Advanced SQL Tutorial

Case Statements, Common Table Expressions, and Pivots

Preliminaries

Goal of the Exercise: The goal of this exercise is to get you more familiar with using Case Statements, Common Table Expressions, and Pivots. The tutorial will walk you through some basic examples of using these powerful tools in SQL. Knowing how to use these tools will benefit you in the future as you continue to use SQL.

Data Required: This tutorial uses the same Pine Valley Database as previous tutorials, available on the course portal. Any required alterations to the data will be made within the tutorial, so for now you can load the data as-is.

Setting up R Studio: Before proceeding with the main part of this tutorial, make sure you have R Studio set up to work with the database. Copy and paste this into an R Script, and substitute in an appropriate file path to get started:

```
setwd("path")
library('RSQLite')
library('sqldf')
conn <- dbConnect(RSQLite::SQLite(), "pineValleyDB1.db")
```

Once you've done so, you're ready to get going with the steps of the tutorial. The steps are broken up into three main sections - Case Statements, Common Table Expressions, and Pivots. Each section has two deliverables to be submitted.

Part 1 Steps - Case Statements

1.1 - Basic CASE Statements

A CASE WHEN or simply a CASE statement runs through a condition or multiple conditions and returns a value when one of those conditions is met. There are two different forms of the case statement. A Simple CASE statement and a Searched CASE statement. The following is the syntax for both types of CASE statements:

Simple CASE statement

```
CASE case_operand
{   WHEN when_operand THEN {statement;}...}...
[   ELSE {statement;}...]
END CASE [label_name];
```

Searched CASE statement:

```
CASE {WHEN boolean_expression THEN {statement;}...}...
[   ELSE {statement;}...]
END CASE [label_name];
```

Let's start off by going through some examples of the Simple CASE statement. Please run the following code:

```
dbGetQuery(conn, "SELECT CustomerState,
          CASE
               WHEN CustomerState = 'NY' THEN 'Yes'
               ELSE 'No'
          END AS FromNewYork
          FROM Customer_T")
```

Notice how the case statement does exactly what it is designed to do. It takes the condition of CustomerState = ‘NY’ and evaluates each row in the column CustomerState to see if any rows meet the condition. When a specific row has that value, it returns the specified value of ‘Yes’. If a row does not have the value ‘NY’, the ELSE keyword is used to specify the value that should be recorded, that being ‘No’. Our END AS keyword sets up the new column where these values will be recorded for each row.

Let's change things up. Copy and paste the following code and run it:

```
dbGetQuery(conn, "SELECT CustomerState,
          CASE
               WHEN CustomerState = 'FL' THEN 'Yes'
               ELSE 'No'
          END AS FromFlorida
          FROM Customer_T")
```

Notice how we changed the CustomerState value to 'FL' and the END AS keyword to 'FromFlorida' and received much different results? Case statements are very malleable and easy to adjust according to what you want. Now onto our first deliverable.

Deliverable 1: Using the Customer_T chart and a Case Statement, change the abbreviated versions of NY, FL, and TX to their full state names, that being New York, Florida, and Texas. The remaining states should be listed as 'Not Required' for their names. The new column where these values should fall under is to be called 'StateName'. The only two columns that are required in this new chart are 'CustomerState' and 'StateName'. Please take a screenshot of your code and submit it when completed. (Note that nesting is required to complete this deliverable. You are limited to 10 nest levels, although only three are necessary)

1.2 - CASE Statements and The Comparison Operator

Comparison operators such as AND can be used to evaluate a condition within a CASE statement. As shown before, once that condition is met, the THEN keyword will specify the resulting value. Should it not be met, the ELSE keyword will specify the resulting value.

Copy and paste the following code and run it in R Studio:

```
dbGetQuery(conn, "SELECT ProductID, ProductStandardPrice,
           CASE
              WHEN ProductStandardPrice >= 1000 AND ProductStandardPrice <= 1650
              THEN 'Expensive'
              WHEN ProductStandardPrice >= 500 AND ProductStandardPrice <= 999
              THEN 'Normal'
              WHEN ProductStandardPrice >= 0 AND ProductStandardPrice <= 499
              THEN 'Cheap'
              ELSE 'Not enough information available'
            END AS ProductPricingCategories
            FROM Product_T")
```

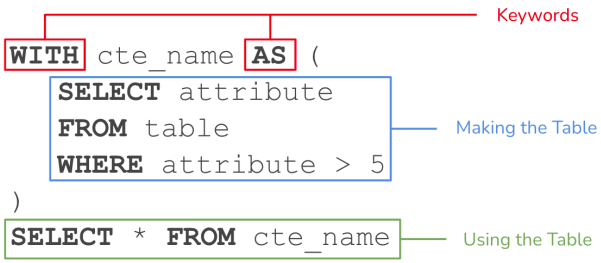
Using comparison operators in conjunction with CASE statements can make classifying information, like product prices in the example above extremely simple and fast. Let's give you a chance to witness this speed on your own with the next deliverable.

Deliverable 2: Using the Payment_T chart and a Case Statement and the comparison operator AND, classify the payment amounts found in the 'PaymentAmount' column of the 'Payment_T' chart according to the discount percentage amount they fall under. 'PaymentAmount' greater than or equal to 1000 have a percentage discount of 30%. 'PaymentAmount' greater than or equal to 500 and less than or equal to 999 have a percentage discount of 20%. Finally, 'PaymentAmount' of greater than or equal to 0 and less than or equal to 499 have a percentage discount of 10%. The percentage discount amounts should be found under the new column title 'PercentageDiscount'. The following is an example of what 30 percent should look like: '30.0'. Other than the 'PercentageDiscount' column, 'PaymentID', 'OrderID', and 'PaymentTypeID' should also be included as columns. (Refer to the above example to see how to include additional columns.) (No ELSE statement is required for this deliverable.)

Part 2 Steps - Common Table Expressions

2.1 Basic CTEs

Common Table Expressions are used to generate temporary “pseudo-tables” for use in a query immediately following them. Their use will become more clear in the coming sections, but for now, the syntax is as follows:



Now that we know the syntax, will start by generating a basic Common Table Expression (CTE). Let’s create a Common Table Expression that will contain the names of customers from New York state, and the name of their city. Copy and paste this code and run it:

```
dbGetQuery(conn, "WITH new_york_customer_cities AS
                  (
                    SELECT CustomerName, CustomerCity
                    FROM customer_t
                    WHERE CustomerState = 'NY'
                  )
                  SELECT * from new_york_customer_cities
                  ")
```

As you can see, we have made a new “temporary” table that shows the name and city for each customer from New York. Note that we can do different things with this table within the dbGetQuery() wrapper; we don’t just have to display the whole table. For instance:

```
dbGetQuery(conn, "WITH new_york_customer_cities AS
                  (
                    SELECT CustomerName, CustomerCity
                    FROM customer_t
                    WHERE CustomerState = 'NY'
                  )
                  SELECT CustomerCity from new_york_customer_cities
                  ")
```

Run this code and examine how the output differs from the previous code. Now, suppose we want to rename the columns in the Common Table Expression to make it look a bit better. Maybe it seems redundant to list “CustomerName” and “CustomerCity” if we already know we are looking at Customers, so let’s instead rename those columns to “Name” and “City”. The great thing about this is we aren’t changing the existing table in any way; we are just setting new names for the columns in the resulting CTE. We can do this with the syntax below:

```
dbGetQuery(conn, "WITH new_york_customer_cities(Name, City) AS
                  (
                    SELECT CustomerName, CustomerCity
                    FROM customer_t
                    WHERE CustomerState = 'NY'
                  )
                  SELECT * from new_york_customer_cities
                  ")
```

Observe that the table now has new column names.

DELIVERABLE 1: Create a CTE called order_people to display the Customer ID and Salesperson ID for each Order ID. Call the first column “OrderNumber”, the second “Buyer”, and the third “Seller”. Display the table in R Studio and screenshot the first 5-10 lines of your output. Submit your screenshot.

2.2 Using Multiple CTEs

Now that we understand the basic operations of a Common Table Expression, let’s try something more advanced - using multiple CTEs.

First, run this code that we saw in a previous tutorial, which was used to “Display all the order records that have at least one of the following items: Dresser, Bookcase, Clock”:

```
dbGetQuery(conn, "SELECT * FROM order_t
                  WHERE orderid IN (
                    SELECT orderid
                    FROM orderline_t
                    WHERE productid IN (
                      SELECT productid
                      FROM product_t
                      WHERE (productdescription LIKE '%Dresser%'
                        OR productdescription LIKE '%Bookcase%'
```

```
                        OR productdescription LIKE '%Clock%')
                  )
                )")
```

Next, copy and paste then run this code:

```
dbGetQuery(conn, "WITH

                  dressers_bookcases_clocks
                  AS
                  (
                    SELECT productid
                    FROM product_t
                    WHERE (productdescription LIKE '%Dresser%'
                      OR productdescription LIKE '%Bookcase%'
                      OR productdescription LIKE '%Clock%')
                  ),

                  orders_of_dressers_bookcases_clocks
                  AS
                  (
                    SELECT orderid
                    FROM orderline_t
                    WHERE productid IN dressers_bookcases_clocks
                  )

                  SELECT * FROM order_t
                  WHERE orderid
                  IN orders_of_dressers_bookcases_clocks
                  ")
```

Here, you can see that we use intermediate CTEs `dressers_bookcases_clocks` and `orders_of_dressers_bookcases_clocks` to gradually build up the answer. We finish with a single `SELECT` statement on the last CTE we made. This achieves the same result, yet uses CTEs instead of nested `SELECT` statements. Which option (nested `SELECT` statements or CTEs) do you find easier to understand or more readable?

Now, consider this code, which was used to "display identification info of all salespeople who have placed orders for customers in NY":

```
dbGetQuery(conn, "SELECT * FROM salesperson_t
                  WHERE salespersonid IN (
```

```
    SELECT salespersonid FROM order_t
    WHERE customerid IN (
      SELECT customerid from customer_t
      WHERE customerstate IN ('NY')
    )
)")
```

Practice Exercise 1: See if you can rewrite this using CTEs instead of nested `SELECT`s.

2.3 Recursion

CTEs can also recursively refer to themselves to be able to progress through hierarchical structures in databases. We won't go through recursion in detail in this tutorial, but we will give a brief overview. Here is a very simple example of recursion with CTEs. It simply prints out the numbers from 1 to 5:

```
dbGetQuery(conn, "WITH

                  count_to_5
                  AS
                  (
                    SELECT 1 as n
                    UNION ALL
                    SELECT n + 1
                    FROM count_to_5
                    WHERE n < 5
                  )

                  SELECT *
                  FROM count_to_5
                  ")
```

Practice Exercise 2: Now, try making it count to 10 instead of 5. Once you've done that, can you make it step-count by 2 each time (the output should be 1, 3, 5, 7, 9, 11)?

2.4 CTEs and Aggregate Functions

Finally, we will use CTEs in combination with Aggregate Functions to yield powerful results. First, we will find the average number of orders per customer. Copy and paste this code and inspect the output:

```
dbGetQuery(conn, "WITH num_orders_by_customer
                  AS
                  (
                    SELECT customerid, COUNT(*) AS num_orders
                    FROM order_t
                    GROUP BY customerid
                  )

                  SELECT AVG(num_orders)
                  FROM num_orders_by_customer
                  ")
```

In the example above, we first create a CTE that lists the number of orders for each customer. It is then trivial to find the average for the number of orders from this CTE. This illustrates how CTEs can be very helpful when using aggregate functions.

DELIVERABLE 2: Use a CTE with an Aggregate Function to find the salesperson that has taken the greatest number of orders (just report their SalespersonID, not their name), and what that number of orders is. Screenshot your output and submit it.

Part 3 Steps - Pivots

Unfortunately for us, there is no PIVOT keyword function in RSQLite. With this being the case, we must manually create the tables using familiar code.

3.1 Pivot Basics

Before constructing any pivot table, make sure you understand and are familiar with how the source table is structured.

Run the following line of code:

```
dbGetQuery(conn, "SELECT * FROM product_t")
```

Upon running the previous query, you will see all the columns from the product_t table. In this example, we will be working to pivot the 'productfinish', and use its values to create the columns in our new table. Again, since we do not have the PIVOT function in RSQLite, we will be manually creating the pivot table using familiar code. Below, you will see the sample code for creating our first column as 'Cherry'. In this example, the code is identifying the instances in which the keyword 'Cherry' appears in the source column 'productfinish'. Whenever an instance of 'Cherry' occurs, the SUM function adds one to the count (this is in lieu of using the COUNT aggregate function which we would have used with the PIVOT function). The count is applied to the new column that we have aliased as Cherry. For our pivot table, we need to combine CASE WHEN statements with SUM in order to count several rows of data.

```
dbGetQuery(conn, "SELECT
           SUM(CASE WHEN productfinish = 'Cherry' then 1 else 0 end)
           AS Cherry,
           FROM product_t")
```

The same process is applied to each of the new columns we want to include in our new pivot table.

```
dbGetQuery(conn, "SELECT
           SUM(CASE WHEN productfinish = 'Cherry' then 1 else 0 end)
           AS Cherry,
           SUM(CASE WHEN productfinish = 'Pine' then 1 else 0 end)
           AS Pine,
           SUM(CASE WHEN productfinish = 'Walnut' then 1 else 0 end)
           AS Walnut,
           SUM (CASE WHEN productfinish = 'Leather' then 1 else 0
end)
```

```
AS Leather,
SUM(CASE WHEN productfinish = 'Birch' then 1 else 0 end)
AS Birch,
SUM (CASE WHEN productfinish = 'Oak' then 1 else 0 end)
AS Oak
SUM (CASE WHEN productfinish IS NULL then 1 else 0 end)
AS None

FROM product_t")
```

It is essential to apply the “FROM table” line at the end of your query, after you have established your columns. Any actions outside of the actual pivot step should be placed after the “FROM table”

In this second example, we will show you how to aggregate your pivot table using the MAX keyword. Here you will display the maximum price for each of the “productfinish” categories.

```
dbGetQuery(conn, "SELECT
          MAX
          (
               CASE WHEN productfinish = 'Cherry'
               THEN productstandardprice else 0 end
          ) AS Cherry,
          MAX
          (
               CASE WHEN productfinish = 'Pine'
               THEN productstandardprice else 0 end
          ) AS Pine,
          MAX
          (
               CASE WHEN productfinish = 'Walnut' THEN
               productstandardprice else 0 end
          ) AS Walnut,
          MAX
          (
               CASE WHEN productfinish = 'Leather' THEN
               productstandardprice else 0 end
          ) AS Leather,
          MAX
          (
               CASE WHEN productfinish = 'Birch' then
               productstandardprice else 0 end
          ) AS Birch,
```

```
MAX
(
      CASE WHEN productfinish = 'Oak' then
      productstandardprice else 0 end
) AS Oak
FROM product_t")
```

Notice how you similarly repetitively use an aggregate function (in this example MAX) to display your columns of data. Also see that in this case we actually return the value of "productstandardprice", which is the price of the product in this case. We would need to return the price of the product because this is the value we are using within the MAX function.

Deliverable 1: Use table product_t, to create a pivot table that shows the total price of all the products for each product material category. Please screenshot and submit your answer.

3.2 Advanced Pivot Tables (Multiple Columns)

Now that we have covered how to create single column pivot tables, we can now move onto more advanced pivot tables with several attributes and columns of data. This becomes particularly helpful when you need to compare values against a certain attribute (ex. comparing sales over a monthly period).

Below is an example code of a multi-column pivot table. Write this into your R Studio to see the result. The pivot table will display the yearly number of sales for each Salesperson across 2009 and 2010:

```
dbGetQuery(conn, "SELECT SalespersonID AS SalesPerson,
            SUM(
                  CASE WHEN OrderDate LIKE '%09%' THEN 1 ELSE 0 END
            ) AS YR09,
            SUM(
                  CASE WHEN OrderDate LIKE '%10%' THEN 1 ELSE 0 END
            ) AS YR10
            FROM order_t WHERE SalespersonID NOT NULL GROUP BY
            SalespersonID")
```

Here, we pivot the years 2009 and 2010 into new columns. There are a few key differences to note with this code.

Observe that we added a new attribute, SalespersonID, to create an additional column in our pivot table. We also slightly changed our `CASE WHEN` condition to specifically target order dates occurring in 2009 and 2010 by looking specifically for '09' and '10' as year values. Lastly, notice how we needed to use `GROUP BY` to group/sort our data according to the SalespersonID. These changes are necessary in order to add a column, filter the pivot table, and ultimately display data related to the two attributes.

Deliverable 2: Using what you have learned above, create a pivot table displaying the number of sales made by salespersons #2-6 specifically on March 11, 2010 (11/Mar/10). In this example, display each Salesperson using SalespersonID as a separate column. Your output should show the pivot table below:

Day	SalesPerson1	SalesPerson2	SalesPerson3	SalesPerson4	SalesPerson5	SalesPerson6
11-Mar-10	4	9	3	3	3	7

HINT: Having SalespersonID as a separate column will require you to use the AND comparison operator to evaluate your CASE WHEN statements. Send screenshots of your answer.

If you finish early, feel free to try to display more information in the pivot table by showing other days as well (ex. March 10, 2010 (10/Mar/10)).