Performing Macro Iteration on Consecutive Lines of a Dataset Using PROC SQL
Syloke J. Soong, Allegro MicroSystems, Inc.

Abstract
You have a dataset where each line is to be consecutively extracted and its fields used as arguments or values for a repeated set of operations.

There are two techniques based on PROC SQL to identify, and progress to, the next row of a dataset within a macro. One is an invasive technique and the other non-invasive. The non-invasive technique does not require modifying the dataset used for iteration. While the invasive technique requires the adjoining of an _N_ column.

As each row progresses, the SELECT INTO phrase becomes useful to extract values of fields to be used as arguments to any operation within the macro loop.

Both methods are discussed as well as when each method is appropriate. The techniques to identify and treat first and last records in the examples should be noted.

Introduction
Macro iteration is a convenient way to repeat a set of operations for each record of a data table. There exist two methods of setting loop control variables to facilitate the iteration.

- Datastep in conjunction with CALL SYMPUT.
- PROC SQL in conjunction with SELECT INTO.

There exist two algorithmic structures to perform the iteration.

- Perform a _NULL_ datastep to scan the data table using CALL SYMPUT to predefine all the relevant variables for every row. If there are 10 macro variables used per iteration, and there are a million rows in the iteration table, a total of 10 million variables would need to be predefined.

- Within the macro loop, subsequently select each record to dynamically define macro variables used for the current iteration only. PROC SQL SELECT INTO is most suitably used for this algorithmic structure, which is the subject of focus in this paper.

This paper presents the combination of PROC SQL with dynamic macro variable assignment to achieve iterative repetitive tasks. This technique is advantageous when the number of table variables extracted for use is huge as well as when macro variables are constructed conditionally and hence is inconvenient to be predefined. It allows intricate conditional macro processing and allows dependence on values of fields of past or current iteration.

Depending on the situation and requirements, there are two ways to treat the iterator dataset. Namely, invasive treatment and non-invasive treatment. The restriction, rather than choice, on using either implementation rests solely on the means of identity of each record of the data set.

Means of identity is the term used to refer to the mechanism employed to uniquely identify a record of a data set. Henceforth, the term data set, as opposed to dataset, would mean a combination of a number of SAS or RDBMS tables accessible by PROC SQL. Whereas, a record would mean a single row or combination of rows within such a combination of tables.

A data set can be used non-invasively when changes need NOT be made for records to be uniquely identified. Otherwise, an invasive modification of the data set has to be made. The most common and convenient invasive means of implementation is the introduction of a new table which contains a column N to make explicit the implicit datastep row identity variable _N_.

This paper employs the functional format to describe table schema,

Table-Name{
[Identity key comma-list],
comma-list of other table columns).

Identity key is the key employed to uniquely identify a record.

Non-Invasive Algorithm: Integer Iterator Key
Consider the simple case of a single table as basis of iteration.

```sas
%MACRO MergeEmpInfo_v0();
%LOCAL KeyN;
%LET KeyN = 0;
%DO %WHILE(&KeyN NE .);
%GetNextIterValue_v0(&KeyN);
%IF (&KeyN NE .) %THEN
%GetValuesAndDoStuffs_v0(&KeyN);
%END;
%MEND MergeEmpInfo_v0;

%MACRO GetNextIterValue_v0(KeyN);
/* Note 1.1 */
PROC SQL NOPRINT;
SELECT min(EmpNum)
INTO :KeyN
FROM Employees
WHERE EmpNum > &KeyN;
%MEND GetNextIterValue_v0;
```

Note 1.1: Identifying the next record is achieved by selecting the
smallest value of the set of all iterator keys larger than the current key.

%MACRO GetValuesAndDoStuffs_v0(KeyN);
/* Note 1.2 */
LOCAL Boss Title Level Dept Region;
SELECT Boss, Title, Level, Dept, Region
INTO :Boss, :Title, :Level, :Dept, :Region
FROM Employees
WHERE EmpNum = &KeyN;
%DoSomeStuffs(
 &Boss, &Title, &Level, &Dept, &Region);
%MEND GetValuesAndDoStuffs;

Note 1.2: Once the next iterator key identity is obtained, use it to extract variables of the current record relevant to the current endeavour.

However, the implementation just suggested is flawed as the last loop will get stuck endlessly repeating on the last iterator key. The reason is, when there are no more identity keys larger than the current identity, SELECT INTO would fail and not modify the value of KeyN. Therefore the following is the remedy. Which compares that the new key value is different from the previous loop to check for end of iteration.

%MACRO MergeEmpInfo_Int();
%LOCAL KeyN KeyNLast;
%LET KeyN = 0;
%LET KeyNLast = -1;
/* Test for end of iteration */
%DO %WHILE(&KeyN > &KeyNLast);
%let KeyNLast = &KeyN;
%GetNextIterValue(&KeyN);
/* Do only if not end of iteration */
%IF (&KeyN > &KeyNLast) %THEN
%GetValuesAndDoStuffs(&KeyN);
%END;
%MEND MergeEmpInfo_Int;

Non-Invasive Algorithm:
Date Iterator Key

Imagine a ridiculously hypothetical case where the employee id is the date of hire. The following are the modifications to be made to the SELECT INTO and clause in macro GetNextIterValue.

%MACRO GetNextIterValue_Alpha(KeyN);
PROC SQL NOPRINT;
SELECT min(EmpNum) format best.
INTO :KeyN
FROM Employees
WHERE EmpNum > "&KeyN";
%MEND GetNextIterValue_Alpha;

%MACRO GetValuesAndDoStuffs_Alpha(KeyN);
%LOCAL Boss Title Level Dept Region;
SELECT
Boss, Title, Level, Dept, Region
INTO :Boss, :Title, :Level, :Dept, :Region
FROM Employees
WHERE EmpNum = "&KeyN";
%DoSomeStuffs(
 &Boss, &Title, &Level, &Dept, &Region);
QUIT;
%MEND GetValuesAndDoStuffs_Alpha;

Non-Invasive Algorithm:
High Precision Real Number or Datetime Iterator Key

The reason for having to address this issue is a macro variable’s actually being a character literal rather than a variable and its inherent inability to store real values precisely. The accomplice to the issue is the legendary aliasing between binary numbers and the real values being emulated.

Preemption towards having to tackle this issue is by having the iterator key rounded to the appropriate precision before storing into the table. Such preemption would have been invasive by destroying the precision of table values.

While retaining macro MergeEmpInfo_Int, in lieu of such preemption, the respective SELECT and WHERE clauses should be thus.

Non-Invasive Algorithm:
Alphanumeric Iterator Key

The algorithm is similar to the case of Integer Iterator Key, but occurrences of unquoted &KeyN are modified to quoted "&KeyN".

/*MAIN*/
%MACRO MergeEmpInfo_Alpha();
%LOCAL KeyN KeyNLast;
/* [sp] = the first visible ASCII char */
%LET KeyNLast = ;
/* The next visible char thereafter */
%LET KeyN = '!
/* Test for end of iteration */
%DO %WHILE(&KeyN > &KeyNLast);
%let KeyNLast = &KeyN;
%GetNextIterValue(&KeyN);
/* Do only if not end of iteration */
%IF (&KeyN > &KeyNLast) %THEN
%GetValuesAndDoStuffs(&KeyN);
%END;
%MEND MergeEmpInfo_Alpha;
In *GetNextIterValue*:

```sql
SELECT round(min(EmpNum),.01) 
    format best15.2 
WHERE round(EmpNum,.01) > &KeyN
```

And in *GetValuesAndDoStuffs*:

```sql
WHERE round(EmpNum,.01) = &KeyN
```

High precision real values and datetime values have the same numeric storage format. Therefore the SELECT and WHERE clauses could, alternatively and respectively be,

```sql
SELECT min(EmpNum) format datetime23.3. 
WHERE EmpNum > "&KeyN"dt
WHERE EmpNum = "&KeyN"dt
```

**Non-Invasive Algorithm: Composite Iterator Key, Single Loop**

Whereas the following feature, which is currently absent in PROC SQL, would have been useful:

```sql
UPDATE table1 
USING table1 t1, table t2 
SET t1.value = t2.value 
WHERE t1.Parameter = t2.Parameter 
AND t1.ProductId = t2.ProductId 
AND t1.InspectDate = t2.InspectDate 
AND t1.value is missing;
```

PROC SQL does not provide for such correlated update/delete. While SAS Institute has yet to accede to our clamouring for this feature, SQL die-hards might be tempted towards,

```sql
CREATE TABLE table3 AS 
SELECT * FROM table1, table2 
WHERE t1.Parameter = t2.Parameter 
AND t1.ProductId = t2.ProductId 
AND t1.InspectDate = t2.InspectDate 
AND t1.TubeId = t2.TubeId 
AND t1.value is missing;
```

UPDATE table1 t1 
SET value = ( 
    SELECT value FROM table3 
    WHERE t1.Parameter = t3.Parameter 
    AND t1.ProductId = t3.ProductId 
    AND t1.InspectDate = t3.InspectDate 
    AND t1.TubeId = t3.TubeId 
) 
WHERE EXISTS ( 
    SELECT * FROM table3 
    WHERE t1.Parameter = t3.Parameter 
    AND t1.ProductId = t3.ProductId 
    AND t1.InspectDate = t3.InspectDate 
    AND t1.TubeId = t3.TubeId 
) 
AND t1.value is missing;
```

Which is unacceptable because using WHERE EXISTS on a large table join is tantamount to locking the tables and perhaps, even the system, from being usable for the rest of the day.

Therefore, there are at least four techniques to update a table with values from another table:

- datastep UPDATE
- datastep CALL EXECUTE set on the transaction table
- PROC SQL UPDATE ... WHERE EXISTS
- PROC SQL UPDATE embedded in a macro looping around the transaction table.

The solution for this is using good ol' macro iteration. In this illustration, we wish to update a base table by iterating on this so-called transaction table by using the last technique.

```sql
DataPoints( 
    [ProductId,InspectDate,TubeId,ParameterId], 
    ParameterName, CharValue1, CharValue2, 
    RealValue, Scale, Units).
```

However, we need to concatenate the composite key values to achieve a single loop iteration.

```
/*MAIN*/
%MACRO UpdateParam_single();
%LOCAL KeyN KeyNLast;
%LET KeyNLast = ;
%LET KeyN = '!
%DO %WHILE(&KeyN > &KeyNLast);
%let KeyNLast = &KeyN;
%GetNextIterValue_Combo(&KeyN);
%IF (&KeyN > &KeyNLast) %THEN
%GetValuesAndUpdate_Combo(&KeyN);
%END;
QUIT;
%MEND UpdateParam_single;
```

```sql
%MACRO IterKey();
trim(ProductId) || 
   btrim(put(InspectDate,best.)) || 
   btrim(put(TubeId,best.)) || 
   btrim(put(ParameterId,best.))
%MEND;
```

```sql
%MACRO GetNextIterValue_Combo(KeyN);
PROC SQL NOPRINT;
SELECT min(%IterKey) 
INTO :KeyN 
FROM DataPoints 
WHERE %IterKey > 
%MEND GetNextIterValue_Combo;
```

```sql
%MACRO GetValuesAndUpdate_Combo(KeyN);
%LOCAL CharValue RealValue;
SELECT 
   btrim(CharValue1)|| 
   btrim(CharValue2), RealValue 
INTO :CharValue, :RealValue 
FROM DataPoints t2 
WHERE %IterKey = 
%MEND GetValuesAndUpdate_Combo;
```

```sql
%MACRO GetValuesAndUpdate_Combo(KeyN);
%LOCAL CharValue RealValue;
SELECT 
   btrim(CharValue), RealValue 
FROM DataPoints t2 
WHERE %IterKey = 
%MEND GetValuesAndUpdate_Combo;
```

```sql
UPDATE ImcompleteDataPoints t1 
SET 
   CharValue = 
   RealValue = 
   WHERE %IterKey = 
   AND t1.value is missing;
%MEND GetValuesAndUpdate_Combo;
```
Non-Invasive Algorithm:
Composite Iterator Key, Nested Loop

The single loop iteration of composite keys is inefficient and inconvenient. Using nested loops to separate each member of the key into iteration levels should be a more convenient approach.

In the following macro NestedIteration is called recursively from KeyName3 through KeyName1. When it reaches KeyName1, then UpdateParamNow is called to perform the final task. It is possible to code NestedIteration in a way which would allow it to take any number of composite key members and perform any final task – a temptation avoided here so as not to complicate the illustration any further.

```sas
/*RECURSIVE MAIN*/
%MACRO NestedIteration(  
  DataPoints, NumKeys,  
  KeyName1, KeyName2, KeyName3);  
  %LOCAL KeyN KeyNLast KeyName;  
  %LET KeyNLast = ;  
  %LET KeyN = '1';  
  %LET KeyName = &&KeyName&NumKeys;  
  %DO %WHILE(&KeyN > &KeyNLast);  
    %let KeyNLast = &KeyN;  
    %GetNextIterValue_Nested(&KeyN);  
    %IF (&KeyN > &KeyNLast) %THEN  
      %DoIfMoreKeys;  
  %END;  
  QUIT;  
%MEND NestedIteration;  

/* Presuming all keys are character type*/
%MACRO GetNextIterValue_Nested(  
  KeyN, DataPoints);  
  SELECT min(&KeyName)  
  INTO :KeyN  
  FROM &DataPoints  
  WHERE &KeyName > "$KeyN";  
%MEND GetNextIterValue_Nested;  

%MACRO DoIfMoreKeys();  
  CREATE VIEW _&KeyName1 AS  
  SELECT * FROM &DataPoints  
  WHERE &KeyName = "$KeyN";  
%MEND DoIfMoreKeys;  

Invasive Algorithm:
Derived Integer Iterator Key, Single Loop

Having to track multiple loops can be hazardous. The treatment gets very complicated when a composite iterator key has a mix of data types. The solution is creating a new table with a single explicit sequential integer key, each of which rows contains all the iteration parameters to allow single loop iteration without the inefficiency found in having to concatenate an iteration key or differentiate their data types.

/*MAIN*/
%MACRO UpdateParamSingleInvasive(  
  TransactionTable,  
  KeyName1, KeyName2, KeyName3, KeyName4);  
  %LOCAL N NLOOPS;  
  %CreateIteratorTable(  
    &TransactionTable, IterationTable);  
  %DO N=1 %TO &NLOOPS;  
    %GetNextIterValue_Invas(&N, IterationTable);  
    %UpdateWithKeyN(&N, IteratorTable);  
  %END;  
  QUIT;  
%MEND;  

%MACRO CreateIteratorTable(  
  DataPoints, IterationTable);  
  PROC SORT DATA=&DataPoints OUT=&DataPoints.N;  
  BY &KeyName1 &KeyName2 &KeyName3 &KeyName4;  
  DATA IterationTable;  
  set DataPointsN END=EOL;  
  N = _N_;  
  /* Note 6.1 */  
  RUN;  
%MEND CreateIteratorTable;

Note 6.1: Define any table variables here that would be of convenience at each iteration.
The invasive single loop algorithm is very flexible and adaptable. It has only one iterator key, which is a contiguous sequential integer. Though using invasive iteration is the solution for cases where duplicate keys are present in a data set, it may be a more convenient tool, rather than using non-invasive algorithm for complex combinations of composite keys. It also absolves the need to treat the phenomenon of aliasing between decimal iterator key values and their stored binary values when real numbers are part of the iterator key. Such aliasing may cause the iteration to be completed without accomplishing anything because rows could not be identified.

Whereas, datastep UPDATE could be more efficient, these examples use correlated update as convenient illustration. The techniques may be used as a template for more complex iteration needs.

Contacting the Author
Author’s Name  Syloke J Soong
Address:  Allegro MicroSystems
115 Northeast Cutoff
Worcester, MA 01615
Email: ssoong@allegromicro.com