Paper 088-2008

Labor-saving SQL Constructs

Brian Fairfield-Carter, ICON Clinical Research, Redwood City, CA
David Carr, ICON Clinical Research, Redwood City, CA

ABSTRACT

Simplicity is one of the most difficult things to achieve. This statement may seem counter-intuitive, but consider that the fundamental purpose underlying the development and adoption of SAS language elements is really to reduce the number of program steps required to complete a task, or in other words, to simplify SAS code. Adopting new language elements and exploring new techniques takes work, but the reward can be code that is simpler and easier to understand, and which is therefore more transportable and maintainable.

In general, 'simplified' SAS code minimizes

- DATA, PROC, MERGE and transposition steps
- Replication of WORK datasets
- 'Macrotization' and fragmentation of tasks

While PROC SQL can offer new approaches to common data manipulation problems, it is essentially a language unto itself, and for this reason its use is often resisted. PROC SQL can, however, be a tremendous asset to code simplification, so acquiring knowledge of even a few SQL constructs is a worth-while investment of time and effort. This paper illustrates six applications of PROC SQL which have proven useful in meeting tenets of 'code simplification':

- Capturing summary statistics 'on the fly'
- Embedding sub-queries in 'where' statements, and in conditional ('case') statements
- Cartesian joins
- Generating repetitive SAS statements as delimited lists
- Summarizing distinct values within grouping levels
- 'Reflexive' joins and cross-record comparisons

These techniques make use of SQL constructs that range from simple to more advanced, and will hopefully be of interest to both novice and experienced SQL users.

INTRODUCTION

With modern processing power, and with the relatively small databases common to clinical trials, it can seldom be argued that program efficiency can be measured by processor time. The greatest source of inefficiency tends to be code that is difficult for a programmer to read, debug and maintain. Humans have a fairly limited capacity for short-term storage of information; this means that code that requires you to remember what went on in a dozen preceding steps will be very difficult to decipher. Code that generates and uses large numbers of intermediate WORK datasets, and that scatters functionality across numerous macros will also be difficult to read and understand. How do we simplify code in order to minimize the number of steps and by-products? The answer is to make appropriate use of the language elements and programming constructs that have been designed to help achieve this end.

Admittedly, many applications of PROC SQL offer no real improvement over comparable data-step approaches, and others are cryptic, obscure and hard to conceptualize; in many cases, there is no incentive to use SQL other than that it offers the opportunity of a fresh approach to stale problems. In addition, in a number of cases PROC SQL, since it strives to be ANSI-compliant rather than to be sympathetic to the idiosyncrasies of SAS programming, is clearly inferior to the data step. As a case in point, PROC SQL offers no analogue to the 'do-loop' construct, and no reasonable equivalent of SAS arrays.

That said, there are without question SQL constructs that save on programming steps, and make program flow cleaner and more intuitive. Among these are:

- Functions like 'mean()', 'min()', 'max()', etc. operating on vertically-structured data
- Compact syntax for initializing macro variables and lists, using the 'into' keyword and 'separated by' clause
- Nested/embedded sub-queries
- The 'distinct' keyword
- Joining datasets on 'inequalities' (i.e. using '>' and '<' operators), and Cartesian joins

This paper describes six applications of these 'labor-saving' constructs, and provides illustration using examples taken from real-world data manipulation and summarization problems.

CAPTURING SUMMARY STATISTICS 'ON THE FLY'

Summary statistics (n, mean, min, max, etc.) can be retrieved from a number of sources, including PROC UNIVARIATE, PROC MEANS, and data step functions such as mean(), min(), max(). While these statistics may be reported directly in summary tables, these values, in particular measures of 'central tendency', often serve other purposes, one of the more common being that of imputation. For certain types of sub-group analyses, missing values are sometimes replaced with the population mean, or the mean within some designated grouping level.

It's fairly easy to envision how such a substitution might take place: PROC UNIVARIATE generates the summary statistic (within each level of any designated 'by' groups), which is then merged with the original dataset, and missing values in the target variable are replaced with the calculated statistic. To reduce the number of steps and by-products, we might consider the datastep 'mean()' function:

```
if myvar=. then myvar=mean(myvar);
```

While this statement does not generate any error messages, it also does not have the desired result. The 'mean()' function actually expects to receive a 'horizontal' list of variables, and calculates the mean within each iteration of the datastep (rather than pooling observations within each level of a 'by' group, as PROC UNIVARIATE does). This means that where 'myvar' has a missing value, 'mean(myvar)' will also produce a missing value; 'mean(myvar)' will never return anything other than the value of 'myvar'.

In contrast, in PROC SQL the 'mean()' function does operate on vertically-structured data, pooling records within each level of any specified 'by' groups (in fact, the function produces the log message "NOTE: The query requires remerging summary statistics back with the original data.", giving some insight into the processing going on behind the scenes). The entire substitution can be achieved in a single line of code:

```
proc sql;
  create table mydata as select treat,
   case when myvar>. then myvar else mean(myvar) end as myvar from test
   group by treat;
quit;
```

The 'case' statement allows for conditional processing, and is the SQL equivalent of the 'if' statement in a data step. Note that the mean is captured (and substituted), within each level of the 'by' group, 'on the fly', without requiring the creation of any WORK datasets or extraneous variables.

EMBEDDING SUB-QUERIES IN 'WHERE' STATEMENTS, AND IN CONDITIONAL ('CASE') STATEMENTS

Program branching and data filtering often make use of 'list processing', where data values are compared to a list of allowable values. The construct makes use of the 'in' operator, and takes a number of very familiar forms:

```
if patient in(1001, 1002, 1003) then do;
  ...
end;

proc print data=test;
  where patient in(1001, 1002, 1003);

proc sql;
  select * from test where patient in(1001, 1002, 1003);
quit;
```

Using PROC SQL, lists can easily be generated and resolved as delimited text strings stored in macro variables:

```
proc sql;
  select patient into :patlist separated by "," from _subset;
quit;

proc print data=mydata;
  where patient in(&patlist);
```

What if we wanted to compare a value against a list of allowable values, but without creating a macro variable? In

other words, what if we wanted the comparison to be made simply against a list of values returned by some data-querying step? A datastep can not 'return' a list; if such a construct existed it might look something like this:

```
proc print data=test;
  where patient in( set subset(keep=patient) );
run;
```

This is of course nonsensical, but PROC SQL actually supports a construct that's not much different:

```
proc sql;
  select * from test where patient in(select patient from subset);
quit;
```

In other words, the sub-query "`(select patient from subset)`" effectively returns the list of patients that each value of 'patient' in dataset 'test' is compared against, and records from 'test' are filtered according to membership in the list. The same construct can be applied to conditional processing as well:

```
proc sql;
  select *,
    case when patient in(select patient from subset) then 1
         else 0 end as in_subset
  from test;
quit;
```

Again, the sub-query returns the list, and membership in the list determines the value assigned to the new variable 'in_subset'. This construct is useful since it avoids the need to create macro variables, and it also allows data to be filtered and processed according to the presence of data values on other datasets without actually requiring merges or joins, or the creation of intermediate WORK datasets.

This approach is particularly useful when creating population flags, since population membership is usually determined by testing for patient inclusion on a number of data sets (and sub-sets). A typical definition for the Safety population is something like "randomized patients who receive any amount of study drug". This definition can be implemented very simply with the following:

```
proc sql;
  create table all_screened as select *,
    case when patient in(select distinct patient from randlist) AND
              patient in(select distinct patient from drgadmin where dose>0) then 1
         else 0 end as safety
  from all_screened
  ;
quit;
```

CARTESIAN JOINS

Cartesian joins, where all possible matches are made between datasets, usually only take place by accident (for instance, where a unique key has not been specified, or where the key does not actually uniquely identify records). The following code, since it uses an unqualified (full) join and specifies no unique key, will create a dataset that matches every observation from dataset 'a' to every observation from dataset 'b' (so the record count of the resultant dataset 'c' will be the product of the record counts from datasets 'a' and 'b'):

```
proc sql;
  create table c as select * from a, b;
quit;
```

Sometimes these artifacts of Cartesian joins (inflation of record count) are actually desirable. Analysis of contingency tables (Chi-square and Fisher's Exact tests), and other analyses based on 'Yes/No' response data, require the presence of records indicating a negative response in addition to those indicating a positive response. Depending on the nature of the data in question, the absence of a record may be taken as equivalent to a negative response. This is usually true of Adverse Event data: records are only created for Adverse Events that were observed, but not to indicate events that failed to occur. This means that this type of event data will only contain positive responses, so in order to run a Fisher's exact test, records for negative responses have to be created.

Essentially, we need a record for every patient for every class of Adverse Event included in the analysis, or in other words, we need the Cartesian product of all unique patient identifiers and all unique Adverse Event classes. These two record sets can easily be created using the 'distinct' keyword:

```
select distinct patient from demog
...
select distinct prefterm from ae
```

A comma-delimited list of unique patient identifiers could of course be created, and the Cartesian product of patient and Adverse Event term could be generated using a 'do' loop:

```
data ae_c;
  set unique_ae;
  do patient=&patlist;
    output;
  end;
run;
```

However, if the study includes a large number of patients, it's possible that internal SAS limitations on string length will be exceeded. To get the Cartesian product using SQL, simply combine the record sets with an unqualified join:

```
proc sql;
  create table ae_c as select * from
    (select distinct patient from demog) ,
    (select distinct prefterm from ae);
quit;
```

Note that a 'full' join can be specified explicitly, but in this case SAS will report a syntax error if the 'on' keyword (normally used to specify the unique key) is not provided. Of course, for this particular join we have no unique key to provide, so we can use a dummy value as a place-holder ("ON 1"):

```
proc sql;
  create table ae_c as select * from
    (select distinct patient from demog) FULL JOIN
    (select distinct prefterm from ae)
  ON 1;
quit;
```

With this complete dataset, matching all patients to all Adverse Event preferred terms, all that remains is to merge back with the original Adverse Event dataset, and flag records as indicating the presence or absence of the given event type for the given patient.

GENERATING REPETITIVE SAS STATEMENTS AS DELIMITED LISTS

One of the more common applications of data transposition is to re-structure datasets to allow for conditional processing based on single or small numbers of variables, in order to avoid statements like this:

```
if var1=10 or var2=10 or var3=10 ... or var<n>=10 then do;
```

In some cases, analysis datasets have to be structured in a 'horizontal' orientation, with a single record per patient (a common example being the usual 'patient characteristics' or 'demography' analysis dataset). Variables are often created to hold dates and other information for key scheduled events; in oncology studies, typical examples are cycle start dates, and dose levels at the start of each cycle. Analyses are often sub-set by dose levels at key study milestones, like cycle start dates; for example, patients who received a 450 mg dose level at any cycle start might be identified using a filter like:

```
where admlev1=450 or admlev2=450 or admlev3=450 or … or admlev<n>=450;
```

While a repetitive statement like this would usually prompt data transposition, so that the same filter could be achieved using a statement like

```
where admlev=450
```

, since SQL offers a very compact syntax for creating delimited lists, why not avoid the transposition step and simply generate the statement dynamically as a list, delimited by the word 'or'? For instance, knowing that each variable in the series shares the prefix 'admlev', and assuming the 'demography' dataset was called 'demog', and was stored in a library called 'derived', we could use the 'sashelp.vcolumn' SAS dictionary table to retrieve the relevant variable names, and generate a SAS statement as a delimited list, as follows:

```
select trim(left(name))||"=450" into :where_ separated by " or "
  from sashelp.vcolumn
  where upcase(libname)="DERIVED" & upcase(memname)="DEMOG" &
        upcase(name) like("ADMLEV%");
```

The 'like' keyword allows for wild-card syntax, and in this case returns all variables starting with the word 'ADMLEV'. This makes the approach flexible, since it does not require that the number of variables in the 'admlev1…admlev<n>' list be known in advance. The macro variable '&where_' is assigned a string of the form

```
admlev1=450 or admlev2=450 or admlev3=450 ... or admlev<n>=450
```

, and can be substituted into a conditional statement:

```
select * from derived.demog where &where_;
```

As a variant, this approach can also be applied to the 'case' (conditional) statement; in oncology studies, adverse events are occasionally summarized by dose level at the time the adverse event started. Again, assuming a horizontal structure in the 'patient characteristics' dataset, where variables cycstdt1…cycstdt<n> store cycle start dates, and variables cycdose1…cycdose<n> store dose levels at each cycle start, to determine the dose level at the time of the AE start, a conditional statement such as this might be employed:

```
case
   when cycstdt<n> <=aestdt then cycdose<n>
   ...
   when cycstdt3<=aestdt then cycdose3
   when cycstdt2<=aestdt then cycdose2
   when cycstdt1<=aestdt then cycdose1
   else . end as doselev
```

(Note that in 'case' processing, once a condition in the list is satisfied, no further conditions are tested; for this reason, conditions must be tested in reverse order, starting with the last cycle start date/dose level and working backwards to the first). As with the first example, this statement contains a lot of repetition, but can be generated as a list:

```
create table temp as select name from sashelp.vcolumn
   where upcase(libname)="DERIVED" & upcase(memname)="DEMOG"
   & upcase(name) like("CYCSTDT%")
   order by name desc;

select
   " when "||trim(left(name))||"<=aestdt then cycdose"||substr(name,length(name))
   into :case_ separated by " "
   from temp;
```

Variable names cycstdt1…cycstdt<n> are first sorted in descending order, so that conditions in the 'case' statement are declared in the desired order. The macro variable "&case_" resolves to something like this:

```
when cycstdt8<=aestdt then cycdose8 when cycstdt7<=aestdt then cycdose7
when cycstdt6<=aestdt then cycdose6 when cycstdt5<=aestdt then cycdose5
when cycstdt4<=aestdt then cycdose4 when cycstdt3<=aestdt then cycdose3
when cycstdt2<=aestdt then cycdose2 when cycstdt1<=aestdt then cycdose1
```

, and can be used in the following way:

```
create table ae_ as select *, case &case_ else . end as cycdose from ae;
```

SUMMARIZING DISTINCT VALUES WITHIN GROUPING LEVELS

Frequency counts are the most common type of statistical summary in clinical trial reporting. A particular form of frequency count, the adverse event summary, provides a staple of safety reporting. Adverse event summaries tend to have a couple of twists that make them more difficult to program than most other types of frequency counts: adverse events are typically recorded in 'log' format, meaning the adverse event database often contains multiple records sharing the same combination of patient identifier, body system, and preferred term, but each patient is counted only once for each combination of body system and preferred term. In addition, adverse event tables actually consist of three summaries: the number of patients reporting any adverse event, the number of patients reporting any adverse event within each body system (system organ class), and the number of patients reporting any adverse event within each body system/preferred term combination. These three levels can be easily seen in the following idealized/typical adverse event summary:

System Organ Class Preferred Term	Treatment 1 (N=19)	Treatment 2 (N=26)	All Patients (N=45)
At Least One TEAE	18 (94.7%)	25 (96.2%)	43 (95.6%)
Blood and lymphatic system disorders	16 (84.2%)	24 (92.3%)	40 (88.9%)
Anaemia	15 (78.9%)	23 (88.5%)	38 (84.4%)
Haemolysis	3 (15.8%)	5 (19.2%)	8 (17.8%)
Leukopenia	3 (15.8%)	1 (3.8%)	4 (8.9%)
Lymphopenia	1 (5.3%)	1 (3.8%)	2 (4.4%)
Febrile neutropenia	0 (0.0%)	1 (3.8%)	1 (2.2%)
Haemolytic anaemia	1 (5.3%)	0 (0.0%)	1 (2.2%)
Leukocytosis	0 (0.0%)	1 (3.8%)	1 (2.2%)
Lymphadenopathy	0 (0.0%)	1 (3.8%)	1 (2.2%)
Thrombocytopenia	1 (5.3%)	0 (0.0%)	1 (2.2%)
Cardiac disorders	4 (21.1%)	1 (3.8%)	5 (11.1%)
Sinus tachycardia	3 (15.8%)	0 (0.0%)	3 (6.7%)
Cardiac failure congestive	0 (0.0%)	1 (3.8%)	1 (2.2%)
Cardiopulmonary failure	1 (5.3%)	0 (0.0%)	1 (2.2%)
Ear and labyrinth disorders	0 (0.0%)	1 (3.8%)	1 (2.2%)
Vertigo	0 (0.0%)	1 (3.8%)	1 (2.2%)
Eye disorders	1 (5.3%)	1 (3.8%)	2 (4.4%)
Photopsia	0 (0.0%)	1 (3.8%)	1 (2.2%)

PROC SQL provides, via the 'distinct' keyword, the equivalent of the 'nodupkey' qualifier in PROC SORT: a 'distinct' select clause returns only the unique values of specified variables. For instance, the following select statement will return the unique combinations of variables a, b, and c from the source dataset 'mydata':

```
select distinct a, b, c from mydata
```

, while the following statement will assign a count of the number of unique values of 'patid' to the variable 'patcount':

```
select count(distinct patid) as patcount from mydata
```

The 'distinct' keyword applies very naturally to the three summary levels in an adverse event table; each summary level can itself be thought of as a SQL query: select distinct patient identifiers among patients reporting any adverse events, select distinct patient identifiers within each body system class, and within each combination of body system and preferred term. The only thing missing is a way of declaring the body system and body system/preferred term grouping levels, but this can be solved by using a 'group by' clause. The three summary levels can therefore be described using the following three queries:

```
select count(distinct patient) from aedata;
select count(distinct patient), bodysys from aedata group by bodysys;
select count(distinct patient), bodysys, prefterm from aedata
       group by bodysys, prefterm;
```

The example table shown above includes the very common vertical stratification 'treatment group', including an 'all patients' category which pools all treatment groups. This stratification is usually dealt with as an additional 'by' group, but this then requires an additional transposition step. In order to reduce transposition, consider the following 'rule of thumb':

Create vertical table divisions using macro loops, and horizontal table divisions using 'by' groups.

In other words, create each vertical division (in this case, treatment group) as a separate dataset, where the

horizontal divisions ('any AE', 'any AE within body system', 'any AE within body system/preferred term combination') are created using 'by' groups, and then merge the datasets via the 'by' groups. Percentage calculations within each vertical division will require a denominator specific to the table division. If the denominator is simply the number of patients in the given treatment group, it can be calculated for each treatment group using the following statement:

```
select count(distinct patient) into :n1 - :n3 from demog group by trt;
```

(This assumes the source dataset has been 'inflated' to include records pooling treatment groups 1 & 2 to create treatment group 3).

The frequency count will require a formatted string of the form 'xx (xxx.x%)'; to cut down on repetitive code, the code to create this string can be assigned to a macro variable (using the masking function '%nrstr()' to prevent embedded macro references from resolving at 'compile' time):

```
%let str=
  %nrstr(put(count(distinct patid),3.0)||" ("||
         put(count(distinct patid)/&&n&trt*100,5.1)||"%%)");
```

The entire frequency count can now be calculated as follows:

```
%do trt=1 %to 3; /*VERTICAL DIVISIONS: TREATMENT GROUP*/
  proc sql;
   create table _&trt as
     select * from (

    /*AT LEAST 1 AE*/
    (select distinct "At Least One TEAE" as bodysys,
            "At Least One TEAE" as prefterm,
       %unquote(&str) as _&trt
       from ae where trt=&trt)

    union all

    /*AT LEAST 1 AE WITHIN BODY SYSTEM*/
    (select distinct bodysys,
            "At Least One TEAE" as prefterm,
       %unquote(&str) as _&trt
       from ae where trt=&trt
       group by bodysys)

    union all

    /*AT LEAST 1 TEAE WITHIN BODY SYSTEM AND PREFERRED TERM*/
    (select distinct bodysys, prefterm,
       %unquote(&str) as _&trt
       from ae where trt=&trt
       group by bodysys, prefterm)
     )
     order by bodysys, prefterm;
  quit;
%end;
```

The three main horizontal table sections ('at least one AE', 'AE within body system', and 'AE within body system/preferred term') are created as three SQL queries, the results of which are concatenated via the 'union all' statement. Unique patient identifiers are counted within each query via 'count (distinct patient)', and 'group by' clauses are used to calculate frequencies within each designated level (within each body system, or within each body system/preferred term combination), to produce horizontal divisions. The code to create the formatted summary values is resolved from the macro variable '&str' by means of the function '%unquote()'.

Vertical divisions are created by means of the macro loop and associated filter 'where trt=&trt'; three datasets (named _1, _2, and_3) are created:

```
bodysys                prefterm                _1
-------------------------------------------------------
At Least One TEAE      At Least One TEAE       xx ( xx.x%)
SOC Name 1             At Least One TEAE       xx ( xx.x%)
SOC Name 1             PT Name 1               xx ( xx.x%)
SOC Name 1             PT Name 2               xx ( xx.x%)
     ----------------(etc.)---------------------

bodysys                prefterm                _2
-------------------------------------------------------
At Least One TEAE      At Least One TEAE       xx ( xx.x%)
SOC Name 1             At Least One TEAE       xx ( xx.x%)
SOC Name 1             PT Name 1               xx ( xx.x%)
SOC Name 1             PT Name 2               xx ( xx.x%)
     ----------------(etc.)---------------------

bodysys                prefterm                _3
-------------------------------------------------------
At Least One TEAE      At Least One TEAE       xx ( xx.x%)
SOC Name 1             At Least One TEAE       xx ( xx.x%)
SOC Name 1             PT Name 1               xx ( xx.x%)
SOC Name 1             PT Name 2               xx ( xx.x%)
     ----------------(etc.)---------------------
```

The final 'output-ready' dataset can then be created simply by merging datasets _1, _2, and _3 by body system and preferred term.

'REFLEXIVE' JOINS AND CROSS-RECORD COMPARISONS

The 'reflexive' join, where a dataset is joined to itself, is useful because it allows you to assemble values originating on different records on the same record, allowing in effect for 'cross-record comparisons', and it accomplishes this without necessitating the creation of additional temporary WORK datasets.

The reflexive join is not unique to SQL; the same thing can also be done in a MERGE step:

```
data DATA_;
  merge DATA_(keep=<variables> where=(<conditions>) rename=(<variables>))
        DATA_(keep=<variables> where=(<conditions>) rename=(<variables>));
  by <key variables>;
run;
```

A common application of this type of merge/join is in capturing baseline values, in preparation for calculating change-from-baseline:

```
proc sql;
   create table DATA_ as select l.*, r.baseline from DATA_ l
  left join
   (select patient, param, value as baseline from DATA_ where baseflag=1) as r
  on l.patient=r.patient & l.param=r.param;
quit;
```

Aside from the fact that the SQL join does not require the source dataset to be sorted by the key variables, while the merge step does, where the reflexive merge differs from the reflexive join is in that the merge step does not allow records to be matched based on unequal relationships between data values; the merge step only allows for an exact match on key variables. In contrast, the SQL join can be qualified by unequal relationships between key variables. For example, if in the preceding example we wanted to include the baseline value only on post-baseline records, we could modify the join as follows:

```
proc sql;
 create table DATA_ as select l.*, r.baseline from DATA_ l
left join
 (select patient, param, value as baseline, date from DATA_ where baseflag=1) as r
```

```
 on l.patient=r.patient & l.param=r.param & l.date>r.date;
quit;
```

The inequality 'l.date>r.date' matches baseline values to records from the source dataset having a higher value for the variable 'date'; such a construct does not exist for the 'merge' step, meaning that additional statements and extraneous variables would be required (for instance, 'date' would probably have to be renamed to something like 'baseline_date' on the baseline record set, to allow for a comparison between observation date and baseline date).

A REFLEXIVE JOIN 'CASE STUDY': LINEAR INTERPOLATION

The SQL reflexive join with unequal matching is an approach perfectly suited to linear interpolation. Specifically, the first step in linear interpolation is to assemble so-called 'boundary values' (non-missing values bracketing one or more missing values) onto a single record. Put another ways, this involves matching each non-missing value to the next, nearest (usually in chronological sequence) non-missing value. A slope is then calculated between boundary values as the 'vertical' difference divided by the 'horizontal' difference, and intervening missing values are imputed by adding to the 'left-hand' boundary value of slope multiplied by increment. These aspects of linear interpolation are illustrated in Figure 1.

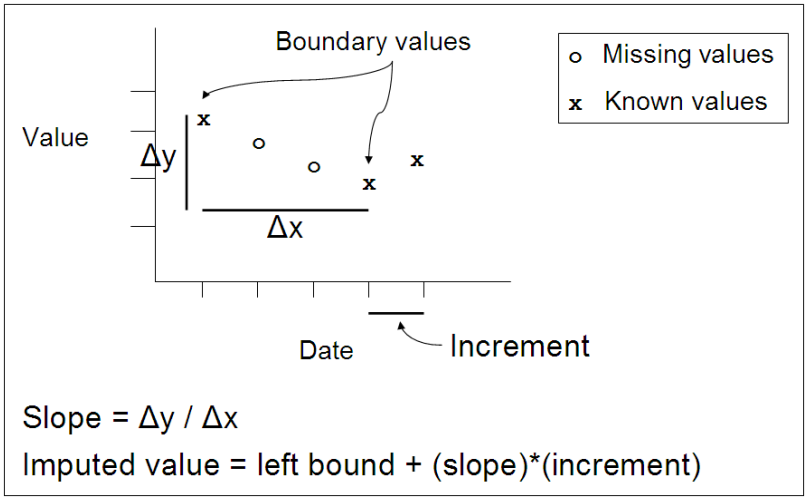

Figure 1: Boundary values, slope and imputation in linear interpolation

The following code excerpt assembles boundary values and calculates the slope on a dataset where the x-axis is provided by a date variable (and therefore increments by whole numbers); although the code is somewhat dense, the task is accomplished in a single step, without generating any intermediate WORK datasets.

```
proc sql;
 create table raw as
 select l.*, r.date-l.date as diff,
        r.nextdate, r.nextvalue,
        (l.value-r.nextvalue)/(l.date-r.nextdate) as slope
  from
    (select * from raw where value>.) as l left join
    (select patient, date, date as nextdate, value as nextvalue
      from raw where value>.) as r
 on l.patient=r.patient and l.date < r.nextdate
 group by l.patient, l.date
 having diff=min(diff);
quit;
```

A 'left-hand' record set is first created to hold all non-missing values:

```
(select * from raw where value>.) as l
```

, and a corresponding 'right-hand' record set is created, also holding non-missing values:

```
(select patient, date, date as nextdate, value as nextvalue
   from raw where value>.) as r
```

Values from the left-hand and right-hand record sets become lower and upper boundary values, respectively, as a result of the inequality '`l.date < r.nextdate`' in the join. However, this inequality on its own does not capture the nearest non-missing values (in fact, a Cartesian join takes place within the confines of the inequality), so a further qualification is imposed such that records written to the output dataset must, for each patient/date combination, have a minimum difference between date values (the variable 'diff' contains the difference between left- and right-hand date values):

```
group by l.patient, l.date
having diff=min(diff)
```

Having assembled boundary values and calculated the slope, all that remains is to generate records for the imputed values. Since PROC SQL does not support anything analogous to the 'do' loop, it is much easier to handle this task in a datastep:

```
data imputed;
  set raw;
  original=value;
  output;
  increment=0;
  if ((nextdate-date) > 1) then do i = (date+1) to (nextdate-1);
    date=i;
    increment+1;
    value=original+(slope*increment);
    output;
  end;
run;
```

Each original (non-imputed) value is output, and then for each day (unit increment) in the date range between boundary values, a record is generated to hold the imputed value, calculated as the lower boundary value plus slope multiplied by increment.

CONCLUSION

For datastep enthusiasts, PROC SQL might seem to require a return to the 'steep part of the learning curve', and for this reason the use of PROC SQL is sometimes controversial. SQL is, however, without question a boon to SAS programming, since it promotes fresh and alternative approaches to common data manipulation and summarization problems. This paper offered practical illustration of a number of useful SQL constructs, showing how these constructs can be used to make code more readable and efficient. The simplicity of many of these techniques, and the ease with which they can be adopted, will hopefully make a compelling argument even to SQL skeptics.

ACKNOWLEDGMENTS

The authors would like to thank all our friends at ICON for their great ideas, enthusiasm and support.

CONTACT INFORMATION

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

Brian Fairfield-Carter
ICON Clinical Research
Email: fairfieldcarterbrian@gmail.com

David Carr
ICON Clinical Research