Paper 1277-2014

Adding Serial Numbers to SQL Data

Howard Schreier

ABSTRACT

Structured Query Language (SQL) does not recognize the concept of row order. Instead, query results are thought of as unordered sets of rows. Most workarounds involve including serial numbers, which can then be compared or subtracted. This presentation illustrates and compares five techniques for creating serial numbers.

INTRODUCTION

In DATA step programming, it is sometimes possible, and useful, to rely on the implicit ordering of observations. For example, suppose you have income and expense data for four time periods in a data set IncExp (see Figure 1).

Income	Expense
64	18
134	47
101	34
92	22

Figure 1. Data Set IncExp

The task at hand is to compute and display a running (cumulative) account of net income (income minus expense). There are numerous ways of doing this, but the sum statement is perhaps the most convenient. Here is the code.

```
data to_date ;
set IncExp ;
Net_Sum + ( income - expense ) ;
run ;
```

See Figure 2 for the result.

Income	Expense	Net_Sum
64	18	46
134	47	133
101	34	200
92	22	270

Figure 2. Data Set TO_DATE

This succeeds because the sum statement's target variable (NET_SUM) is automatically RETAINed and SAS® does its processing in the order in which the observations are stored in the input data set (IncExp).

PROC SQL has no similar features and cannot be made to operate in accordance with the implicit order of the rows. So, it is not possible for PROC SQL to derive this result.

Note: Other (that is, non-SAS) implementations of SQL may have addressable, manipulable cursors that can offer other possibilities, but this paper is concerned only with the native PROC SQL environment.

Here is another example of processing that is dependent on implicit order. Start with the two corresponding data sets ST_NAMES (Figure 3) and ST_CAPS (Figure 4).

StateName
Maryland
Virginia
North Carolina

Figure 3. Data set ST_NAMES

StateCap
Annapolis
Richmond
Raleigh

Figure 4. Data Set ST_CAPS

Next, produce a single data set with variables for corresponding state and capital city names. It's pretty simple to do with the DATA step.

```
data states ;
merge st_names st_caps ;
run ;
```

Figure 5 presents the result (STATES):

StateName	StateCap
Maryland	Annapolis
Virginia	Richmond
North Carolina	Raleigh

Figure 5. Data Set STATES

Turning to SQL, we are again at a loss; replicating the DATA step output is impossible.

However, if the given data sets are extended to include columns of serial numbers, SQL workarounds become possible. By serial numbers we mean sequential integers starting with 1 and incremented by 1 from row to row. Call the extended data sets IncExp_SR (Figure 6), ST_NAMES_SR (Figure 7), and ST_CAPS_SR (Figure 8).

Serial	Income	Expense
1	64	18
2	134	47
3	101	34
4	92	22

Figure 6. Data Set IncExp _SR

Serial	StateName
1	Maryland
2	Virginia
3	North Carolina

Figure 7. Data Set ST_NAMES_SR

Serial	StateCap
1	Annapolis
2	Richmond
3	Raleigh

Figure 8. Data Set ST_CAPS_SR

Do not be concerned with **how** the serial numbers are constructed. That is the main topic of this paper and will be addressed at considerable length later.

The PROC SQL solution for the running net income problem is

```
proc sql ;
select distinct ref.* ,
                sum(addup.income
                  - addup.expense) as Net_Sum
 from IncExp_sr as ref
      cross join
      IncExp_sr as addup
 where ref.serial GE addup.serial
 group by ref.serial ;
```

Figure 9 displays the result.

Serial	Income	Expense	Net_Sum
1	64	18	46
2	134	47	133
3	101	34	200
4	92	22	270

Figure 9. SQL Result Set (Running Net Income Problem)

Notice that the input data set (IncExp_SR) is joined with itself; each instance is given its own alias. That in turn allows the critical WHERE clause to operate.

Turning to the state/capital name task, the SQL code is

```
select st_names_sr.* , statecap
 from st_names_sr join st_caps_sr
  on st_names_sr.serial EQ st_caps_sr.serial
 order by serial ;
quit ;
```

The output is presented in Figure 10.

```
  Serial  StateName              StateCap
-------------------------------------------
       1  Maryland               Annapolis
       2  Virginia               Richmond
       3  North Carolina         Raleigh
```

Figure 10. SQL Result Set (State/Capital Name Task)

It should now be obvious that having serial numbers incorporated in tables that are going to be processed by PROC SQL can be advantageous. That's certainly something to consider looking forward in data design. But what if you are in a situation where serial numbers are lacking and your task is to remedy that? The DATA step can do the job. The code to expand the income-expense data would be something like this.

```
data reference ;
Serial + 1 ;
set IncExp ;
run ;
```

But this is a paper about SQL, so we'll concentrate on techniques which themselves utilize SQL or at least operate **within** an SQL session (that is, between a PROC SQL statement and the corresponding QUIT statement).

SQL Methods for Generating Serial Numbers

In the introduction we saw that the inclusion of a serial number column in a table enables us to develop SQL solutions that would otherwise be impossible. However, it does not show **how** such serial numbers can be generated. Now we'll see five different techniques to accomplish that task.

(A) The (Undocumented) MONOTONIC Function

This is rather clearly the simplest of the five solutions offered. The MONOTONIC function returns successive integers (1,2,3,…) from successive calls. So the solution for the row-numbering problem is

```
proc sql ;
create table WayA as
 select monotonic() as Serial, *
  from IncExp ;
quit;
```

The catch is that the MONOTONIC function is not documented. Organizations may have policies prohibiting the use of undocumented features. Even in the absence of such a policy, a coder may be reluctant to rely on the function.

In developing examples to illustrate the usage of the MONOTONIC function, the author encountered some surprising and disconcerting results. In some cases, series returned by MONOTONIC began **not** with 1 (one), but rather with higher integers. Unfortunately, it seemed not possible to produce such results consistently, or to experimentally determine just what circumstances cause the surprising and suspicious offsets.

Here is code that **simulates** the issue. Begin with a tiny data set.

```
data demo ;
do CharVar = 'a', 'b', 'c', 'd' ;
   output ;
   end ;
run ;
```

Then use the MONOTONIC function to generate serial numbers that begin with 3 (three).

```
proc sql ;
create table offset as
 select monotonic() + 2 as Suspect , * from demo ;
```

The output is shown in Figure 11.

Suspect	CharVar
3	a
4	b
5	c
6	d

Figure 11. Table OFFSET

Now imagine getting the same results **in the absence** of the additive constant ("+ 2").

There are ways to insulate calculations from the effects of such irregular behavior, but first consider that the offset may **not** be a problem; it depends on how the serial numbers will be used. If used in an ORDERED BY clause to preserve the initial ordering of rows, the offset does not change anything and can therefore be tolerated. In a self-join (like the one in the solution of the income-expense example above), where a table is joined to itself, the offset of course occurs on both sides, so the outcome is unaffected. But if serial numbers are used to implement a join where keys are lacking, as in the example involving state and capital city names, the situation is very different; if the serial numbers are offset by different amounts, the results will be utterly wrong.

The cure is to transform the serial numbers so that they **reliably** begin with 1 (one). Here is the code.

```
create table fixed as
 select suspect - min(suspect) + 1 as Serial , *
  from offset ;
```

The output is seen in Figure 12.

Serial	Suspect	CharVar
1	3	a
2	4	b
3	5	c
4	6	d

Figure 12. Table FIXED

The logic is simple. For example, if there is an undesirable offset of 2 (two) in the serial number vector, the MIN summary function will always return 3 (three), so that in the first row 3 maps to 3-3+1=1, in the second row 4 maps to 4-3+1=2, and the serial numbers progress from there.

It's also possible (1) to call on the DROP data set option to eliminate the offset series (SUSPECT) and (2) to wrap two statements into one via an inline view. Here is the resulting code.

```
create table fixed2(drop = suspect) as
 select suspect - min(suspect) + 1 as Serial , *
  from ( select monotonic() + 2 as suspect , *
           from demo ) ;
quit ;
```

See the result set in Figure 13.

Serial	CharVar
1	a
2	b
3	c
4	d

Figure 13. Table FIXED2

Keep in mind that the arbitrary addition of 2 is employed to simulate an **unexpected** offset.

(B) The DATA Step View

To start, consider this straightforward DATA step.

```
data increment / view=increment ;
Serial + 1 ;
set IncExp ;
run ;
```

It's identical to code presented above, except that it produces a view rather than a table. Also, it's not an SQL solution. But if the input data set (IncExp) is available before the SQL session begins, the view can be defined before PROC SQL is launched, then referenced within the SQL session.

```
proc sql ;
create table WayB as
 select *
  from increment ;
drop view increment ;
```

One can at least argue that this is an SQL solution. The DROP VIEW statement is there for tidiness, presuming no subsequent need for the view.

(C) The Output Delivery System (ODS)

PROC SQL has an option, called NUMBER, which presents a serial number at the start of each row of output. Here is example code.

```
reset number ;
select *
 from IncExp. ;
reset nonumber ;
```

Figure 14 gives us the result set.

```
   Row     Income  Expense
------------------------
     1         64        18
     2        134        47
     3        101        34
     4         92        22
```

Figure 14. Result Set for Query with NUMBER Option in Effect

This is output in the sense of the Output Delivery System (ODS). These serial numbers are **not** produced via a CREATE statement; in that respect this method is unlike the other four being presented. So the NUMBER option here succeeds in generating serial numbers, but not in embedding them where we need them, in tables or views.

Fortunately, ODS has an OUTPUT destination that records results of various procedures as tables (that is, SAS data sets). SQL is one of the supported procedures. This makes it possible to solve the problem by wrapping the oppropriate ODS code around the query. For example, consider this code.

```
reset number ;
ods output SQL_Results=WayC( rename=(row=Serial) ) ;
select * from IncExp ;
ods output close ;
reset nonumber ;
quit ;
```

The two RESET statements toggle the state of the NUMBER option (presumably we want RESET in effect just for the duration of the SELECT statement). The ODS statement preceding the SELECT redirects the output from the LISTING destination (presumably the default) to the OUTPUT destination, and the ODS statement after the SELECT statement reverses that process.

(D) The FCMP Procedure

The FCMP Procedure enables one to use a variant of the DATA step language to create SAS functions. This provides a workaround for the issue of absent documentation of the MONOTONIC function: Use **documented** tools to create a similar but user-specified function.

We call the user-defined function INCREMENT and declare one character argument, MV. That argument is used to pass in the **name** of a macro variable that can then be used to hold the value of one serial number and make it available for the calculation of the next. It's necessary because the memory space associated with the function does **not** persist from one function call to the next.

Here is the function-defining code.

```
proc fcmp outlib=work.funcs.sgf2014 ;
function increment        (mv $) ;
   integer = input( symget(mv),16.) + 1 ;
   call             symput(mv,put(integer,z16.) ) ;
   return(integer) ;
   endsub ;
run ;
options cmplib=work.funcs;
```

The OUTLIB specification controls the storage location for the function definition. In this example a temporary location within the WORK library is chosen, but it's also possible to use a permanent location. The CMPLIB system option points to this location.

INTEGER is a user-chosen variable name. It is local to this particular PROC FCMP use, so you need not worry about name conflicts with other FCMP invocations or with the calling environment. Such naming insulation also applies to the local name(s) of the arguments(s), MV in this case.

When the function is called, the assignment statement for INTEGER first uses the SYMGET function to retrieve the previous value of the serial number from the macro variable identified by the argument MV. Then the INPUT function is employed to convert the character value to a numeric one. Finally, the addition operator (+) does the incremenation.

When control passes to the CALL statement, the process is essentially reversed. The PUT function converts INTEGER's numeric value to type character (zero-filled), and the SYMPUT routine stores it back into the macro variable, ready for use in the next function call. Notice that there is no macro variable resolution in this process. The symbol table (storage location for macro variables) is merely a convenient place to carry forward the serial numbers, and SYMGET and SYMPUT are the tools that manage the traffic.

Now we can apply the user-defined function to the problem at hand.

```
proc sql;
%let mvD = 0 ;
create table WayD as
 select increment('mvD') as Serial, *
  from IncExp ;
%symdel mvD ;
```

Notice the code surrounding the CREATE statement. The %LET statement is critical. It creates the macro variable and initializes it to 0 (zero) so that the serial numbers begin with 1 (one). Also note that the name of the macro variable has to be passed, as a character value, to the function. The %SYMDEL statement is there just for housekeeping.

(E) The RESOLVE Function

We now come to the last, and most intricate, of the techniques for generating row serial numbers. Take a look at this code.

```
%let mvE = 0 ;
create table WayE as
 select
   input(resolve('%let mvE=%eval(&mvE+1); &mvE'),32.)
   as Serial, *
  from IncExp ;
%symdel mvE ; ;
quit ;
```

Notice that this approach, like the one built with PROC FCMP, requires the declaration of a macro variable. That is done with the %LET statement. The macro variable then appears several times in the RESOLVE function reference, which in turn is an argument to the INPUT function that defines the first column of the SELECT query. However, because the RESOLVE argument is contained in **single** quotes, the processing is deferred until runtime. In other words, that long quoted string is seen as nothing but a sequence of characters until PROC SQL is actually processing (reading data and performing evaluations).

To explain how this code works, we analyze the evaluation of the expression for the serial number. That code begins with function name INPUT and ends with the closing parenthesis after the informat "32.". That is, we want to evaluate

```
input(resolve('%let mvE=%eval(&mvE+1); &mvE'),32.)
```

Starting from the first row of the source table and analyzing the quoted string from the inside out, we first have

```
%eval(&mvE+1)
```

The macro variable mvE was pre-initialized to 0 (zero), so the macro function %eval performs the arithmetic 0+1=1. Thus the expression is simplified to

```
input(resolve('%let mvE=1; &mvE'),32.)
```

Next, the %LET statement operates, thereby incrementing the serial number that the target macro variable (mvE) holds, from 0 to 1. However, that in itself does nothing to feed the serial number to the SQL expression that populates the first column of the SQL query. Setting aside the %LET statement (its work being done) and turning to the newly populated macro variable referenced to the right of the semicolon, we have

```
input(resolve('1'),32.)
```

This simplifies to

```
input('1',32.)
```

Finally, the INPUT function operates to convert the character type intermediate result to a numeric. The informat "32." is the maximum allowed, much bigger than necessary, but harmlessly so. When the second row is processed, the macro variable mvE contains a value of 1 (one), so the incrementation raises it to 2 (two), which populates the second row's serial number. The process continues, one row at a time.

As with the FCMP technique, the macro variable has to be initialized outside the CREATE statement, and the %SYMDEL statement is available for cleanup, if appropriate.

COMPARISONS

Notice that the tables generated by the five SQL techniques have not been presented. To verify that all five techniques produce the same correct results, we run the COMPARE Procedure five times, with the DATA step solution (REFERENCE) serving as benchmark. The code is

```
proc compare data=reference compare=WayA ; run ;
proc compare data=reference compare=WayB ; run ;
proc compare data=reference compare=WayC ; run ;
proc compare data=reference compare=WayD ; run ;
proc compare data=reference compare=WayE ; run ;
```

In each PROC COMPARE output there is the same note, shown in Figure 15.

```
NOTE: No unequal values were found.
All values compared are exactly equal.
```

Figure 15. PROC COMPARE Outcome

However, there are some minor differences at the metadata level for the method (C), which uses the SQL NUMBER option and the ODS OUTPUT destination. To see that, we look at a couple of clips taken from a PROC CONTENTS report run against the table (WayC) created with ODS. In Figure 16 we see that ODS inserts the very generic data set label "Query Results", whereas the other four output data sets, all generated by the CREATE TABLE statements of PROC SQL, have no data set labels.

```
Label                    Query Results
```

Figure 16. Data Set Label (ODS Output Destination)

In Figure 17 we see the variable names and attributes.

```
Alphabetic List of Variables and Attributes

  #    Variable    Type    Len    Format

  3    Expense     Num       8    BEST8.3
  2    Income      Num       8    BEST8.3
  1    Serial      Num       8    F6.
```

Figure 17. Formats and Other Attributes (ODS Output Destination)

Notice the formats, "F6." for the column generated by the NUMBER option and "BEST8.3" for other numeric variables. Both are a bit odd. "F6." is simply an alias for "6.". "BEST8.3" is a bit more strange. The ".3" fragment would indicate that there should be three places to the right of the decimal point, but "BEST" directs SAS to use an internal algorithm to determine the number of decimal places. They cannot **both** be in effect, and in fact SAS ignores the ".3" and treats "BEST8.3" as "BEST8."

USAGE WITH GROUP BY CLAUSES

Things become a bit more complex when a GROUP BY clause is involved. Figure 18 presents an expanded version of the income-expense data set; call it IncExp_ID.

ID	Income	Expense
B	84	88
B	120	77
B	98	54
A	64	18
A	134	47
A	101	34
A	92	22
C	0	0

Figure 18. Data Set INC_EXP_ID

Employing (for variety) the user-defined function INCREMENT (created above with PROC FCMP), we can generate serial numbers. Here is the code.

```
proc sql ;
%let carry = 0 ;
create table IncExp_ID_sr as
 select increment('carry') as Serial , *
  from IncExp_ID ;
%symdel carry ;
```

Figure 19 reflects the outcome, which we call IncExp_ID_sr.

Serial	ID	Income	Expense
1	B	84	88
2	B	120	77
3	B	98	54
4	A	64	18
5	A	134	47
6	A	101	34
7	A	92	22
8	C	0	0

Figure 19. Table Inc_Exp_ID_sr

From there we can move to a solution of this more general form of the income-expense problem.

```
select distinct ref.* ,
                sum(addup.income
                  - addup.expense) as Net_Sum
 from IncExp_ID_sr as ref
      join
      IncExp_ID_sr as addup
  on ref.id EQ addup.id
 where ref.serial GE addup.Serial
 group by ref.id , ref.Serial
 order by serial ;
```

The result set is shown in Figure 20.

Serial	ID	Income	Expense	Net_Sum
1	B	84	88	-4
2	B	120	77	39
3	B	98	54	83
4	A	64	18	46
5	A	134	47	133
6	A	101	34	200
7	A	92	22	270
8	C	0	0	0

Figure 20. Result Set (Income-Expense Sample Problem with GROUP BY)

PRODUCING VIEWS

Four of the five techniques can be made to yield views rather than tables. The exception is the ODS approach, which of course does not utilize a CREATE statement. Each of the other methods should be able to produce a view via the simple act of changing "table" to "view" in the CREATE statement. However, there are issues to consider. Since views do their computations on the fly, a view employing the MONOTONIC function (Method A) would not be able to insulate its results from the "offset" issue. The DATA step view approach could work with a PROC SQL view, but why build two views to do the same thing? In the cases of Method D (PROC FCMP) and Method E (PROC RESOLVE), the macro variables supporting them would have to be maintained as long as the views are maintained, and of course reinitialized before each reference.

PERFORMANCE

Just because the five derivations of serial numbers produce the same results does not mean that all five use the same amount of time in doing so. To investigate the efficiency, a test table of one million rows was processed six times to generate serial numbers. The first run used the pure DATA step method, to serve as a benchmark. The other five employed the five SQL techniques. See Figure 21.

Technique	Real Time (Seconds)
DATA Step	0.8
(A) SQL / MONOTONIC	1.0
(B) SQL / DATA Step View	2.0
(C) SQL / NUMBER + ODS	1.5
(D) SQL / FCMP	5.8
(E) SQL / RESOLVE	44.5

Figure 21. Time Requirements

REFERENCES

Carpenter, Arthur. 2013. "Using PROC FCMP to the Fullest: Getting Started and Doing More." *Proceedings of the SAS Global Forum 2013 Conference*, Cary NC: SAS Institute. Available at http://support.sas.com/resources/papers/proceedings13/139-2013.pdf.

King, John [aka Data Null]. "How to Do It." [Use the Resolve function to generate serial numbers in PROC SQL]. 1 September 2009. Available at http://www.listserv.uga.edu/cgi-bin/wa?A2=ind0909a&L=sas-l&D=0&P=33264.

CONTACT INFORMATION

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

Howard Schreier
703-979-2720
hs AT howles DOT com
http://www.sascommunity.org/wiki/User:Howles

Contact other readers, and other people interested in the subject matter, at:

http://www.sascommunity.org/wiki/Adding_Serial_Numbers_to_SQL_Data

APPENDIX: CODE TO CREATE EXAMPLES

Run this code before running the code in the body of this paper.

* BUILD TABLES ;

```
data IncExp ;
input Income Expense ;
datalines ;
 64 18
134 47
101 34
 92 22
;
data IncExp_ID ;
input ID $ Income Expense @@ ;
datalines ;
B  84 88  B 120 77  B  98 54
A  64 18  A 134 47  A 101 34 A  92 22
C   0  0
;
data st_names ;
input StateName : & $20. ;
datalines ;
Maryland
Virginia
North Carolina
;
data st_caps ;
input StateCap : & $20. ;
datalines ;
Annapolis
Richmond
Raleigh
;
```

* ADD SERIAL NUMBERS ;

```
%macro rownumbers(ds=) ;
data &ds._sr ;
serial + 1 ;
set &ds. ;
run ;
%mend rownumbers ;

%rownumbers(ds=IncExp)
%rownumbers(ds=st_names)
%rownumbers(ds=st_caps)
```