Paper 1400-2017

More than a Report: Mapping the TABULATE Procedure as a Nested Data Object

Jason Phillips, The University of Alabama

ABSTRACT

The TABULATE procedure has long been a central workhorse of our organization's reporting processes, given that it offers a uniquely concise syntax for obtaining descriptive statistics on deeply grouped and nested categories within a data set. Given the diverse output capabilities of SAS®, it often then suffices to simply ship the procedure's completed output elsewhere via the Output Delivery System (ODS). Yet there remain cases in which we want to not only obtain a formatted result, but also to acquire the full nesting tree and logic by which the computations were made. In these cases, we want to treat the details of the Tabulate statements as data, not merely as presentation. I demonstrate how we have solved this problem by parsing our Tabulate statements into a nested tree structure in JSON that can be transferred and easily queried for deep values elsewhere beyond the SAS program. Along the way, this provides an excellent opportunity to walk through the nesting logic of the procedure's statements and explain how to think about the axes, groupings, and set computations that make it tick. The source code for our syntax parser is also available on GitHub for further use.

INTRODUCTION

PROC TABULATE is truly unique among the numerous statistical and reporting methods available within SAS, for it allows you to request a hierarchical and deeply nested partitioning of your data within a single step, in contrast to the majority of other procedures that are primarily linear in execution. It does so by way of a distinct syntax (in the TABLE statement) that can concisely express nearly any desired map of subdivisions and statistics to be explored within a single dataset. And, most importantly for the purposes of this paper, it produces a visual representation—in the form of a table nested across two axes—that inherently straddles the line between presentation and information.

To quickly illustrate the latter characteristic, examine in Figure 1 the visual output of a very simple TABULATE procedure and compare it to the accompanying results dataset that SAS produces when using the OUT= option:

Mean GPA by Status, Sex, and Year

		2014	2015	2016	2017
Full-time	Female	2.18	2.18	1.18	2.18
	Male	3.51	2.12	2.20	2.18
Part-time	Female	2.91	2.18	2.67	2.18
	Male	3.01	2.60	1.18	3.81

	status	sex	year	Type of Observation	Page for Observation	Table for Observation	gpa_Mean
1	Full-time	Female	2014	111	1	1	2.18
2	Full-time	Female	2015	111	1	1	2.18
3	Full-time	Female	2016	111	1	1	1.18
4	Full-time	Female	2017	111	1	1	2.18
5	Full-time	Male	2014	111	1	1	3.51
6	Full-time	Male	2015	111	1	1	2.1183333333
7	Full-time	Male	2016	111	1	1	2.195
8	Full-time	Male	2017	111	1	1	2.18
9	Part-time	Female	2014	111	1	1	2.91
10	Part-time	Female	2015	111	1	1	2.18
11	Part-time	Female	2016	111	1	1	2.6666666667
12	Part-time	Female	2017	111	1	1	2.18
13	Part-time	Male	2014	111	1	1	3.01
14	Part-time	Male	2015	111	1	1	2.595
15	Part-time	Male	2016	111	1	1	1.18
16	Part-time	Male	2017	111	1	1	3.81

Figure 1: Tabulate visual output (left) versus dataset output (right)

Each distinct summary statistic represented in the visual output can be located in the dataset; however, the two forms of output are not strictly isomorphic, for there is no consistent technique that can be applied to the dataset in order to reproduce the accompanying table's form. This limitation is not purely a matter of knowing which variables should be assigned to the vertical rather than the horizontal axis, for it touches upon the distinction between having all of the groupings and statistics available as isolated data points versus being able to trace the particular pathways or ordered sequences of subdivisions intended by the author of the report. And in the case of more complex uses of TABULATE featuring various

concatenated divisions of information at every level, the comparison between the visual output and the dataset is even more obscure.

By contrast, in the case of other major methods of output like the REPORT procedure, the dataset produced primarily differs from the visual output only in ways that more strictly belong to presentation, like fonts, styles, alignments, and other adjustments; the structural form of the information itself is very closely mirrored.

Why does this matter? In my work at a large university where we regularly produce reports in a wide variety of formats for numerous audiences (accreditors, federal agencies, etc.), there are frequently cases in which it is desirable to store or transmit our reporting in formats that can be easily reinterpreted within our other systems or platforms, particularly on the web. And while we make heavy use of the Output Delivery System for targeting multiple static presentations, it is not well-suited to the goal of making portions of a report dynamically reproducible from the ground up within another framework, for ODS results are primarily presentational in nature rather than strictly structured as data. Datasets, however, are a universal concept (a collection of records with uniform fields), and can be ported from SAS to nearly any platform with ease—except that output datasets do not, in the case of PROC TABULATE, contain sufficient information to reproduce the particular shape intended by the report author.

The challenge I have often faced, and ultimately the impetus for creating a solution and eventually this paper, was to answer the question: how can I transmit the results of a complex TABULATE procedure to another system in such a way that I neither settle for a list of isolated statistics, nor for a picture-like visual presentation, but instead insist on communicating both the data *and* the precise mapping of divisions intended by the report author?

In what follows, I will briefly explain the manner in which I conceptualized the logic of the TABULATE procedure as essentially a kind of network or graph query, so that I could use the latest tools from that latter domain to request and render accurate tables using only the output dataset and a syntax tree drawn from parsing the TABLE statement itself. Whether or not you leave with any particular use or desire for our specific solution, the approach to analyzing PROC TABULATE that follows should offer you a different and useful way of reasoning about the procedure.

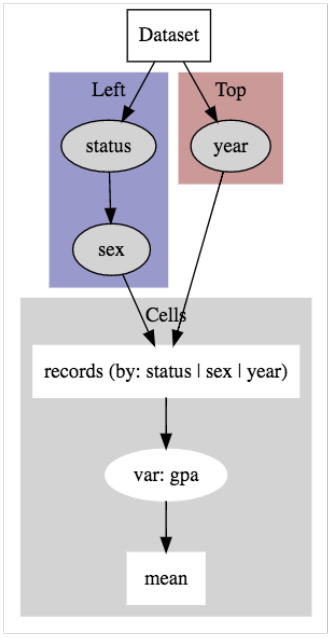
UNDERSTANDING THE TABLE STATEMENT AS PATHS

In order to make the TABLE statement intelligible as a kind of network query, examine this basic TABULATE procedure (which has all labels removed in order to simplify the resulting structure):

```
title "Mean GPA by Status, Sex, and Year";

proc tabulate data=studentgpa out=results;
    class status sex year;
    var gpa;
    table status='' * sex='',
            year='' * gpa='' * mean='';
run;.
```

Figure 2 uses a simple network tree to visually depict the logic of the code above, which specifies that the data should be traversed by two classes in succession on the left side (status and sex) and by one class on the top side of the table (year). The intersection of these two traversals defines the contents of the resulting table cells, each of which will represent the chosen variable (GPA) drawn from its intersection of the data, upon which the mean statistic is then to be calculated. This network illustration is purely a blueprint interpreting the logic written in the code sample above, prior to interacting with the dataset or performing any examination of its contents.



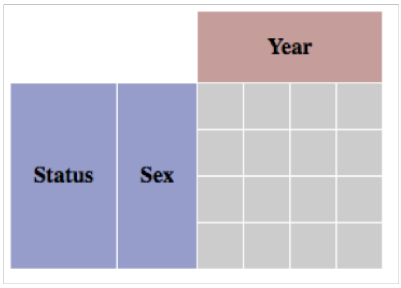


Figure 2: Simplified network graph (left) and blueprint table (right)

Note that this diagram places the variable (GPA) within the gray shaded box representing cells, rather than on top where the TABLE statement declares it. As a representation of what actually occurs in the logic of execution, this approach is somewhat more helpful, for it emphasizes that the top and left of the table primarily divide the dataset into nested subsets of records by classes, while the cells are properly the location of computed statistics on particular variables. However, if you rewrite the TABULATE procedure with a label for GPA, or with more than one analysis variable or statistic included, those labels do of course show up in your visual result as part of either the top or left axis as written in the TABLE statement, while the cells strictly contain merely the result.

Once you run the procedure, the classes expand into all their distinct values available in the dataset, and SAS renders the completed table. Figure 3 demonstrates the resulting shape of the generated table (right) and traces the full pathway traversed (left) to reach a particular data cell.

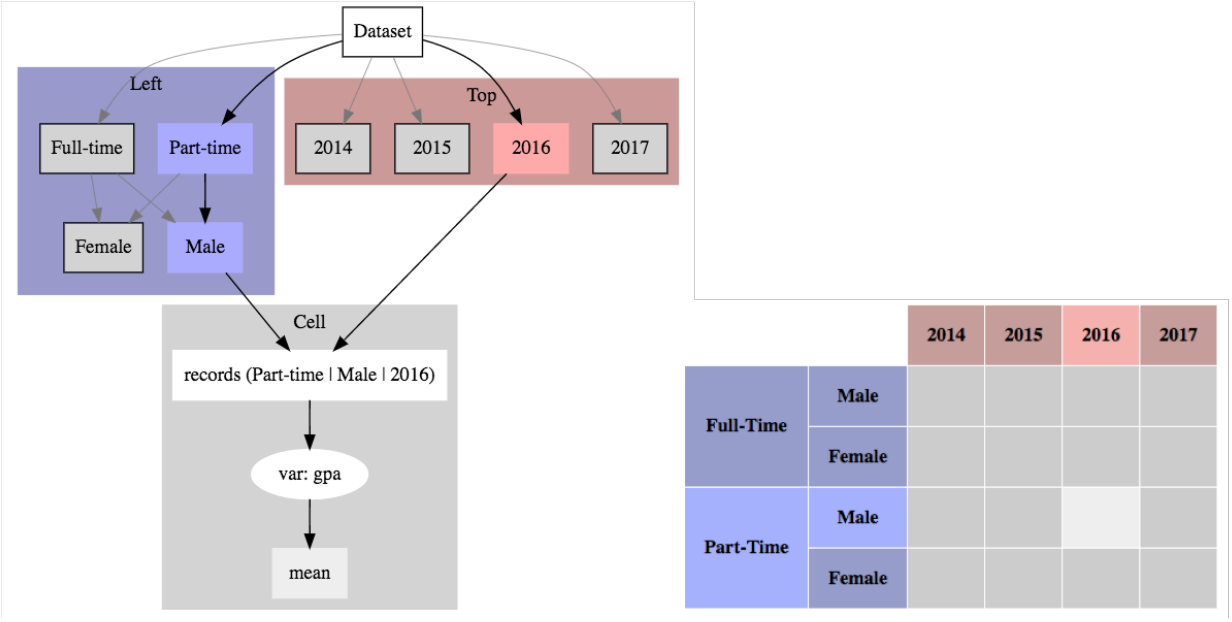


Figure 3: Realized path (left) to a particular cell (right)

The pathway highlighted determines the set of records that will be represented by the resulting cell, as a union of the narrowing conditions passed through during traversal of the left and top of the table. In this case, the highlighted cell represents those records having a **status of part-time**, a **sex of male**, and an **academic year of 2016**. From this resulting subset of the original dataset, the chosen analysis variable (GPA) is pulled out to form a single data series, after which the statistic chosen (mean) is calculated against that series.

FROM TABLE STATEMENT TO QUERY

The diagrams above illustrate a way to break down the tabulation process into (1) a network graph per axis, ultimately resolving into (2) an intersection of records per cell, from which (3) a particular analysis variable is extracted and (4) a statistic on that series is executed. The next task is to reshape the TABLE statement in order to create a graph-like query that can be transported to another framework. Revisit the code sample examined above, slightly expanded to show two statistics:

```
proc tabulate data=studentgpa out=results;
    class status sex year;
    var gpa;
    table status='' * sex='',
            year='' * gpa='' * (mean='' n='');
run;.
```

The TABLE statement uses two types of ordering syntax to represent a nested tree of information. The asterisk (*) indicates a descent into a subset of data beneath the current level (by expanding a class, or designating a variable and statistic at the deepest level), whereas items that are grouped together by spaces or within parentheses exist at the same level. By splitting these, we can easily convert the TABLE statement into a tree-like structure, placing the top and left parts of the statement two top-level paths:

```
Table:
  - Left
     -> Each(status)
       -> Each(sex)

  - Top
     -> Each(year)
```

```
        -> Variable(gpa)
          -> Statistic(mean)
          -> Statistic(n)
```

To call this resulting tree a “graph query” means that it describes a series of paths by which to traverse the data. It also means that the shape of the resulting data can be presented in a homologous form, by populating the levels above with all of the actual values found while narrowing the dataset along each pathway, so that every “edge” connection above (represented by the “->” arrow) may point to multiple realized nodes:

```
Table:
  - Left:
      Full-time
        Female
        Male
      Part-time
        Female
        Male

  - Top:
     2014
       GPA
          Mean
          n
     2015
       GPA
          Mean
          n
    ...
```

This simplified example avoids the full complexity of having many groupings and nested parentheses, yet it demonstrates the basic process of transforming a TABLE statement into a query tree (and then a realized data tree) by merely parsing its syntax and representing the result in a nested form. The full parser I built reads the syntax of a written TABULATE procedure and can convert a great deal of nested complexity into a clean tree like that above, aiming to cover most use cases we will encounter in practice.

EXECUTING A NETWORK QUERY

Before moving to the specific language framework I adopted for implementing the graph data structures shown above, it is helpful to restate, in abstract form, the general sequence by which this hierarchical query would be executed against a dataset:

1. Begin with the source dataset, and follow down the two axes (Left and Top)
2. For every edge (“->”) that connects a new Class variable, expand all the values of that class found in the current subset of data, and continue onto each of those as a new node containing a new subset of records matching that condition.
3. Expand to the deepest point of each axis, including analysis variables and statistics, and treat these resulting leaves as the two sides (left and top) of a matrix of cells.
4. For any given cell, take the intersection of the subsets of records represented in the leaf at its row and column, slice the chosen analysis variable of out of those records as a series, and compute the statistic indicated.

Revisit Figure 3 to see how closely this sequence matches the initial network diagrams sketched above.

OUR SOLUTION: GRAPHQL

It is my hope that the above analysis and translation of PROC TABULATE into a hierarchical graph query and the logic by which that query would be resolved into the final data and table shape proves useful as a learning exercise, even to those who do not have an interest in the specific solution I implemented. However, what follows is a brief summary of the realized layer of live queries and presentation I built in accordance with the specifications above, and that is offered online for use or experimentation (see References).

GRAPHQL AS A QUERY LANGUAGE

GraphQL is a language and specification created by Facebook for querying deep structures of nested graph data (Schrock, 2015). Since it is platform-agnostic and has been widely adopted in web applications over the past couple of years, I have been able to use this query language to tie together systems or make application requests for complex data across multiple programming languages, often in our case connecting servers running Python or Node.js. In short, since I needed a way to transform the abstract network trees detailed above into a specification that could be executed and transmitted consistently, GraphQL was a natural choice with broad language support and industry backing.

GraphQL's syntax models JSON, a data exchange format which has overtaken the web for nearly the past decade, and essentially mimics the shape of the JSON data it expects to receive in return. Transforming a tree similar to the one above into a GraphQL query, you would see obtain the (abridged) code shown in Figure 4 below.

```
query tabulate {
  table(dataset:"studentgpa") {
    top {
      classes(key:"Level", all:"Grand Total") {
        classes(key:"Sex") {
          leaf
        }
      }
    }
    left {
      classes(key:"Year") {
        variable(key:"GPA") {
          aggregation(method:"mean") {
            leaf
          }
          aggregation(method:"n") {
            leaf
          }
        }
      }
    }
    cells {
      value
    }
  }
}
```

Figure 4: GraphQL Syntax

The query tree shown above is transmitted from the client (web or mobile) application to the server, which then interprets it as a set of hierarchical paths and executes it against the dataset, returning the data in a shape that is well suited to rendering an accurately nested table.

USING THE RESULTS OF PROC TABULATE

By using the **OUT=** option in the TABULATE procedure, you can produce a dataset containing all of the unique summarized statistics; see the example output shown in Figure 1. While I have written our GraphQL interpreter to be able to run simple statistics against a dataset on its own, to use the dataset output by a PROC TABULATE procedure directly is a far better option for gluing together our ecosystem and taking advantage of the more complex computational powers of SAS while not sacrificing interoperability.

This means that we can achieve the workflow originally desired, which puts the control of the report's logic in the hands of our SAS analysts while making their work available to other systems:

1. A data analyst designs and writes the report using an ordinary TABULATE procedure, and may produce one or more ODS outputs for specific destinations.
2. Using the **OUT=** option, they also output their resulting dataset to a database table that is accessible to our web applications.
3. Their TABULATE statement is given to our web group and automatically parsed into a GraphQL query tree in the appropriate format.
4. Our web application stores the query, and can execute it on demand against the output dataset, rendering the resulting table.
5. If the data need to be refreshed regularly, the underlying dataset can simply be replaced by executing the SAS program again, while the query statement contained in the web application remains static.

OUR TABLE RENDERER

One advantage of this somewhat more complex application stack over the simplicity of something static like ODS is that the hierarchical data logic is entirely understood by the web application that renders the report to the client. This means that we can easily integrate interactivity into the table or its cells, since each cell is "live" in the sense that it knows systematically what information it represents. The screenshot in Figure 5 below shows one of our tables with a simple summary popup that appears upon touching any cell, illustrating the benefits of allowing a user to navigate a table this is responsive and live rather than statically rendered in advance.

		2016 Spring				2016 Fall				Grand Total			
		Grad		Undergrad		Grad		Undergrad		Grad		Undergrad	
		Mean Grade	Count	Mean Grade	Count	Mean Grade	Count	Mean Grade	Count	Mean Grade	Count	Mean Grade	Count
Female	N	.	0	3.11	1	1.91	1	2.97	3	1.91	1	3.00	4
	Y	3.33	1	3.51	1	.	0	.	0	3.33	1	3.51	1
Male	N	.	0	3.19	2	.	0	3.44	1	.	0	3.27	3
	Y	.	0	3.44	2	.	0			.	0	3.44	2

GPA | mean
Gender: M
Instate: N
Term: 2016 Fall
Level: U

Figure 5: Live rendered table with overlay

CONCLUSION

PROC TABULATE is an indispensible part of the SAS language, and exceeds any other framework on the market in its ability to succinctly express and then efficiently execute a deeply nested set of groupings and statistics on a dataset. By taking a fresh look at the underlying logic of the procedure and reinterpreting it as a kind of structured graph or network query, it is possible to take not only the isolated statistical results—and not only the static presentation—but also the entire nested tree of information it represents and integrate that structure with other systems. Our project has shown how to travel very far down this road, and whether you have an interest in the specific implementation or merely in the concepts, I believe this approach demonstrates the benefits of thinking systematically about what PROC TABULATE can do and how it essentially functions.

REFERENCES

Phillips, Jason. "Retabulate Project on GitHub." Available at: https://github.com/jasonphillips/retabulate

Schrock, Nick (2015, May 1). "GraphQL Introduction." Available at: https://facebook.github.io/react/blog/2015/05/01/graphql-introduction.html

ACKNOWLEDGMENTS

I would like to offer thanks to the Office of Institutional Research at The University of Alabama and in particular to its director Lorne Kuffel for indulging and encouraging side projects of this nature, in which the practical payoff is not always evident in the early stages.

RECOMMENDED READING

- https://en.wikipedia.org/wiki/Graph_theory
- http://graphql.org/
- https://github.com/chentsulin/awesome-graphql

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Jason Phillips
The University of Alabama
jphillips@ua.edu
https://github.com/jasonphillips

All data presented above is randomized sample data and does not reflect actual statistics on The University of Alabama.