a dataset.
- They are especially useful for calculating running totals, counts, averages, etc.
- Sliding windows can also be partitioned by one or more columns, like a non-sliding window.

Sliding Window

- A sliding window specifies the data for use in the **OVER** clause:

  ROWS BETWEEN <start> **AND** <finish>

  Keywords for <start> and <finish>:
  PRECEDING
  FOLLOWING
  UNBOUNDED PRECEDING
  UNBOUNDED FOLLOWING
  CURRENT ROW

- Examples:
  1) **BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**
  2) **BETWEEN** 10 **PRECEDING AND CURRENT ROW**