Jonathan Sacramento
25 September 2015

On Transposing Data

Data Analysis in the UNIX Environment

A transpose (or "pivot") is a way of transforming data from one view to another. There are infinitely many ways of transforming data, and each attempts to get data in a form that's easier to view, analyse or process further. The definition of the verb "transpose" is to "*cause (two or more things) to exchange places*". So, how does this apply to data, and when might such a transformation be useful? To answer this, let's go through some quick examples…

Example 1 - Simple row-to-column transpose

When reading about transposing data, some will (naively) think solely of exchanging rows for columns (and vice versa). This example illustrates this transformation. The data shows the sales volume per quarter for a set of regions.

Raw table 1 - Sales volume data for a given year

Quarter	Europe	Asia	US
Q1	2	5	3
Q2	3	4	1
Q3	3	5	2
Q4	4	6	3

Transposed table 1

Quarter	Q1	Q2	Q3	Q4
Europe	2	3	3	4
Asia	5	4	5	6
US	3	1	2	3

Visually, this transpose "rotates" the data over an invisible diagonal line running through the table. No information is lost through aggregation - the values simply change position, giving a different view of the data. This might be useful in making data easier to read or chart.

Example 2 - Transposing over a "group" field

Now consider a billing table that captures customer transactions in a given month, as shown below (Raw table 2). Say we want to analyse the total amount for each revenue group, and that we want this visible in one row (we discuss why later on) - in other words, we want an output table with the different revenue groups as column headers, and a row of total values for each.

Raw table 2 - Billing data for a given month

Customer ID	Revenue Group	Amount
1	rev_A	2
1	rev_B	3
2	rev_C	6
3	rev_B	7
3	rev_B	2
3	rev_C	9
4	rev_A	8
4	rev_A	3

Transposed Table 2

rev_A	rev_B	rev_C
13	12	15

This can be achieved by transposing the table over a given *group* field ("Revenue Group"). The process entails aggregating a *numeric* field ("Amount") for each distinct group. Notice that we're still performing a form of 'row-to-column' exchange on a subset of the table (the "Revenue Group" field values become column headers, and the "Amount" field is aggregated into a single row).

It isn't hard to see the value of this transpose operation - useful information can be gleaned from raw data with a single transformation. This can be used to verify totals, run integrity checks on raw data, view the evolution of variables etc.

Example 3 - Transposing over a "group" field, preserving an "id" field

Taking the previous example further, let's say that we want to view each customer's total revenue breakdown for the month - in other words, we want a 'one-row-per-customer' view broken-down into the different revenue groups. Using the same data as in Example 2, the resulting table would look like this:

Transposed Table 3

Customer ID	rev_A	rev_B	rev_C
1	2	3	0
2	0	0	6
3	0	9	9
4	11	0	0

We arrive at this result by transposing the table over a given *group* field ("Revenue Group"), whilst preserving an *id* field ("Customer ID"). The process entails aggregating *numeric* field ("Amount") for each distinct group and id. This transformation naturally keeps distinct values of the id field in a separate row.

Apart from giving a clear view to the human eye of the value of each group for every id (everything's in one row), there is another reason why such a view is handy. Each row can be viewed as an array of values that can be iterated over and processed further. To illustrate this, imagine that instead of different revenue groups ('rev_A', 'rev_B',…), we had transposed the monthly snapshots of a single revenue group ('rev_A_month_1', 'rev_A_month_2',…), resulting in the following table:

Transposed Table 4

Customer ID	rev_A_month_1	rev_A_month_2	rev_A_month_3	…
1	2	3	0	
2	0	0	7	
3	1	5	4	
4	3	1	0	

Using this view of the data, we could iterate over all rows (i.e. all customers) - treating each as *an array of values* - and run our analyses on each array, in turn. For example, we could run an analysis that tries to predict customer churn based on monthly spend activity.

We now have a bit of more clarity on how data can be transposed - but how can we perform this transformation on our own data, and what tools are available to us? If you're like me, you want to be able to do this via a shell, processing delimited text files through a carefully crafted pipe of standard UNIX programs (or even better, a single program!). But in order to appreciate the beauty of data analysis in the UNIX environment, let's examine the alternatives.

SQL **using postgresql 9.4*

SQL is the de facto language for data analysis on databases. However, carrying-out a transpose in SQL can force us to write unclean code that is hard to maintain. There is also the issue of having to import the raw data into a database before following this approach. If we wanted to transpose the table from Example 2 (Raw table 2) by *group* and *id*, we could write the following code:

```
SELECT
   customer_id
   ,sum(rev_A) as rev_A
   ,sum(rev_B) as rev_B
   ,sum(rev_C) as rev_C
FROM (
   SELECT
      customer_id
      ,CASE WHEN revenue_group in ('rev_A') THEN amount END as rev_A
      ,CASE WHEN revenue_group in ('rev_B') THEN amount END as rev_B
      ,CASE WHEN revenue_group in ('rev_C') THEN amount END as rev_C
   FROM
       raw
) t1
GROUP BY
   customer_id
```

But this code is not exactly an elegant solution to our problem. Imagine that next month the business decides to add a new revenue type '**rev_D**'. The above code would not capture this

new group in the output table - in other words, changes in structure of the source data will require a modification of the code; If we were running this code as part of a monthly summary report, this solution just wouldn't do!

Postgresql provides another means of transposing the data the way we want via the `CROSSTAB` function (other database systems like Microsoft's SQL Server, or Oracle have a `PIVOT` function that achieves the same result):

```
SELECT
   *
FROM
   CROSSTAB (
      'SELECT customer_id, revenue_group ,sum(amount) as amount
       FROM raw
       GROUP BY customer_id, revenue_group
       ORDER BY customer_id'
     ,'SELECT DISTINCT revenue_groups FROM raw'
   )
   AS (
      customer_id int
      ,rev_A real
      ,rev_B real
      ,rev_C real
   );
```

However, this still suffers from the previous problem, as we need to modify the code in response to changes in the structure of the data; specifically, we are required to define the columns and data types for the output table (see the '`AS`' definition in the '`FROM`' clause). We could of course generate dynamic SQL to remedy this - that is, executing a code-generated SQL query that adds the distinct groups via a *cursor*. We would, however, have to modify the code (or write a separate query) if we wanted to transpose over the group field (without the id) instead.

SAS

SAS, with it's archaic 70's syntax, does actually have a transpose function that gets us close to an ideal solution… the drawbacks are:

- SAS does require us (like databases) to import data before we process it, which means having to define column data types and sizes. This also forces us to redefine these should our data change or exceed these limits later on;
- SAS's transpose function expects the data in a form that has no duplicate *group* values for each *id*; that is, the data would have to be aggregated before transposing. Taking the data from Example 2, we would have to go from this:

Sample data from Example 2

Customer ID	Revenue Group	Amount
1	rev_A	2
1	rev_B	3
2	rev_C	6
3	rev_B	7
3	rev_B	2
3	rev_C	9
4	rev_A	8
4	rev_A	3

to this…

Aggregation required before SAS transpose

Customer ID	Revenue Group	Amount
1	rev_A	2
1	rev_B	3
2	rev_C	6
3	rev_B	9
3	rev_C	9
4	rev_A	11

Once the data is in a SAS-acceptable form, we can go ahead and transpose as follows:

Simple transpose ('rows-to-columns')

```
proc transpose data=raw out=output_simple;
run;
```

Transpose over group **requires prior aggregation*

```
proc transpose data=raw out=output_group;
   id revenue_group;
   var amount;
run;
```

Transpose over group, preserving id **requires prior aggregation*

```
proc transpose data=raw out=output_group_id;
   by customer_id;
   id revenue_group;
   var amount;
run;
```

Alas, SAS is proprietary software (with extremely prohibitive license costs) that ultimately restricts our freedom as users. Moreover, there are *many* technically excellent, free/libre tools available that allow you do everything you do currently in SAS (and more). My advice is to avoid SAS.

Awk

Awk is a natural choice for this type of problem as it provides a simple processing model and language that enables a fast '*write-exectute-test*' feedback-loop. For instance, the code for transposing over the *group* field (as in Example 2) can be written in 19 lines:

```
BEGIN {
   FS="\t"; OFS="\t"
}

NR > 1 {
   if($(grp) != "")
      unique[$(grp)] += $(num)
}

END {
   for(group in unique)
      printf("%s\t", group)
   print ""

   for(group in unique)
      printf("%d\t", unique[group])
   print ""
}
```

If we saved this to a file called *transposeGroup.awk*, we could run it to transpose a text file holding our data (*data.txt*) like this:

```
$ gawk -f transposeGroup.awk grp=4 num=5 data.txt
```

Awk doesn't recognise field names, so you better know the column index of the fields you want to use (notice that we pass the column indexes for both the field containing the group values, and the numeric values to be aggregated, respectively). Although column indexes can be extracted easily ("`head -n1 data.txt | tr $'\t' $'\n' | cat -n`"), these indexes could change should the structure of our data change, e.g. if new fields where added to data.txt. A rule of thumb is to never rely on 'magic numbers' - in this case, the ability to refer to our fields by name would provide a more future-proof solution (typically, field names will change little over time, if at all).

Similarly, the code to transpose over the *group* and *id* fields can composed as a shell script in 76 lines:

```
#!/bin/bash

dataFile=$1
idField=$2
groupField=$3
numField=$4

gawk '
BEGIN {
   FS="\t"; OFS="\t"
}

# Get id column name (to print later)
NR == 1 && pass == 1 {
   idHeader = $(id)
}

# Get unique groups
FNR > 1 && pass == 1 {
   if($(grp) != "")
      unique[$(grp)] = 0
}
```

```
# print unique groups as headers
pass == 2 && FNR == 1 {
      printf("%s\t", idHeader)

      for(group in unique)
         printf("%s\t", group)
      print ""
}

# aggregate group values for each id
FNR > 1 && pass == 2 {
   rowId = $(id)
   if(currentId == "")
      currentId = rowId

   if(rowId != "" && $(grp) != "") {
      if(rowId == currentId) {
         unique[$(grp)] += $(num)
      }
      else {
         # print output line
         printf("%s\t", currentId)
         for(group in unique)
            printf("%d\t", unique[group])
         print ""

         # clear array values
         for(group in unique)
            unique[group] = ""

         # aggregate current row
         currentId = rowId
         unique[$(grp)] += $(num)
      }
   }
}

#print last row
END {
   printf("%s\t", currentId)
   for(group in unique)
      printf("%d\t", unique[group])
   print ""
}
' id=$idField grp=$groupField num=$numField pass=1 $dataFile pass=2 $dataFile
```

The fact that the code scans twice over the data file forces us to encapsulate the code in a shell script. To run this on our data, we would save the code in a file *transposeIdGroup.awk*, and execute the following:

```
$ ./transposeIdGroup.awk data.txt 1 4 5
```

Awk allows us to script the transpose functionality we want in few lines (I've left the 'row-to-column' transpose for the reader to figure out). However, these scripts are more suited to be executed in an ad-hoc fashion, than in a repeatable process (such as cron job) as it cannot handle the data changing structure.

rs

This UNIX tool only handles simple transposes ('row-to-column'), but deserves a quick mention for the sake of completeness. If we have the following data in *data_simple.txt*:

```
$ cat data_simple.txt
A       B       C
x1      y1      z1
x2      y2      z2
x3      y3      z3
```

We can transpose this by running:

```
$ cat data_simple.txt | rs -c$'\t' -T
A   x1  x2  x3
B   y1  y2  y3
C   z1  z2  z3
```

Now that we have a better idea of the methods available, we are better placed to decide which best fits our needs. You might have noticed, however, that there isn't a method that simultaneously matches all of the following requirements:

1. Doesn't need importing
2. Doesn't involve writing a script
3. Allows field-name matching (not only column indexes)
4. Accepts delimited text files
5. Can be run from the terminal

6. Provides different aggregation types (sum, count, average)
7. Executes in parallel

Enter **tpose**… a UNIX-based terminal program for transposing delimited text-files.

Simple transpose

A simple transpose ('row-to-column') needs no options. The data from Example 1 ("Raw table 1 - Sales volume data for a given year") can be processed by:

```
$ tpose data_ex1_simple.txt
Quarter Q1      Q2      Q3      Q4
Europe  2       3       3       4
Asia    5       4       5       6
US      3       1       2       3
```

Group transpose

To transpose a numerical variable over a a set of groups, we need to provide the `-G` (or `--group`) and `-N` (or `--numeric`) options, to specify the *group*, and *numeric* fields. The data from Example 2 ("Raw table 2 - Billing data for a given month") is transposed as follows:

```
$ tpose data_ex2_group.txt -Grevenue_group -Namount
rev_A   rev_B   rev_C
13.00   12.00   15.00
```

Group and ID transpose

Using the same data from Example 2 ("Raw table 2 - Billing data for a given month"), we can transpose over groups and id's via the `-I` (or `--id`), `-G` (or `--group`), and `-N` (or `--numeric`) options:

```
$ tpose data_ex2_group.txt -Icustomer_id –Grevenue_group -Namount
Customer_id     rev_A   rev_B   rev_C
1       2.00    3.00    0.00
2       0.00    0.00    6.00
3       0.00    9.00    9.00
4       11.00   0.00    0.00
```

There are a few points to note:

- Transposing over group, and ID fields requires the first line of the data file to be a header row with field names
- All operations are performed using floating-point values - values are output as such
- `-I, -G,` and `-N` options are case-insensitive (revenue_group = rEvEnUe_GrOuP)

Field indexes instead of names

It's sometimes easier to refer to fields by number - that is, the position relative to other fields. This is supported in tpose via the `-i` (or `–indexed`) option. If this option is used, tpose will expect a number instead of a name. If we take the same data from Example 2 ("Raw table 2 - Billing data for a given month"), we would perform the previous transpose operation by:

```
$ tpose data_ex2_group.txt -i –I1 –G2 -N3
Customer_id    rev_A   rev_B   rev_C
1       2.00    3.00    0.00
2       0.00    0.00    6.00
3       0.00    9.00    9.00
4       11.00   0.00    0.00
```

An easy way of getting the field indexes is via the following pipe (assuming tab delimiter):

```
$ head -n1 data_ex2_group.txt | tr $'\t' $'\n' | cat -n
     1 Customer_id
     2 Revenue_group
     3 Amount
```

Types of aggregation

Imagine that instead of transposing over groups (or groups and IDs) and summing the numerical field, we want to count each instance instead, or calculate an average using the sum of the numerical field and the count of each instance. We can do this with tpose by passing the type of aggregation via the `-a` (or `--aggregate`) option:

```
$ tpose data_ex2_group.txt -i –I1 –G2 -N3 -acount
customer_id    rev_A   rev_B   rev_C
1       1       1       0
2       0       0       1
3       0       2       1
4       2       0       0
```

```
$ tpose data_ex2_group.txt -i –I1 –G2 -N3 -aavg
customer_id    rev_A   rev_B   rev_C
1       2.00    3.00    nan
2       nan     nan     6.00
3       nan     4.50    9.00
4       5.50    nan     nan
```

Executing in parallel

Processing large files sequentially might take some time, so why not leverage the multiple cores in our machines? Files larger than one gigabyte, can be transposed in parallel via the `-P` (or `--parallel`) option:

```
$ tpose data_large.txt output.txt -P –Icustomer_id –Grevenue_group -Namount
```

Notice that in this case we specify an output file (`output.txt`) instead of letting tpose print everything to the screen.

Changing delimiter

If the data is not tab-delimited, we can specify a different delimiter via the `-d` (or `--delimiter`) option. Here we specify a comma for CSV files:

```
$ tpose data.csv output.txt -d, -P –Icustomer_id –Grevenue_group -Namount
```

Add a prefix/suffix to output fields

Suppose we wanted to add a prefix or suffix to the transposed group names (say, so that they follow a pattern and are easy to group with a wildcard later on). This can be achieved with the `-p` (`--prefix`) and/or `-s` (`--suffix`) options:

```
$ tpose data_ex2_group.txt -i –I1 –G2 -N3 -pxxx_ -s_yyy
customer_id    xxx_rev_A_yyy  xxx_rev_B_yyy  xxx_rev_C_yyy
1       2.00    3.00    0.00
2       0.00    0.00    6.00
3       0.00    9.00    9.00
4       11.00   0.00    0.00
```

Grab tpose from it's home page at: http://www.bitbucket.org/jmsmistral/tpose

If you have any bug reports or feature requests, please email me (jmsmistral@gmail.com) or submit a pull-request. Remember, tpose is free software (licensed under GPLv3)!

If you enjoyed this article, you can find more at www.jonathansacramento.com