

PP22

TREE TRAVERSAL

Why You Should Never Transpose in PROC SQL

By J. Whittaker-Dixon

If you look online for ‘how to transpose using PROC SQL’ you will be told - repeatedly - that it cannot be done. My solution came to 345 lines, achieving what PROC TRANSPOSE can do in 6; proving that you can do it - you just shouldn’t. However, tackling the challenge requires us to look deeper at tree traversal, used for finding combinations of variables for the transposed columns. Within SAS, basic tree traversal methods include nested and macro do loops, and as we get more advanced we can create efficient algorithms dealing with file searching, data-structuring and much more.

The Problem

Whilst training as a programmer I was told that you could do almost everything in PROC SQL, except find a median and transpose. Naturally, I started to try. Median is quite simple, however transposing took quite a bit more work, eventually leading me to 345 lines of code and the writing of this paper. I do want to stress that *you should never transpose in PROC SQL* for any practical purposes; this is purely an exercise in what can be done, allowing us to increase our knowledge of SAS, SQL and programming in general.

Example

Consider the simple dataset `sales` from the library `sasuser`, shown on the right. We can transpose it in many different ways to create many different structures:

- One entry for each sales person
- One entry for each quantity
- One column for each month

Any of these can be achieved in PROC TRANSPOSE, and if approached correctly, PROC SQL.

The dataset with one entry for each quantity is shown on the right, named `work.long`. This is a *wide-to-long transpose*. On the other hand, a dataset with one entry for each sales person would be a *long-to-wide transpose*. A dataset with one column for each month requires both.

VIEWTABLE: Sasuser.Sales				
	LastName	Month	Residential	Commercial
1	SMITH	JAN	140097.98	440356.7
2	DAVIS	JAN	385873	178234
3	JOHNSON	JAN	98654.32	339485
4	SMITH	FEB	225983.09	12250
5	DAVIS	FEB	88456.23	55564
6	JOHNSON	FEB	135837.34	32423
7	SMITH	MAR	0	584593
8	DAVIS	MAR	89342	173222
9	JOHNSON	MAR	398573.9	193567

VIEWTABLE: Work.Long				
	LastName	Month	SaleType	Var
1	DAVIS	FEB	Commercial	55564
2	DAVIS	FEB	Residential	88456.23
3	DAVIS	JAN	Commercial	178234
4	DAVIS	JAN	Residential	385873
5	DAVIS	MAR	Commercial	173222
6	DAVIS	MAR	Residential	89342
7	JOHNSON	FEB	Commercial	32423
8	JOHNSON	FEB	Residential	135837.34
9	JOHNSON	JAN	Commercial	339485
10	JOHNSON	JAN	Residential	98654.32
11	JOHNSON	MAR	Residential	193567
12	JOHNSON	MAR	Residential	398573.9
13	SMITH	FEB	Commercial	12250
14	SMITH	FEB	Residential	225983.09
15	SMITH	JAN	Commercial	440356.7
16	SMITH	JAN	Residential	140097.98
17	SMITH	MAR	Commercial	584593
18	SMITH	MAR	Residential	0

The dataset `work.long` is simple to achieve using PROC SQL. If we have `&varn` variables (two variables in our case), we can make `&varn` copies of our dataset, rename each variable to the column name of our choice, and use the union join:

```
PROC SQL;  
  CREATE TABLE long AS  
    (SELECT /*by variables*/, "Residential" as SaleType,  
      Residential as Var FROM sasuser.sales)  
  UNION  
    (SELECT /*by variables*/, "Commercial" as SaleType,  
      Commercial as Var FROM sasuser.sales)  
QUIT;
```

An Introduction to Tree Traversal

Our main problem comes when we try a long-to-wide transpose: how do we know how many columns to create? In our `sasuser.sales` example, we have 2 time points and 3 parameters, so we will need 6 columns total. What if more than 2 variables are listed in the `ID` statement? If we have 5 `ID` variables, with 4, 2, 3, 2 and 5 distinct values each, we would need 240 columns. Is it possible to make this selection? To understand how we can solve this problem, we require tree traversal.

The Tree Structure

A tree begins with a root and branches off repeatedly to different *nodes* - often referred to as leaves. These nodes can be any kind of data, whether that be **folders**, **webpages**, or in our case, **columns**. At its most basic, tree traversal is the computer science concept of accessing every point on such a tree.

Our example dataset `sasuser.sales` can be thought of as a tree, with `Last-Name` representing a root, `Month` as the first layer of branches and `SaleType` as the nodes. Notice that we could assign these in any order, but the order we are picking reflects the nature of our data and the output we are aiming for: the sales person Davis (root) in January (node) had residential sales totalling 385873 (leaf).

In order to traverse our tree, we can use a method called *depth first traversal*. We do this by following our tree down to the end of the first leaf, then following around the perimeter. This will allow us to pass every node in the tree. We can choose to pick up the data either when we pass it on the left (pre-order traversal) or when we pass it on the right (post-order traversal). We then collect parts of a `SELECT` statement in PROC SQL. Similarly, we could be collecting parts of a file name to search an extensive library, or perhaps looking at possible outcomes for a patient’s treatment schedule.

The Solution

We have the dataset `work.long` which we recognize as a tree structure, with some `BY` variables, an `ID` variable and a `VAR` variable. With only one `ID` variable, we can store the distinct `ID` values in a list and use a macro `DO` loop to call each one:

```
PROC SQL;  
  CREATE TABLE wide AS  
    SELECT &byvars  
      %select_idvars  
    FROM long  
    GROUP BY &byvars;  
QUIT;  
%MACRO select_idvars;  
  %DO i = 1 %TO &N;  
    ,SUM (CASE WHEN /*ID=i*/  
      THEN var ELSE 0 END)  
    AS col&i  
  %END;  
%MEND;
```

When we move up to two `ID` variables, we can nest our `%DO` statement, looking for each combination of `idvar1` and `idvar2`. In a general case, we would need to move to a recursive macro in order to traverse the tree in PROC SQL. A recursive macro calls itself, which allows us to ‘nest’ any statements, including `DO` and `%DO` blocks.

Collecting our SELECT statement

With understanding of the tree structure behind the problem we can manipulate, traverse and transpose many problems in SAS. While transposing in PROC SQL is not a practical approach in itself, tackling the problem can give you a deeper understanding of your dataset and of transposition in general.

My general solution to the problem of transposing in SQL can be found through my corresponding paper on this topic, available online. I also recommend that before viewing this, an interested reader should make an attempt to recreate their own transpose in SQL. Additionally, please feel free to get in touch to discuss the problem by emailing me at: jessica.whittaker-dixon@quanticate.com.

Recommended Reading

- Storing Hierarchical Data in a Database; Gijs Van Tulder
Tree structures in databases
- Managing Hierarchical Data in MySQL; Mike Hillyer
Handling a tree structure in SQL
- Ted Conway; "It's a Bird, It's a Plane, It's SQL Transpose!"; Ted Conway
An method of tranposing in PROC SQL