PhUSE EU Connect 2018
Paper PP22

Tree Traversal

Why You Should Never Transpose in PROC SQL

J. Whittaker-Dixon, Quanticate, Manchester, UK

Abstract

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. In this paper I will be giving an overview of my method for transposing and use it to show how tree traversal can be understood and implemented in SAS.

Introduction

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[1], 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.

In this paper I will outline the problem in depth and discuss the outline of our transposition, break off to introduce tree traversal, and finally use this new found knowledge to build a solution. I will use the dataset `sasuser.sales` throughout, which is available in many SAS environments by default and will allow you to test the code snippets provided. If your SAS environment does not have this example dataset, it is available for download from [1].

The problem has been approached once before by Ted Conway [2], who suggested the code structure for basic long-to-wide transposition, which I will be building on in this paper; I would recommend reading his paper as well as my own. This problem does rely on a familiarly of what is capable using both PROC TRANSPOSE and PROC SQL. For more reading surrounding these procedures, I recommend both the SAS procedure documentation [3] or the Listen Data guides [4].

[1] This is a much smaller challenge which we will not be discussing in this paper, but if you are interested I would recommend looking into the `monotonic()` function.

The Problem

For the purposes of this paper, we will concentrate on a basic transpose, setting aside the PROC TRANSPOSE statements `IDLABEL` and `COPY` for the moment. Instead we will focus on what can be done using the `VAR` , `ID` and `BY` statements. We use the `VAR` statement to take a dataset from wide to long, while `ID` takes us from long to wide. The `BY` statement is used to group over one or more variables, and will be analogous to the `GROUP BY` statement in PROC SQL.

Example

Consider the sample dataset `sasuser.sales`, which has the structure:

`sasuser.sales:`

LastName	Month	Residential	Commercial
SMITH	JAN	140097.98	440356.7
DAVIS	JAN	385873	178234
JOHNSON	JAN	98654.32	339485
SMITH	FEB	225983.09	440356.7
⋮	⋮	⋮	⋮

Giving the residential and commercial sales figures in January, February and March for the sales persons Smith, Davis and Johnson. We would like to transpose the data into the following:

`work.output:`

LastName	SaleType	JAN	FEB	...
SMITH	Residential	140097.98	225983.09	
⋮	⋮	⋮	⋮	

This is both a "wide-to-long" and a "long-to-wide" transpose, first doubling the number of observations to move the variables `Residential` and `Commercial` into the single variable `SaleType`, then reducing the number of observations to create the variables `JAN`, `FEB` and `MAR`.

Wide-to-Long

The first stage, "wide-to-long", is simple to achieve using PROC SQL. If we have `&varn` variables (2 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 work.long AS
       (SELECT LastName, Month, "Residential" as SaleType,
          Residential as Var FROM sasuser.sales)
    UNION
       (SELECT LastName, Month, "Commercial" as SaleType,
          Commercial as Var FROM sasuser.sales)
QUIT;
```

This can easily be moved into a macro for any given number of `VAR` variables, and gives us the following long dataset:

`work.long:`

LastName	Month	SaleType	Var
SMITH	JAN	Residential	140097.98
SMITH	JAN	Commercial	440356.7
⋮	⋮	⋮	⋮

We can then use the dataset `work.long` for the remaining "long-to-wide" transposition. Notice that if we were to do a transpose which only required a "long-to-wide" step, the code above would just rename the only `VAR` variable.

Long-to-Wide

Difficulty arises when we move on to the "long-to-wide" transpose, creating new columns for the listed `ID` variables and grouping by `BY` variables. In principle, we would like the statement to look as follows:

```
PROC SQL;
    CREATE TABLE work.wide AS
       SELECT &byvars, col1, col2, col3, col4
       FROM work.long
       GROUP BY &byvars
QUIT;
```

where `col1`, `col2`, `col3`, `col4` are columns containing the transposing variable for the correct combination of `ID` variables. Above we have used `&byvars` to store our list of `BY` variables. In a similar way, we will use the macro variables `&idvar1`, `&idvar2`... to store each `ID` variable, and `&idvar1_1`, `&idvar1_2`... to store each possible value of `&idvar1`, and so on. As suggested in [2] we can use `MAX`, `SUM` or any other data-type appropriate summary function to make this selection:

```
PROC SQL;
    CREATE TABLE work.wide AS
       SELECT &byvars,
              MAX (CASE WHEN &idvar1=&idvar1_1 and &idvar2=&idvar2_1
                   THEN &var END) as col1,
              MAX (CASE WHEN &idvar1=&idvar1_1 and &idvar2=&idvar2_2
                   THEN &var END) as col2,
              MAX (CASE WHEN &idvar1=&idvar1_2 and &idvar2=&idvar2_1
                   THEN &va END) as col3,
              MAX (CASE WHEN &idvar1=&idvar1_1 and &idvar2=&idvar2_2
                   THEN &var END) as col4
       FROM work.long
       GROUP BY &byvars;
QUIT;
```

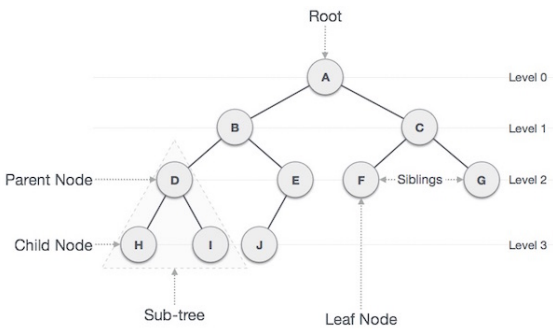
By listing all the `BY` variables in the `GROUP BY` statement, we create one observation for every combination of BY variables. So in our example, where `&byvars=LastName`, we have one observation per `LastName`. Each `CASE` then points us to a specific combination of `ID` variables; in our example these are a combination of `Month` and `SaleType`. Overall, our `MAX` function will always be taking the max of one observation, in place of each combination of `LastName`, `Month` and `SaleType`.

But how do we know how many columns to create? In our `sasuser.sales` example, we have 1 `ID` variable with 3 distinct values, so we require 3 new columns. What if 2 variables were given in the `ID` statement, or more? 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.

Tree Traversal

In order to understand tree traversal, we first need to understand a tree. A tree begins with a *root* and branches off repeatedly to different data points, called *nodes*, until it reaches the final data points, known as *leaves*. A diagram from Tutorials Point [5] is shown on the right and show a selection of basic terms, including *levels*, *parents* and *sub-trees*. These nodes can be any kind of data, whether that be folders, webpages, or in our case variables.

Figure 1: A Basic Tree Structure [5]

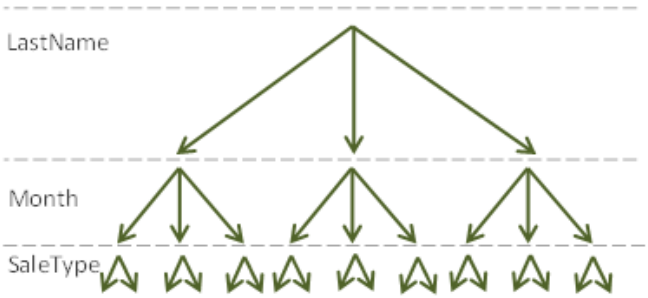


Example

Our example dataset `sasuser.sales` can be thought of as a tree, `LastName` representing the root, `Month` as the first layer of branches and `SaleType` as the leaves. Notice that we could assign these in a different order, but the order we pick should reflect the context of our data and the output we are aiming for: the sales person Davis (root) in January (node) had residential sales totalling £385873 (leaf). Although we have multiple `LastName` values, we are grouping them together using the `MAX - GROUP BY` statements and can be considered together as one root.

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	Commercial	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



In our example we have 3 levels, with 3 branches at the first and second level and 2 branches at the third level. In general, we will be able to find out this information and place it in macro variables, however our solution should be able to adjust to any number of levels and branches.

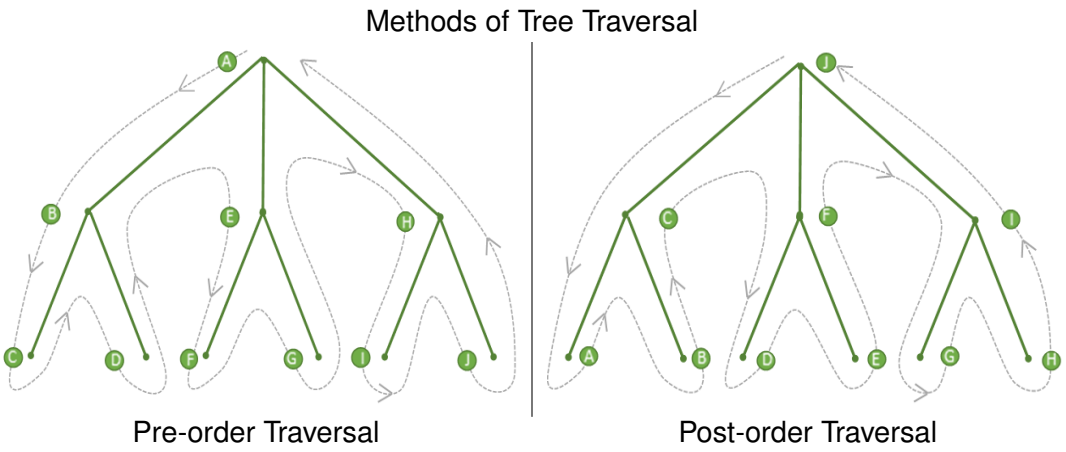
We can see that all our *sibling* nodes have the same number of *child* nodes. This is not necessarily always true, as we may by missing records for one sales person in one month, for example. In a general solution one could add a temporary empty observation or treat the missing branch as a "ghost" branch, however for brevity we will assume that we always have a complete set of records moving forward. In other scenarios, such as a file path search, for example, uneven sibling nodes would be expected regularly and would require us to rethink our traversal method, which we will discuss later in more detail.

Finally, in our example we can see 3 main *subtrees*, belonging to each of the `LastName` values. As sibling nodes have the same number of child nodes in our trees, each of these subtrees will be identical. We also have 9 smaller subtrees in each combination of `Month` and `LastName`. These are very small, just a root with two leaves, but they are still trees in structure. We can even consider the leaves as 18 distinct and identical sub-trees, each with one root node that is also its only leaf node.

Depth First

In order to traverse our tree, we can use many methods, including *breadth first* and *depth first* traversal. Given that we need to get to the *leaves* of the tree quickly, we will focus on *depth first* traversal. We do this by following our tree down to the end of the first leaf, then following around the perimeter of the tree. This will allow us to pass every node. We can choose to select the data either as soon as we visit the node, before moving on to its child nodes (pre-order traversal) or on the way back afterward (post-order traversal)[2]. Both methods are illustrated below:

[2] a third method, in-order traversal, is also an option but will not be covered in this paper. For more information see [4].



We can apply either of these methods in SAS, as we will see in the next section. While both methods will successfully list all nodes, the order of the output will change. We are collecting parts of a `SELECT` statement in PROC SQL for variables, so this will affect the order of the variables we create. 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 now have the information we need to implement our transposition in SAS. We have the dataset `work.long` which we recognize as a tree structure, with some `BY` variables, `ID` variables and one `VAR` variable ready for transposition. We will store the number of `ID` variables in `&id_N`, and the number of values for each `ID` variable in `&idvar1_N`, `&idvar2_N` and so on.

Starting in the simplest case, let `&id_N=1`. With only one `ID` variable, we can store the distinct `ID` values in a list and use a macro `%DO` loop to call each node:

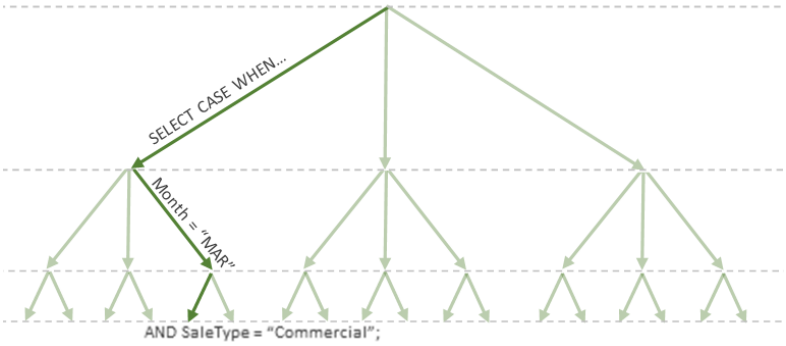
```
PROC SQL;
    CREATE TABLE work.wide AS
       SELECT &byvars
              %select_idvars
       FROM work.long
       GROUP BY &byvars;
QUIT;
```

```
%MACRO select_idvars;
    %DO i = 1 %TO &idvar1_N;
       ,MAX (CASE WHEN &idvar1=&&&idvar1_&i
                  THEN Var END)
        AS col&i
    %END;
%MEND;
```

When we increase to `&id_N=2`, we can nest our `%DO` statement, looking for each combination of `idvar1` and `idvar2`.

```
%MACRO select_idvars;
    %LET m = 0;
    %DO i1 = 1 %TO &idvar1_N;
       %DO i2 = 1 %TO &idvar2_N;
          %LET m = %EVAL(&m+1);
          ,SUM (CASE WHEN &idvar1=&idvar1_&i1
                     AND &idvar2=&idvar2_&i2
                     THEN var ELSE 0 END) AS col&m
       %END;
    %END;
%MEND;
```

We can continue using this method for any fixed number of `ID` variables, and we can see the tree structure playing out as we do it, each nested `%DO` is a new layer of our tree diagram. This is a simple tree traversal, and applying it to our `sasuser.sales` example we can see how it follows the diagram:

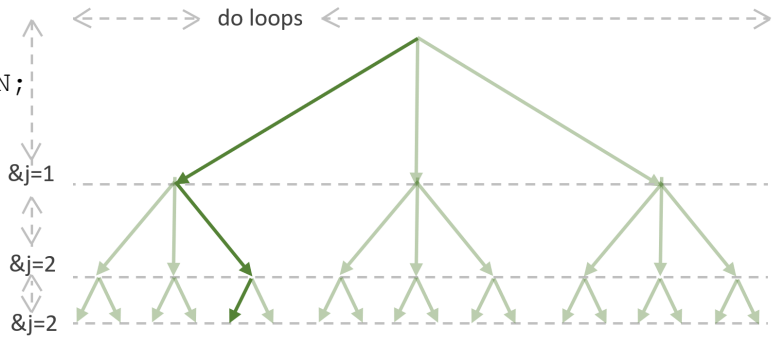


VIEWTABLE: Work.Long

	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	Commercial	193567
12	JOHNSON	MAR	Residential	398573.9

If we were to use a `DATA` step at this stage, we could utilize the following form of nested do loop statements. This method does rely on the fact that all sibling nodes have the same number of child nodes, which we established was true for our particular problem. By nesting a `DO` loop within a `%DO` loop, we can control how many levels we need and the size of those levels, without needing to type out some unknown number of nested loops. It is also a helpful advancement toward a complete PROC SQL solution, as we can visualize this code on a tree structure diagram as shown below.

```
DATA _NULL_;
    %DO j = 1 %TO &id_N;
       DO i&j = 1 %TO &idvar&j._N;
    %END;
          /*select needed rows*/
    %DO j = 1 %TO &id_N;
       END;
    %END;
RUN;
```



Recursion

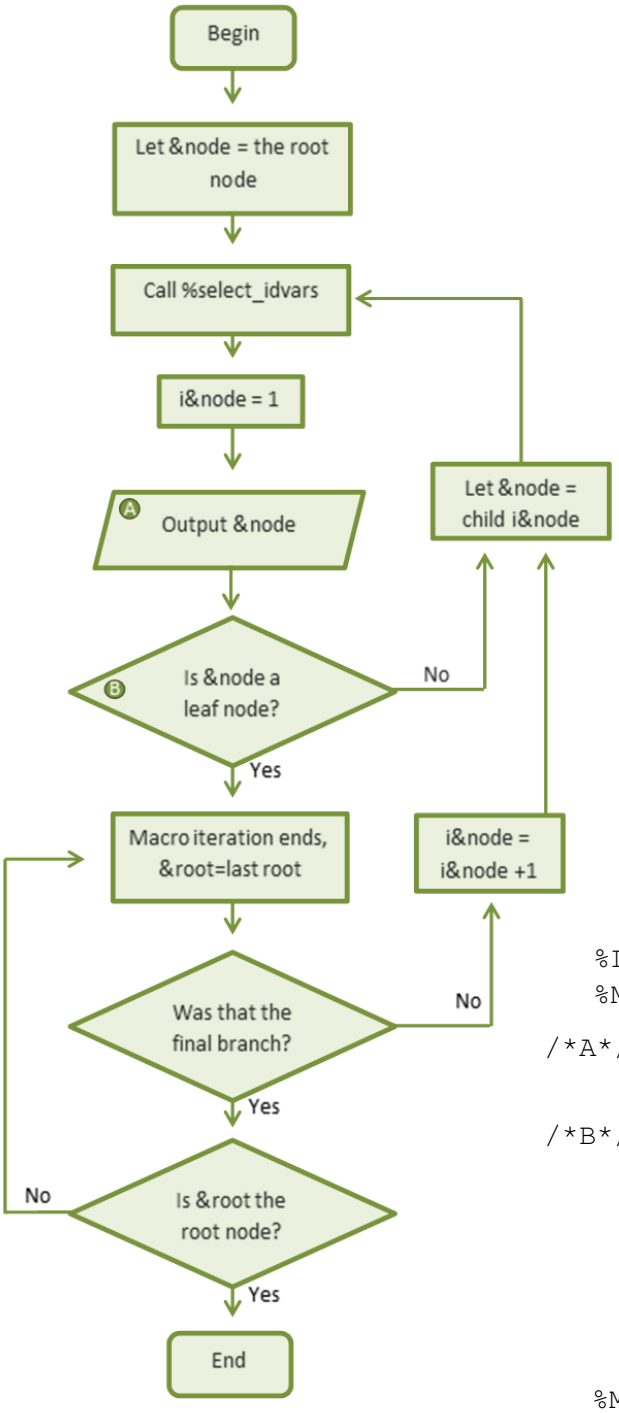
Unfortunately we cannot use this method with `%DO` loops only, so in order to create a PROC SQL only solution we will need to add *recursion* to the `%select_idvars` macro we created earlier. Recursion is where we allow a macro to call itself, creating more complex loops. Similar to the `DO WHILE` and `DO UNTIL` statements, it runs a high risk of causing an infinite loop in SAS, which we would like to avoid. If recursion is used in code, it should have a "safety variable" to minimize issues, along with usual defensive programming techniques, and be commented clearly so future edits do not cause issues.

```
%MACRO select_idvars(n=1);
    %*macro function*;

    /*this error exists to make sure any future code edits do not cause loops*/
    %IF &n > 999 %THEN %DO;
       %PUT %STR(ER)ROR: You are trying to produce far too many columns;
          %END;
    %ELSE %select_idvars(n = %EVAL(&n+1));
%MEND;
```

In the above, the macro variable `n` is only used to count the number of calls to `%select_idvars`, more than 999 calls is considered an error, and the macro will print an error and abort.

For each level of our nested `%DO` statement, we can now call the SAS program into a new iteration of the macro. Ensuring that we have a clause to stop the macro, we can use this method to build loops in any form we like. In the body of our macro, we can use the following algorithm to look through our database: if the data passed is a branch of a tree then call the macro again for its subtree, if it is a leaf then output the value.



When our macro `%select_idvars` is called, we set macro variable `&node` to the root node at level 0 of our tree dataset. The macro will then set a counter associated with the node, `i&node=1`, which it will use to count through the branches. Once it has output the portion of the `SELECT` statement, it checks if `&node` is a leaf node. If it isn't, we let `&node` equal `&i`th child node, and start a new iteration of the macro. If it is a leaf node, the macro iteration ends, returning `&node` to the last node.

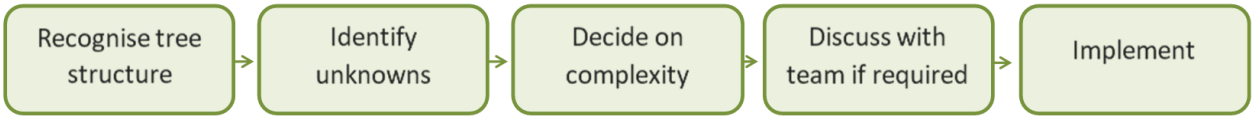
We can see that this method is *depth first*; the algorithm will always start by calling `%select_idvars` with child nodes until it reaches the first leaf. It is also a *pre-order* method as we are outputting `&node` when we first reach it, before we continue through the tree. We can easily adapt the flow chart on the left to be *post-order*, by moving points A and B in the diagram on the left or in the code below. In the simplified example code below, this is the same as moving the output below the macro `DO` loop.

```
%LET root = 0;
%MACRO select_idvars(n=1);
/*A*/ /*output &node*/
    %DO i&node = 1 %TO &&total_&node;
/*B*/ %IF &&level_&node < &level_max %THEN %DO;
        %LET node = child_&node;
        /*error warning*/
        %select_idvars(n = %EVAL(&n+1));
    %END;
%END;
%MEND select_idvars;
```

Of course this flowchart still leaves many options open to the programmer, in particular how to determine if a node is a leaf node or a final branch. In the example code above, I have assumed we have created macro variables storing this information earlier in the macro, which can be achieved with metadata databases (such as the `dictionary` library [6]) and PROC SQL summary functions earlier in the macro.

Other Applications

An understanding of tree traversal opens up many new avenues for solving complex problems, and when applied correctly can save huge amounts of time and energy for both you and for your SAS environment. The first step is to notice the tree within your problem. Once you have identified your tree, with defined root, branches and leaves, you can begin to consider the type of traversal required by considering the unknowns in your problem.



When recursion is involved in your solution, I recommend informing any relevant personnel involved with the project, including the error message suggested on page 5, and commenting in detail to explain your decision and method. When used properly, it should minimize the resources required significantly.

Conclusion

At the beginning of this paper we set out to transpose a dataset using only PROC SQL. While this problem may not be directly applicable to our day to day SAS use, the knowledge gained is a very powerful tool. Tree structures, traversal and recursion are all advanced programming techniques which can be used to tackle a wide range of problems. My full solution is attached to this paper as an appendix. 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, request hints, or suggest additions to my own solution.

References

[1] SAS example datasets
`https://support.sas.com/learn/statlibrary/`

[2] Ted Conway; "It's a Bird, It's a Plane, It's SQL Transpose!"
`http://www2.sas.com/proceedings/forum2008/089-2008.pdf`

[3] SAS Documentation Pages
`https://support.sas.com/documentation/`

[4] Listen Data SAS Guides
`https://www.listendata.com/p/sas-tutorials.html`

[5] A Basic Tree Structure Diagram
`https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm`

[6] Information about the PROC SQL `dictionary` libraries.
`https://v8doc.sas.com/sashtml/proc/zsqldict.htm`

Acknowledgements

Thank you to Will Greenway for reviewing this paper, and for presenting the corresponding poster at PhUSE Frankfurt 2018.

Recommended Reading

As well as the papers I have references directly, I recommend the following reading:

- Storing Hierarchical Data in a Database; Gijs Van Tulder
  `https://www.sitepoint.com/hierarchical-data-database/`
- Managing Hierarchical Data in MySQL; Mike Hillyer
  `http://mikehillyer.com/articles/managing-hierarchical-data-in-mysql/`

Contact Information

Jessica Whittaker-Dixon
Quanticate
Beech Lane
Wilmslow
SK9 5ER
+44 1625 544 841
jessica.whittaker-dixon@quanticate.com
https://www.linkedin.com/in/jessica-whittaker-dixon

Appendix

```
/**macro for transposing with only sql**/
/**works just like proc transpose**/

***parameters not avalible***;
    *idlabel;
    *copy;
*it is also assumed that the contents of id, prefix, suffix and delim are variable-permitted
    characters only, distinct from pre-existing columns;
*all other parameters work exactly as proc transpose;
*data is the only required variable;

%macro sqltranspose(data=,out=,by=,var=,id=,delim=, let=NO,
    name=_NAME_,label=_LABEL_,prefix=,suffix=,tidyup=YES);

**********checking parameters are all present*************;

    %if &data = %str() %then %do;
        %put %str(ER)ROR: 'in' is a required parameter;
        %goto exit;
    %end;

    *if &out not specified, set equal to data1 (or lowest avalible);
    %if &out = %str() %then %do;
        %let workdata=' ';
        proc sql noprint;
            select memname into :workdata separated by ' '
            from dictionary.tables
            where libname = 'WORK';
        quit;
        %let i = 1;
        %do %while (&out=%str());
            %if %sysfunc(index(&workdata,DATA&i)) = 0 %then %let out = data&i;
            %let i = %eval(&i+1);
        %end;
    %end;

    %if &name = %str() %then %let name = _NAME_;
    %if &label = %str() %then %let name = _LABEL_;

    *&ds is the dataset name, &lb is the library name;
    *library assumed to by WORK if no library given;
    %if %index(&data,.) %then %do;
        %let ds = %upcase(%scan(&data,2,.));
        %let lb = %upcase(%scan(&data,1,.));
    %end;
    %else %do;
        %let ds = %upcase(&data);
        %let lb = WORK;
    %end;

    *if ID statement is used, we will need custom prefixes;
    %if &id = %str() %then %let idyn = 0;
    %else %let idyn = 1;

    *if prefix statement not used, need to use COL;
    %if &prefix = %str() and &idyn=0 %then %let p = COL;
    %else %let p = &prefix;

```

```
57      *if delim statement used, need to enter delimeter;
58      %if &delim = %str() %then %let delimyn=0;
59      %else %let delimyn = 1;
60
61  **********storing macro variables**************;
62      proc sql noprint;
63
64          *get all columns of &data, plus the data types and labels;
65          create table columns as
66              select distinct name, type, label
67              from dictionary.columns
68              where libname="&lb" and memname="&ds";
69
70      quit;
71
72      *BY VARIABLES;
73      %sqlsep(str=&by,name=by,table=_by);
74
75      proc sql noprint;
76
77          *check that all &by variables are columns;
78          select upcase(a.name) from _by as a except select upcase(name) from columns;
79
80          %if &sqlobs>0 %then %do; %put %str(ER)ROR: Variable(s) &by not found; quit; %goto
                exit; %end;
81
82      quit;
83
84      *VAR VARIABLES;
85      %sqlsep(str=&var, name=var,table=_var);
86
87      proc sql noprint;
88
89          *if no vars specified, take all numeric columns not found in _by;
90          %if &varn=0 %then %do;
91
92              create table _var as
93                  select upcase(name) as name
94                  from columns
95                  where type='num'
96                  except select upcase(name) from _by;
97
98              select name into :var1-
99              from _var;
100
101             %let varn=&sqlobs;
102
103         %end;
104         %else %do;
105
106             *check that all &var variables are columns;
107             select upcase(a.name) from _var as a except (select upcase(name) from columns
                    union select upcase(name) from _by);
108
109             %if &sqlobs>0 %then %do; %put %str(ER)ROR: Variable(s) &var not found; quit;
                    %goto exit; %end;
110
111         %end;
112
113         *assign macro variables to all variable labels;
114         select label into :varlabel1- from columns as a, _var as b where
```

```
            upcase(a.name)=upcase(b.name);
115
116         *check if we have any labels;
117         %let vlabelyn=0;
118         %do i = 1 %to &varn;
119             %if &&varlabel&i ne %str() %then %let vlabelyn=1;
120         %end;
121
122     quit;
123
124     *ID VARIABLES;
125     %sqlsep(str=&id, name=id, table=_id);
126
127     proc sql noprint;
128
129         *if no ids specified, take all numeric columns not found in _by;
130         %if &idn=0 %then %do;
131
132             create table _id as
133                 select upcase(name) as name
134                 from columns
135                 where type='char'
136                 except select upcase(name) from _by;
137
138             select name into :id1-
139             from _id;
140
141             %let idn=&sqlobs;
142
143         %end;
144         %else %do;
145
146             *check that all &id variables are columns;
147             select upcase(a.name) from _id as a except (select upcase(name) from columns
                    union select upcase(name) from _by);
148
149             %if &sqlobs>0 %then %do; %put %str(ER)ROR: Variable(s) &id not found; quit; %goto
                    exit; %end;
150
151             *if we are not allowing duplicate id values, we will group the &data by all &by
                    and &id values;
152             *any observation appearing more than once after this grouping is a duplicate;
153             %if &let=NO %then %do;
154
155                 select count(*) into :dups1- from &data group by
156                     &id1 %do i = 2 %to &idn; ,&&id&i %end;
157                     %if &byn>0 %then %do; ,&by1 %do i = 2 %to &byn; ,&&by&i %end; %end;;
158
159                 %do i = 1 %to &sqlobs;
160                     %if &&dups&i>1 %then %do;
161                         %put %str(ER)ROR: Duplicate ID values found;
162                         quit;
163                         %goto exit;
164                     %end;
165                 %end;
166
167             %end;
168
169         %end;
170
171         *if we have no id values, we still want to make a single column;
```

```
172         %if &idn = 0 %then %let idn = 1;
173
174     quit;
175
176     *COLS VARIABLES;
177         *cols are all observations appearing in id variables;
178         *they are in the form &colAxB where A is the id number and B is the index;
179         *&col1n is the number of cols in id1, and so on;
180         *&coln is the product of all colNn, the total number of columns in the transpose;
181
182     proc sql noprint;
183         %let coln=1;
184         %do i = 1 %to &idn;
185
186             select distinct &&id&i
187                 into :col&i.x1 -
188                 from &data;
189
190
191             %let col&i.n = &sqlobs;
192             %let coln = %eval(&&col&i.n*&coln);
193         %end;
194
195     quit;
196
197 ********wide-to-long: creating a longer dataset with all vars seperated************;
198     proc sql;
199         *create &varn temporary versions of &data, each with one var named 'col' and all
                other var columns dropped;
200         *also create the name and label varible for &out;
201         %do i = 1 %to &varn;
202
203             *create a table with everything and duplicate copy of one &&var&i, renamed;
204             create table temp&i as
205                 select *, "&&var&i" as name,"&&varlabel&i" as &label, &&var&i as col
206                 from &data;
207
208             *drop all &vars from the table, leaving only non-var vriables and the renamed
                    copy of &&var&i;
209             alter table temp&i
210                 drop column &var1 %do j = 2 %to &varn;
211                                       ,&&var&j
212                                   %end; ;
213         %end;
214
215         *verically merge all temporary versions created above;
216         create table long as
217             select * from temp1
218             %do i = 2 %to &varn;
219                 union
220                 select * from temp&i
221             %end;;
222
223     quit;
224
225 **********long-to-wide*********;
226     proc sql;
227         create table &out as
228
229             /*select all by variables, the name column and the label column if applicable*/
230             select %do i = 1 %to &byn; &&by&i, %end; name as &name label='NAME OF FORMER
```

```
                VARIABLE' %if &vlabelyn %then , &label label='LABEL OF FORMER VARIABLE';
231
232             /*call %columnselect, part of the select statement for all transposed columns*/
233             %columnselect()
234             from long
235
236             /*group by all &by variables, along with name and label of applicable*/
237             group %do i = 1 %to &byn;
238                       &&by&i,
239                  %end; &name %if &vlabelyn %then , &label;;
240     quit;
241
242
243     %if %upcase(&tidyup)=YES %then %do;
244
245         proc sql;
246             drop table columns, long, _id, _by, _var;
247             %do i = 1 %to &varn;
248                 drop table temp&i;
249             %end;
250         quit;
251
252     %end;
253
254     %exit:
255 %mend sqltranspose;
256
257 *name takes the prefix of the macro variables to be produced;
258 *table takes the name of the table to be made - if left blank, no table made;
259 %macro sqlsep(str=,name=,dlm=' ', table=);
260
261     %global &name.1; %global &name.n;
262
263     %let &name.1 = %scan(&str,1,&dlm);
264     %let i = 1;
265
266     proc sql noprint;
267
268         create table &table(name char(200));
269
270         %do %while (&&&name&i^=%str());
271
272             insert into &table values("&&&name&i");
273
274             %let i = %eval(&i+1);
275             %global &name&i;
276             %let &name&i = %scan(&str,&i,&dlm);
277
278         %end;
279
280     quit;
281
282     %let &name.n = %eval(&i-1);
283 %mend sqlsep;
284
285 *macro called to select transposed columns;
286     *assumes it is run within %sqltranspose;
287     *this macro requires the following variables in order to run:
288         &idn - the number of ID columns
289         &idA - the name of ID column A (A ranges from 1 to &idn)
290         &colAn - the number of columns being created by &idA
```

```
291         &colAxB - the name of column B from ID column A (B ranges from 1 to &colAn);
292 %macro columnselect(n=1);
293
294     /*on the first run of columnselect, create temporary variables i&i*/
295     %if &n=1 %then %do;
296         %do i = 1 %to &idn;
297             %let i&i = 1;
298         %end;
299     %end;
300
301     /*set up prefixes for the column*/
302     %if &idyn %then %let prefix = &p&&&col1x&i1;
303     %else %let prefix = &p&n;
304
305     /*for a given column colAxB, select the correct observation*/
306     /*(for each columnselect we have a combination of &&i&c values, where &c corresponds to
            the id)*/
307     , sum (case when 1
308                 %do c = 1 %to &idn;
309
310                     /*add to the prefix is needed*/
311                     %if &idyn and &c>1 %then %do;
312                         %if &delimyn %then %let prefix = &prefix&delim;
313                         %let prefix = &prefix&&&&col&c.x&&i&c;
314                     %end;
315
316                     and &&id&c="&&&&col&c.x&&i&c" /*evaluate to colAxB for each combination of
                            A and B*/
317
318                 %end;
319
320            then col else 0 end) as &prefix&suffix
321
322     /*use recursive traversal over all &&i&c combinations within ranges*/
323
324     %do x = 1 %to &idn;
325
326         %if &&i&x < &&col&x.n %then %do;
327
328             %let i&x = %eval(&&i&x+1);
329
330             /*this error should never happen*/
331             /*it exists to make sure any code edits do not cause recusive loops*/
332             /*do not call columnselect without this error*/
333             %if &n > 999 %then %do;
334                 %put %str(ER)ROR: you are trying to produce far too many columns;
335                 %abort;
336             %end;
337
338             %columnselect(n = %eval(&n+1))
339
340         %end;
341
342         %let i&x = 1;
343
344     %end;
345 %mend columnselect;
346
347
348 /*an example call*/
349 %sqltranspose (data=sasuser.sales,
```

```
350             out=sales,
351             by=,
352             var=,
353             id=lastname month,
354             delim=,
355             let=yes,
356             name=_NAME_,
357             label=_LABEL_,
358             prefix=,
359             suffix=,
360             tidyup=YES);
```