# Calculating Changes and Differences Using PROC SQL — With Clinical Data Examples

Chang Y. Chung, Princeton University, Princeton, NJ

Lei Zhang, Celgene Corporation, Summit, NJ

## ABSTRACT

It is very common in the clinical data analysis to calculate various kinds of changes or differences. Despite the varied statistical objects and the quantities involved, there is a common programming pattern underlying these calculations. We make the common programming pattern explicit in order to help understand, communicate, and execute these calculations correctly, clearly, and easily. We demonstrate a straightforward way of implementing this pattern with versatile SAS® PROC SQL.

## INTRODUCTION

Suppose that we have a clinical trial data from a double-blind, randomized, parallel study that looks at the different effects of two drugs (A and B), and placebo on the physical growth of pediatric patients. Suppose also that the study has two treatment periods. Period I is a 2-visit placebo run-in period; Period II is a 3-visit double-blind treatment period. The patient's height is measured at each visit (measured in centimeters). Here are some of the possible statistical calculations you may perform:

1. The change between the mean height during the run-in and the mean height during the treatment period for each patient;
2. The change between the two heights for each patient: one at the last visit during the run-in period and another at the last visit during the treatment period;
3. The differences between the mean height during the run-in period and the height at each treatment visit for each patient (excluding those who don't have any observations in the treatment period);
4. The differences between three pairs of mean heights at each visits, where a mean is taken over all patients within a treatment arm;
5. The number of time units between visits for each patient.

All the above involve finding out the difference or change between some measurements on the (statistical) objects. We show that there is a common underlying programming pattern, or idiom. By recognizing the programming pattern, the implementation of the specific calculation will be easier and the written code more robust.

In the next section, we first present the traditional SAS DATA solutions to two of the tasks using a made-up data (see APPENDIX). Then we introduce the common underlying programming pattern that we call *Delta*. Based on the Delta pattern a PROC SQL solution is implemented in a straightforward way. We continue with a few more examples with the increasing complexity. We demonstrate ways to keep the code simple by taking advantage of SQL VIEWS and simple MACRO's.

## TRADITIONAL WAY

Given the hypothetical data (dataset HEIGHTS. see APPENDIX), the first task above (calculating the change between the mean height during the run-in and the mean height during the treatment period for each patient) can be achieved in several steps. First we sort the data by PATIENT; then we calculate the appropriate means using PROC MEANS. Notice that we calculate two means for each patient: a mean during the run-in period (the first and the second visits) and another during the treatment period (visit numbers 3 to 5). Finally, in the DATA step, we merge the means datasets and calculate the change.

```
proc sort data=heights out=sHeights;
  by patient;
run;

/* run-in period */
```

```
proc means data=sHeights noprint;
  by patient;
  var height;
  where 1 <= visit <= 2;
  output out=rmHeights(drop=_:) mean=rmHeight;
run;

/* treatment period */
proc means data=sHeights noprint;
  by patient;
  var height;
  where 3 <= visit <= 5;
  output out=tmHeights(drop=_:) mean=tmHeight;
run;

/* change */
data task1;
  merge rmHeights tmHeights;
  by patient;
  if missing(rmHeight) or missing(tmHeight) then do;
    change = .;
  end; else do;
    change = tmHeight - rmHeight;
  end;
run;
```

The second task (calculating the change between the two heights for each patient: one at the last visit during the run-in period and another at the last visit during the treatment period) can be handled a bit differently. Here we first separate the run-in's and treatments into their own datasets. Then each is sorted by the variable, PATIENT and VISIT before the subsequent data step selects only the last observation for a patient. The data step also re-name HEIGHT variable appropriately. The two datasets are then merged together in the end and the change is calculated.

```
/* separate the data set into two */
data rHeights tHeights;
  set heights;
  if      1 <= visit <= 2 then
    output rHeights;
  else if 3 <= visit <= 5 then
    output tHeights;
  else do;
    put "ERR" "OR: out-of-range visit.";
    stop;
  end;
run;

/* get the last visit height for each patient in the runIn data */
proc sort data=rHeights out=rHeightsSorted;
  by patient visit;
run;
data rlHeights;
  set rHeightsSorted(rename=(height = rlHeight));
  by patient visit;
  if last.patient then output;
  keep patient rlHeight;
run;

/* get the last visit height for each patient in the trt data */
proc sort data=tHeights out=tHeightsSorted;
  by patient visit;
```

```
run;
data tlHeights;
  set tHeightsSorted(rename=(height=tlHeight));
  by patient visit;
  if last.patient then output;
  keep patient tlHeight;
run;

/* put together the runIn and trt datasets and calc the change */
data task2;
  merge rlHeights tlHeights;
  by patient;
  if missing(rlHeight) or missing(tlHeight) then do;
    change = .;
  end; else do;
    change = tlHeight - rlHeight;
  end;
run;
```

We note several characteristics of the traditional approach. Although it is easy to understand (as it should be as being *traditional*), it requires multiple DATA and PROC steps. The number of steps does not stay the same – it rather varies depending on the problem. It is not easy to discern a clear, common, programming pattern, either. The above examples demonstrate that the code for one task may be quite different from the code to accomplish another, even though the tasks are similar. This suggests that when a task is modified slightly, the code should be changed significantly. There must be a better way and there is. Learning it starts with recognizing the similarities among the tasks.

## DELTA PATTERN

A programming pattern (our inspiration comes from the *Design Pattern* by Gamma et. al. (1995)) provides a convenient way to express recurring patterns in the code so that we can easily reuse common code or solution between projects and among programmers.

In order to apply this method, we first extract what are commonly involved in calculations of changes and differences. Once we make the abstraction, we then figure out, once and for all, a general way to deal with the group of similar tasks. Once we do that, handling each specific task becomes just a straightforward application of the established programming pattern.

In the clinical data analyses, the calculation often involves the characteristics of different statistical objects which can be patients, visits, combinations of both, and other meaningful clinical items. Sometimes we calculate differences of counts, of means, or of maxima. It is common to calculate the differences in terms of efficacy and safety endpoints. For a given statistical object, $i$, the difference can be expressed as:

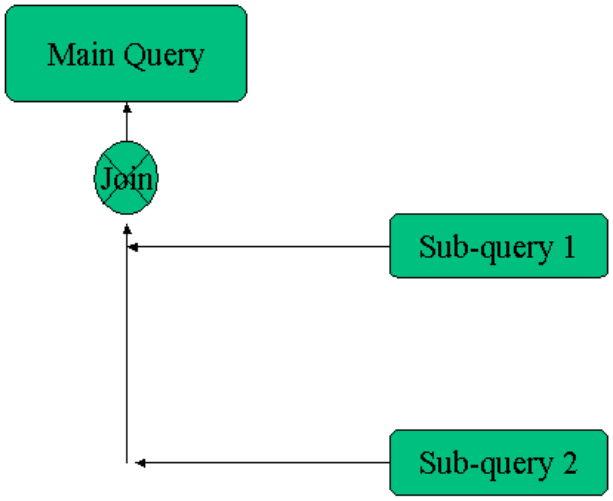
$$d_i = f_1(A_i) - f_2(B_i)$$

where

$d_i$ is the change or difference calculated for the (statistical) object $i$;
$A_i$, $B_i$, sets of values (or measurements) associated with the object $i$; and
$f_1$, $f_2$, some statistical functions of the measurements, such as mean, sum, max, and count.

We point out a few obvious but useful aspects of this formulation. Notice first that we have all three terms that share the same subscript indicating that the measurements are about the same object, $i$. That is, in calculating the changes or differences, we don't want to compare apples and oranges. On the other hand, the two statistical functions have different subscriptions indicating that they can be different statistical functions. And finally, a change and a difference can be calculated in the same way. By "change," we distinguish what are measured or observed before ($B$) and after ($A$); by "difference" we do not.

Now, a programming pattern can be readily developed to carry out the difference calculation. The pattern is expressed in terms of PROC SQL:




The Delta pattern consists of an outer-query (main-query) accompanied by two simple sub-queries (sub-queries 1 and 2). The role of the sub-query 1 and the sub-query 2 is to calculate $f_1(A_i)$ and $f_2(B_i)$, respectively. The main-query joins them while performing the subtraction. A specific calculation requires an implementer's attention to three aspects of the pattern. They are:

1. The specification that describes how to select (and aggregate) the statistical object, $i$, and how to acquire the two measurements on it ($A_i$ and $B_i$);
2. The functions $f_1$ and $f_2$ that further transforms $A_i$ and $B_i$; and
3. The merge key that is to be used to join the two sub-queries.

## PROC SQL WAY

Utilizing the Delta pattern, the task 1 can be implemented as the following. The task specifies that we calculate the change in the mean heights: one is the mean during the run-in period and the other during the treatment period.

```
proc sql;
  create table task1sql as
  select A.patient
       , A.mHeight as rmHeight
       , B.mHeight as tmHeight
       , B.mHeight - A.mHeight as change
  from
         ( select patient, mean(height) as mHeight
           from   heights
           where  1 <= visit <= 2 /* run-in */
           group by patient
         ) as A

         full join

         ( select patient, mean(height) as mHeight
           from   heights
           where  3 <= visit <= 5  /* treatment */
           group by patient
         ) as B

         on  A.patient = B.patient
;
```

```
quit;
```

The implementation faithfully resembles the Delta pattern. We have two simple sub-queries in implementation as with the programming pattern. The outer query, both in the pattern and the implementation, joins the two sub-queries by matching on the key field (here the variable, PATIENT).

Similarly, the task 2 above can be implemented as the following. Notice that the overall structure of the main and two simple sub-queries remain the same.

```
proc sql;
  create table task2sql as
  select A.patient
       , A.height as rlHeight
       , B.height as tlHeight
       , B.height - A.height as change
  from
         ( select patient, height
           from   heights
           where  1 <= visit <= 2 /* run-in */
           group by patient
           having visit = max(visit) /* last visit */
         ) as A

         full join

         ( select patient, height
           from   heights
           where  3 <= visit <= 5  /* treatment */
           group by patient
           having visit = max(visit) /* last visit */
         ) as B

         on  A.patient = B.patient
   ;

quit;
```

Compared to the previous one, we find that the main query does not change at all (except the variable names). The two simple sub-queries remain mostly the same. They now have HAVING clause that singles out the patient's last visit within each of the run-in and treatment period.

Again, we see that the Delta pattern can accommodate different problems and data structures. In many cases, all we need to do is to change the WHERE/HAVING clause(s). Using a pattern makes it easy to verify the code. Especially in this case, the sub-queries are not correlated with the main query (that is they are *simple* queries), making it a breeze to read through and understand what the code does.

The joins do not always have to be the full join. Left or right joins come in handy in some situations. For example, the task 3 above requires that we remove any observations from the patient who do not have any observations in the treatment period. The Delta pattern does not have to be altered. By right join on the patients who did make a visit during the treatment period, we automatically drop those who did not.

```
proc sql;
  create table task3sql as
  select B.patient
       , B.visit
       , B.height as tHeight  /* treatment height at the visit */
       , A.mHeight as rmHeight /* run-in mean */
       , A.mHeight - B.height as diff
  from
       ( select patient, mean(height) as mHeight
           from   heights
```

```
            where  1 <= visit <= 2 /* run-in */
            group by patient
          ) as A

          right join

          ( select patient, visit, height
            from   heights
            where  3 <= visit <= 5  /* treatment */
          ) as B

          on  A.patient = B.patient

    order by B.patient, B.visit
    ;
quit;
```

Again, we find that the Delta pattern can accommodate different problems and data structures. In many cases, all we need to do is to change the WHERE/HAVING clause(s). Using a pattern makes it easy to verify the code. Especially in this case, the sub-queries are not correlated with the main query (that is they are *simple* queries), making it a breeze to read through and understand what the code does.

## GENERALIZATION AND SIMPLIFICATION

Delta pattern generalizes naturally to calculate more than one difference or change. Let's consider the task 4. In order to complete the task, we have to calculate the mean heights for the patients in the same treatment group at each of the three visits during the treatment period. Let's call them mt1, mt2, and mt3, for the mean of the treatment group1, 2, and 3, respectively. Then we are to calculate the three differences: between mt1 and mt2, between mt1 and mt3, and finally between mt2 and mt3. In terms of the programming pattern, this is a straightforward generalization of going from a one-difference-two-sub-query pattern to a two-difference-three-sub-query one.

In doing the implementation, however, we notice that the almost same simple sub-queries are used three-times, which is one time too many and calls for a way to avoid repeatedly coding the same thing. Here we demonstrate utilizing a parameterized macro.

```
%macro mt(trt=);
  (select visit, mean(height) as mHeight
   from   heights
   where  trt = (&trt.)
   group by visit)
%mend  mt;

proc sql;
  create table task4sql as
  select Z.visit
       , A.mHeight as mt1
       , B.mHeight as mt2
       , C.mHeight as mt3
       , (calculated mt1 - calculated mt2) as diff12
       , (calculated mt1 - calculated mt3) as diff13
       , (calculated mt2 - calculated mt3) as diff23
  from   (select distinct visit from   Heights) as Z
         left join %mt(trt=1) as A on Z.visit = A.visit
         left join %mt(trt=2) as B on Z.visit = B.visit
         left join %mt(trt=3) as C on Z.visit = C.visit
  where  3 <= z.visit <= 5 /* treatment period */
  order by Z.visit
  ;
quit;
```

PROC SQL also supports SQL VIEWS that can also be parameterized using (global) MACRO variables and SYMGET() function.

## CONCLUSION

In this paper, we introduce the pattern-based programming technique using PROC SQL. We demonstrate the powerful and versatile programming pattern, *Delta*, which calculates differences and changes between varied quantities. It is capable of coping with diverse data structures. We also show that the Delta pattern is easily generalized to deal with differences among three and more quantities. Combined with the MACRO facility or SQL VIEWS, the implementation remains simple and easily readable.

## APPENDIX

The hypothetical data set, HEIGHTS, is created using the following piece of SAS DATA step code.

```
data heights;
  input patient trt visit height visDate ddmmyy.;
  format visDate date.;
datalines;
100 1 1  123.45 010201
100 1 2  123.49 090401
100 1 3  124.17 180801
100 1 4  124.60 211201
100 1 5  124.70 270402
200 2 1  122.45 030100
200 2 2  122.49 040400
200 2 3  123.12 020800
200 2 4  123.60 011200
200 2 5  124.15 030401
300 3 1  125.45 070201
300 3 2  125.49 050601
300 3 3  125.53 061101
300 3 4  125.61 030502
300 3 5  126.10 080902
400 3 1  125.45 020101
400 3 2  125.49 030501
400 3 3  125.90 010901
500 1 1  127.45 150100
500 1 2  127.49 150500
600 1 1  125.41 161000
600 1 2  124.29 140201
600 1 3  127.17 150601
600 1 4  127.60 141001
600 1 5  128.10 160202
;
run;
```

## REFERENCES


Fuller, J. 1999. "Programming Idioms Using SET Statement." The 24th Annual Proceedings of SAS Users Group International Conference.

Gamma, E, R. Helm, R.Johnson, and J.Vlissides. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Tabladillo, M. 2003. "Application Refactoring With Design Patterns." *The 28th Annual Proceedings of SAS Users Group International Conference*.

Zhang, Lei, and D. Yi. 1998. “Working With Subquery in The SQL Procedure.” *The 15th Annual Proceedings of the NorthEast SAS Users Group Conference*.

## ACKNOWLEDGMENTS

Lei would like to thank Izabella B. Peszek, who generously provided helpful comments and encouragements.



## CONTACT INFORMATION

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

Chang Y. Chung
Princeton University
#216 Wallace Hall
Princeton, NJ 08540
(609) 258-2360
http://changchung.com
chang_y_chung@hotmail.com

Lei Zhang
Celgene Corporation
86 Morris Avenue
Summit, NJ 07901
(908) 673-9477
lezhang@celgene.com