Paper 5184-2020

Update to the Rescue

Robert Virgile, retired

ABSTRACT

The UPDATE statement provides a unique method of combining two SAS® data sets. That method is invaluable when needed; on the other hand, it is rarely needed. This paper explores a variety of ways to expand the usefulness of the UPDATE statement, making it the simplest solution to a broad variety of programming problems.

INTRODUCTION

This paper explores the UPDATE statement within a DATA step. (If you are familiar with the SQL UPDATE statement, that unfortunately has nothing to do with this paper.) A typical program looks like this:

```
data combined;
  update one two;
  by id;
run;
```

This DATA step follows typical rules:

- You can choose the name for the output data set.
- The input data sets refer to existing SAS data sets. The names of the data sets are not important, but whatever names appear (such as ONE and TWO) must already exist.
- The BY variable (or variables) refer to existing variables that are part of both incoming data sets. ID is not a key word or a reserved word, just a variable name.
- Both incoming data sets must be in sorted order to support the BY statement. As usual, if the data sets are already in order, you do not need to run PROC SORT.
- Mismatches can exist. A value of ID might appear in one data set but not the other.

What is special and different about UPDATE? Consider:

- The UPDATE statement requires exactly two incoming SAS data sets.
- The first data set (named ONE in this sample program) should contain at most one observation for each value of the BY variable(s). While it is possible to use a data set with more than one such observation, the results would be entirely useless.
- The second data set (named TWO in this sample program) can contain many observations for each value of the BY variable(s).
- UPDATE returns exactly one observation for each value of the BY variable(s), no matter how many observations exist in the incoming data sets.
- UPDATE combines the data sets by utilizing nonmissing values only from the second data set. For each ID, it starts with values from the ONE data set, then copies any nonmissing values from the TWO data set. It outputs an observation for each ID once all the changes have been applied.

Here is an example of the inputs and output for this typical program.

ONE contains:

ID	Height	Weight
Alice	.	130
Bob	72	180
Carol	66	125

TWO contains:

ID	Height	Weight
Alice	68	.
Alice	.	132
Carol	.	120

Table 1. Inputs to a Typical UPDATE

Based on ID, all nonmissing values from TWO replace the original values from ONE.

COMBINED contains:

ID	Height	Weight
Alice	68	132
Bob	72	180
Carol	66	120

Table 2. Output from a Typical UPDATE

Is this outcome useful? Once in a while. This paper explores how to embellish the sample program above, to make it more useful.

BEYOND THE BASICS

With a little ingenuity, UPDATE can apply to a broader array of problems. In the remainder of this paper, we explore some tougher scenarios, and how UPDATE comes to the rescue.

CASE #1: KEEP ALL NONMISSING VALUES

As usual, we begin with two SAS data sets: a master data set, and a set of changes to apply. While some mismatches might exist, both data sets contain a single observation per ID. Now here's the wrinkle. The master data set should have its missing values replaced, but its nonmissing values preserved. For example, the incoming data sets might be:

MASTER

ID	Height	Weight
Alice	.	130
Bob	72	180
Carol	66	125

CHANGES

ID	Height	Weight
Alice	68	.
Bob	.	132
Carol	.	120

Table 3. Preserve Nonmissing Values

Since MASTER contains just one missing value, that's the only value that should be replaced.

While we could apply this program, it falls short of the goal:

```
data combined;
  update master changes;
  by id;
run;
```

Such a program applies all the nonmissing values in CHANGES, generating:

ID	Height	Weight
Alice	68	130
Bob	72	132
Carol	66	120

Table 4. Failing to Preserve Nonmissing Values

However, this time we need to preserve any nonmissing values in the MASTER data set, obtaining:

ID	Height	Weight
Alice	68	130
Bob	72	180
Carol	66	125

Table 5. Required Results

How can we revise the UPDATE process:

- Use nonmissing values from the first data set, and
- Replace only the missing values in the first data set with nonmissing values from the second data set?

No tool does the job. Not UPDATE, not MERGE. But mildly clever programming uses UPDATE to get the job done:

```
data combined;
  update changes master;
  by id;
run;
```

Switching the data sets around lets UPDATE generate the right outcome. The revised program saves the available data from CHANGES, and then overwrites that data with the nonmissing values from MASTER.

CASE #2: COLLAPSING ROWS

This time, begin with a single data set. Inconveniently, its data values are spread across multiple observations:

MY_DATA

ID	Height	Weight
Alice	68	.
Alice	.	132
Bob	.	180
Bob	70	.
Carol	62	.
Carol	.	140

Table 6. Inputs Spread Over Multiple Observations

How do we collapse the rows to obtain each ID on a single observation?

COLLAPSED

ID	Height	Weight
Alice	68	132
Bob	70	180
Carol	62	140

Table 7. Collapsing Multiple Observations Into One

At first, this doesn't even look like a problem for UPDATE. After, there is only one data set, not two. Still, UPDATE comes to the rescue:

```
data collapsed;
  update my_data (obs=0) my_data;
  by id;
run;
```

The same set of variables appears in both the master and the transaction data sets. Yet the DATA step reads nothing from the master data set because of the OBS=0 data set option. For each ID, the DATA step assimilates all the nonmissing values, then outputs a single observation.

CASE #3: COMBINING MULTIPLE SOURCES

Suppose multiple sources of data contain conflicting information. Let's name the input data sources according to the quality of information they contain: GOOD, BETTER, and BEST. For a particular data point, the idea is to use data from BEST if it exists. Otherwise, use data from BETTER if it exists. As a last resort, use data from GOOD. To illustrate, the values in bold should be used:

GOOD: Use Values Unavailable Elsewhere

ID	Height	Weight
Amy	60	140
Bob	64	155
Dan	70	198
Eve	60	140
Eve	.	135
Joe	72	220
Lou	63	150

BETTER: Use values not in BEST

ID	Height	Weight
Amy	61	135
Bob	65	160
Dan	68	.
Eve	61	135
Irv	61	.
Joe	71	135

BEST: Use All Values

ID	Height	Weight
Amy	.	150
Bob	.	150
Eve	.	142
Irv	.	150
Joe	73	.
Lou	.	160
Lou	.	165

Table 8. Multiple Data Sources

As usual, UPDATE works with just two data sets. Begin by combining the three sources:

```
data all3;
  set good better best;
  by id;
run;
```

The order is important. Since UPDATE will come into play in a moment, the order must place the least important observations first for each ID (those from GOOD), and the most important observations last (those from BEST).

At this point, just apply UPDATE:

```
data final;
  update all3 (obs=0) all3;
  by id;
run;
```

The final nonmissing value for each data point will remain.

CASE #4: LOCF

LOCF is a method of replacing missing values. LOCF = "Last Observation Carried Forward". Here is an example of the before and after picture. For each ID, nonmissing values carry forward to the next observation in case the DATA step requires them as a replacement for missing values. The result should be:

BEFORE

ID	Recno	Amount
Amy	1	1234
Amy	2	.
Amy	3	2468
Bob	1	.
Bob	2	3456
Bob	3	.
Bob	4	.
Bob	5	4567
Bob	6	.

AFTER

ID	Recno	Amount
Amy	1	1234
Amy	2	1234
Amy	3	2468
Bob	1	.
Bob	2	3456
Bob	3	3456
Bob	4	3456
Bob	5	4567
Bob	6	4567

Table 9. LOCF Results

The values in bold carried forward from the prior observation for that ID.

When only one variable "carries forward", the problem does not require UPDATE:

```
data after;
  set before;
  by id;
  if first.id then replacement = amount;
  else if amount > . then replacement = amount;
  else amount = replacement;
  retain replacement;
  drop replacement;
run;
```

But suppose 20 variables should "carry forward". The program becomes much longer and messier. What about UPDATE? UPDATE can carry forward the nonmissing values, while ignoring missing values:

```
data after;
  update before (obs=0) before;
  by id;
run;
```

However, UPDATE fails here because it outputs only one observation per ID. Unless …

```
data after;
  update before (obs=0) before;
  by id;
  output;
run;
```

After all, this is a DATA step. So use DATA step tools such as the OUTPUT statement. Override the default actions of UPDATE (outputting a single observation per BY group), forcing the DATA step to output every observation.

CASE #5: LOCF, WITH AN EXPIRATION DATE

For a slightly more complex variation, apply LOCF but with an expiration date. Each observation can replace subsequent missing values, as long as the subsequent observation occurs no more than five days later. For example, consider that the DATE values represent true SAS dates, not character strings:

BEFORE

City	Humidity	Date
Boston	30	2020-01-05
Boston	20	2020-01-06
Boston	.	2020-01-08
Boston	.	2020-01-13
Boston	40	2020-01-14
Boston	.	2020-01-17
Boston	.	2020-01-18
Boston	.	2020-01-19

AFTER

City	Humidity	Date
Boston	30	2020-01-05
Boston	20	2020-01-06
Boston	20	2020-01-08
Boston	.	2020-01-13
Boston	40	2020-01-14
Boston	40	2020-01-17
Boston	40	2020-01-18
Boston	.	2020-01-19

Table 10. LOCF for Five Days Only

We are using previous HUMIDITY as an estimate to replace missing values for the current HUMIDITY. However, as time passes, the previous humidity becomes less and less relevant to the current estimate. So we invent a rule that previous humidity "expires" after five days, and should no longer be used beyond that point. Therefore, 20 remains available through 2020-01-10, and 40 remains available through 2020-01-18.

A moderately simple program can accomplish the task:

```
data after;
  set before;
  by city;
  if first.city then call missing(previous_humidity, previous_date);
  if humidity > . then do;
     previous_humidity = humidity;
     previous_date = date;
  end;
  else do;
     if date - previous_date < 5 then humidity = previous_humidity;
  end;
  retain previous_humidity previous_date;
  drop previous_humidity previous_date;
run;
```

Save the most recent HUMIDITY and DATE. When a missing HUMIDITY comes along, check to see whether the missing value occurs within five days. If so, use the retained recent HUMIDITY to replace the missing value.

This approach is good, perhaps better than UPDATE. However, what if 20 variables fall into the "replace if missing" category. Then this program becomes lengthy and cumbersome. Could UPDATE help here? What would the program look like?

Consider expanding the original data set, creating a much larger data set:

```
data projections;
  set before;
  if humidity > . then do date = date to date + 4;
     output;
  end;
run;
```

This data set contains all the values that might be used to update an observation:

City	Humidity	Date
Boston	30	2020-01-05
Boston	30	2020-01-06
Boston	30	2020-01-07
Boston	30	2020-01-08
Boston	30	2020-01-09
Boston	20	2020-01-06
Boston	20	2020-01-07
Boston	20	2020-01-08
Boston	20	2020-01-09
Boston	20	2020-01-10
Boston	40	2020-01-14
Boston	40	2020-01-15
Boston	40	2020-01-16
Boston	40	2020-01-17
Boston	40	2020-01-18

These five observations are all based on the second observation from the original data set BEFORE

Table 11. Expiring LOCF, Expand the Observations

While we can generate this data set, how would it be useful? How could UPDATE use it?

Let's rearrange it:

```
proc sort data=projections;
  by city date;
run;
```

Sorting rearranges set puts the data into a special order. For each DATE, the last observation is the one that should be used.

City	Humidity	Date	Original Date
Boston	30	2020-01-05	2020-01-05
Boston	30	2020-01-06	2020-01-05
Boston	20	2020-01-06	2020-01-06
Boston	30	2020-01-07	2020-01-05
Boston	20	2020-01-07	2020-01-06
Boston	30	2020-01-08	2020-01-05
Boston	20	2020-01-08	2020-01-06
Boston	30	2020-01-09	2010-01-05
Boston	20	2020-01-09	2010-01-06
Boston	20	2020-01-10	2010-01-06
Boston	40	2020-01-14	2010-01-14
Boston	40	2020-01-15	2010-01-14
Boston	40	2020-01-16	2010-01-14
Boston	40	2020-01-17	2010-01-14
Boston	40	2020-01-18	2010-01-14

Table 12. Sort the Expanded Observations

The ORIGINAL DATE column is NOT part of the data. But it indicates which original observation provided the HUMIDITY value. For each DATE, the last HUMIDITY is the most recent available value, and is therefore the proper replacement. All that remains:

```
data want;
  update before (in=original_date_list) projections;
  by city date;
  if original_date_list;
run;
```

Because expanding observations may have added new dates that never appeared in the data before, the program uses the IN= data set option to select only those dates that appeared in the original data set. Just as UPDATE can add the OBS= data set option, it can also utilize the IN= data set option.

CONCLUSION

With most programming tools, add some thought and ingenuity and you can expand the usefulness and applicability. The UPDATE statement is no exception.

CONTACT INFORMATION

Questions and comments are always welcome:

Robert Virgile

rvirgile@verizon.net