Paper 117-2007

Data Matching

David H. Johnson FRSA, DKV-J Consultancies, Moorabbin, Vic Australia
Wendy B. Dickinson, Ph.D., Ringling School of Art and Design, Sarasota, Fl

ABSTRACT

It is rare that a single data set or table will hold all the data we need to answer the questions facing our business. We usually need to match data from multiple sources to assemble the meaningful intelligence required for our SAS® experts to deliver to the business.

Indeed, SAS provides substantial capability for the matching of data, but all too often an inefficient or inaccurate method can be employed leading to incomplete or incorrect results.

In this workshop we will explore some data and code the correct syntax to provide the results we want.

This workshop deals with the data step and the process of bringing in multiple data sources. It needs to be clearly understood at all levels of SAS programming expertise, but the examples demonstrated will be of most benefit to people with less experience in SAS programming.

INTRODUCTION

In this workshop, we will demonstrate the application and use of three SAS statements: (I) Set, (II) Merge, and (III) Update. The purpose of these statements is to form new data from existing data, and we'll look at how we can apply these statements effectively, and identify some of the traps in using them incorrectly.

For each of these statement demonstrations, we will start with a scenario describing a business need, and then translate that need into a set of code. For the purposes of the workshop, most of the code has been written, and is available on the workshop computers.

However, the code is either incomplete, or incorrect, and the workshop will focus on identifying the outcome intended from the business request, and correcting or completing the code so it fulfils that purpose. Each set of corrected code should present a complete solution to the initial problem identified by the scenario.

The SAS code underlying the exercises is distributed through this paper. The paper indicates the changes needed to the code, but the fully corrected code is not provided so that the attendee has the opportunity to identify and code these changes personally. After all, the purpose of the Workshop is a learning exercise, and it is important that the SAS programmer develops those skills needed to verify an outcome matches a business requirement.

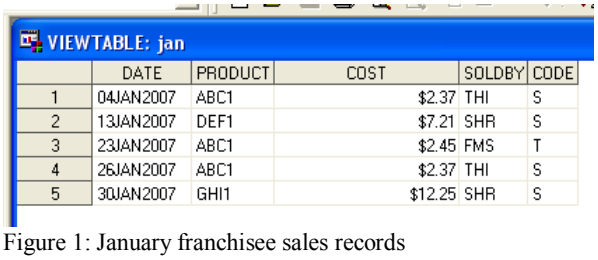
I. SET

EXAMPLE A. CONCATENATE TWO TABLES

SCENARIO

We have received two Excel spreadsheets from a franchisee for January and February. The franchise management team wants us to gather this data into a single table and produce a summary report. The Excel import code, and the summary reporting code are provided and are correct. The code that collates the data is not complete.

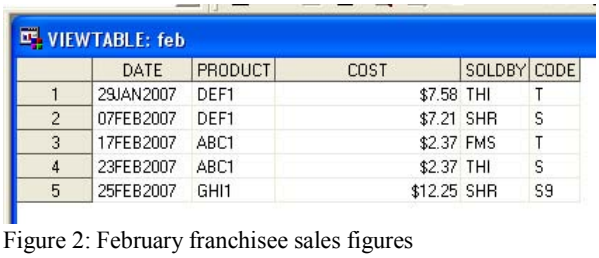
The Excel import code has created SAS tables called WORK.JAN and WORK.FEB. Here is a screen view of each of the two tables.



VIEWTABLE: jan

	DATE	PRODUCT	COST	SOLDBY	CODE
1	04JAN2007	ABC1	$2.37	THI	S
2	13JAN2007	DEF1	$7.21	SHR	S
3	23JAN2007	ABC1	$2.45	FMS	T
4	26JAN2007	ABC1	$2.37	THI	S
5	30JAN2007	GHI1	$12.25	SHR	S

Figure 1: January franchisee sales records



VIEWTABLE: feb

	DATE	PRODUCT	COST	SOLDBY	CODE
1	29JAN2007	DEF1	$7.58	THI	T
2	07FEB2007	DEF1	$7.21	SHR	S
3	17FEB2007	ABC1	$2.37	FMS	T
4	23FEB2007	ABC1	$2.37	THI	S
5	25FEB2007	GHI1	$12.25	SHR	S9

Figure 2: February franchisee sales figures

We need to create a SAS table called WORK.YTD so that we have a data source for our report process. Run the following code and look at the table produced. Why is this not correct?

```
Data YTD;
  Set JAN;
  Set FEB;
Run;
```

When the data is read, the data step reads the first record from the "JAN" table, and then reads the first record from the "FEB" table in the same iteration of the SAS data step. Since the two tables have columns of the same name, the following happens:

- The structure of the output table is defined at compile time to mirror the "JAN" table. This structure is part of the "Program Data Vector", or "PDV".
- As the data step iterates, the data retrieved from the "JAN" input table is overwritten by data read from the last named table: "FEB". Therefore, the data we see is from the "FEB" data.
- Since each table contains five rows, the data step iterates five times and we see five records written out. We actually wanted 10 records.
- To correct this, we need to specify a single "Set" statement that refers to both tables.
- This will cause the data step to iterate five times as it reads the "JAN" data, and a further five times as it reads the "FEB" data.
- Ten records will be written to YTD.

Correct the code so that ten records are created.

The use of more than one "Set" statement in a data step is dealt with later in this workshop.

EXAMPLE B. REPORT THE DATA SEQUENTIALLY

SCENARIO

We need data to go into our General Ledger from this source. The accounting software package we use expects that data will be imported in date order, and when we try to read the YTD table, our package complains that the data is not in order. Fix the data so the software package will read it.

As a simple cure, we could use the Sort procedure to re-order our data. However, the data was written within each spreadsheet in date order. The problem is that a backdated transaction was written to the "FEB" table with a January sale date.

If we "Set" the statement together and then sort it, the following happens:

- We use two steps to get the result.
- At one point in the sort we will have the YTD table we created, and a utility file containing all the data from the records from the YTD table, and a temporary table containing the sorted YTD data before this is used to replace the original YTD table. In other words, we are holding three times as much data as we need.
- It is simpler to read the data in the right order from both tables. This process is called interleaving, and we use a BY statement. This indicates to SAS that the YTD table should be built by selecting the data from each table in "DATE" order.

Add the appropriate "BY" statement to the original step.

With a ten record table it is hard to see the benefit in terms of space and time that interleaving gives us. However, the authors have been processing and summarising tables in the last year that contain over 200m records. This process takes a long time, and any coding efficiency that reduces this time has significant value.

Note that when we created the YTD table by interleaving the data, we effectively created a sorted table. However, SAS does not know the data is sorted and if we ask SAS to sort the data later on, it will do so. If we could tell SAS the data is sorted, then when we asked for a SORT to be performed, SAS would identify the data was already sorted and potentially save substantial processing time and space resources. We can identify the data is sorted if we modify our "Data" statement as follows:

```
Data YTD( SortedBy = DATE);
```

Here is the message produced when we try to sort the data (in the same way) again.

```
239   Proc Sort Data = YTD;
240     By DATE;
241   Run;

NOTE: Input data set is already sorted, no sorting done.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.03 seconds
```

While it may seem unnecessary to spend this time on a small temporary table, there is a viewpoint that you should write SAS code consistently. In this way, when we most need to have well-structured and documented data, consistent practices will ensure we have what we require. There is also an expectation that the temporary code we write now will need to be used more than once.

While in this process we are examining January and February's data, we may face the same issues again with March, April and so on. The code we write today will be available for us to use again, and a little care now will pay dividends in quicker and more reliable results later.

EXAMPLE C. DATA (PDV) DEFINITION

SCENARIO

The shop front is open six days a week, and the franchisee has identified one of the sales as possibly being fraudulent because it occurred when the shop was closed. The transaction has been identified with a code value "S9". Report all such flagged fraudulent transactions.

In our February data above, we can see there was a record with "S9" in the CODE value. We should be able to pick up this record and any other suspicious transaction by using the following print code.

```
259   Proc Print Data = YTD;
260     Where CODE Eq "S9";
```

```
261    Run;
```

The log note tells us that the record has not been found.

```
NOTE: No observations were selected from data set WORK.YTD.
```

What is wrong with this process? In the definition notes for our first example, we specified the following: *"The structure of the output table is defined at compile time to mirror the "JAN" table. This structure is referred to as the Program Data Vector, or PDV."* Our problem is two fold, and both problems have been seen on a number of occasions by the authors.

- "Code table" type data is badly structured, with a combination of one byte and two byte codes. The codes are poorly defined in that "S" codes are acceptable, but "S9" codes indicate problems. All it takes is the truncation of a code to turn it from a "problem" to "acceptable". A better structure would change the "S" code to a two byte value, and flag the exceptions with a different prefix such as "X" so that if the code is truncated, it is not misidentified.
- We have read data from an unstructured source, where SAS has defined the data structure based on what it finds in this source. Since the "JAN" data only had one-byte code values, this structure has propagated throughout our data. If we must use unstructured sources, then we should create a model for the data we expect, and use this model when we assemble the data.

There are two ways to model the data; the first is to use definition statements in our SAS data step. Before the SET statement has read any incoming data, we define the PDV to hold CODE as a two-byte string.
We can use a simple "Length" statement in this form to correct the problem.

```
Length CODE $2;
```

Include each of these options one at a time and observe the effect on the table, and the print. Note that if you browse the YTD table after using the length statement, you will still only see "S" in the offending record. However, the print procedure will select the record. That's because the first set also defines the format for the column, and since this is "$1.", we only see the first byte. Consequently, you would need to add a format statement if you want to browse the data and see it correctly.

```
Length CODE $2;
Format CODE $2.;
```

The second way to model data is to use the "Attrib" statement since this can apply both length and format specifications, and has the extra available benefit of labelling the columns so that we can clearly identify the purpose of the underlying data.

```
Attrib  DATE     Format = Date9.       Label = "Transaction date"
        PRODUCT  Length = $4           Label = "Product code"
                 Format = $4.
        COST     Format = Dollar13.2   Label = "Price paid in sale"
        SOLDBY   Length = $3           Label = "Sales person identifier"
                 Format = $3.
        CODE     Length = $2           Label = "Transaction code"
                 Format = $2.;
```

The authors believe that the "Attrib" statement should be used on all permanent data sets to provide program and data documentation to the minimum required level for the user.

VIEWTABLE: YTD

	Transaction date	Product code	Price paid in sale	Sales person identifier	Transaction code
1	04JAN2007	ABC1	$2.37	TH	S
2	13JAN2007	DEF1	$7.21	SH	S
3	23JAN2007	ABC1	$2.45	FM	T
4	26JAN2007	ABC1	$2.37	TH	S
5	29JAN2007	DEF1	$7.58	TH	T
6	30JAN2007	GHI1	$12.25	SH	S
7	07FEB2007	DEF1	$7.21	SH	S
8	17FEB2007	ABC1	$2.37	FM	T
9	23FEB2007	ABC1	$2.37	TH	S
10	25FEB2007	GHI1	$12.25	SH	S9

Figure 3: data set with labels and formats

Write a suitable "Attrib" statement for your YTD data set and view the result.

Then check and ensure the "CODE" values are correct.

EXAMPLE D. UPDATING AND SPLITTING RECORDS

SCENARIO

To verify the bookkeeping, we want to split the sale cost into two components, called "Co" for "Stock cost" and "Pr" for "Sale profit". Produce a table that contains the sale data recorded as two unbundled items. Add a unique identifier onto each record.

Most of us have seen the SAS automatic variable _N_, either in code we have written or read, or perhaps in the log when error records have been produced. We'll use this value to generate an identifier for the record.

The following code creates a format table that holds the cost information for our stock on each of our three items. Then we use the format to include the cost in our calculation and write out the difference.

```
Proc Format;
  Value $PrdCost "ABC1" = "1.12"
                 "DEF1" = "2.27"
                 "DLC1" = "3427986.28"
                 "GHI1" = "3.42"
                 "RRC1" = "9.38";
Run;

Data COSTING;
  Set YTD;
  UNIQ_SEQ = Put( Sum( ( DATE * 100000), _N_), 10.);
  COST     = COST - Input( Put( PRODUCT, $PrdCost.), 8.2);
  CODE     = "Pr";
  Output;
  UNIQ_SEQ = Put( Sum( ( DATE * 100000), _N_), 10.);
  COST     = COST - Input( Put( PRODUCT, $PrdCost.), 8.2);
  CODE     = "Co";
  Output;
Run;
```

Note that rather than hard code the costs of our stock items we would probably use a control table to generate the format. There are SUGI papers that deal with generating formats from source data, including one listed as a reference at the end of this paper.

When we use a format, we create a text representation of a given value – this is why our code includes the use of the Input() function to convert the textual value of "1.12" to 1.12.

Our unique sequence number uses the _N_ mentioned above to increment a large number that is based on the transaction date. However, this method did not work as we expected, as the following screen shot of the data shows.

VIEWTABLE: COSTING

	Transaction date	Product code	Price paid in sale	Sales person identifier	Transaction code	UNIQ_SEQ
1	04JAN2007	ABC1	$1.25	TH	Pr	1717000001
2	04JAN2007	ABC1	$1.12	TH	Co	1717000001
3	13JAN2007	DEF1	$4.94	SH	Pr	1717900002
4	13JAN2007	DEF1	$2.27	SH	Co	1717900002
5	23JAN2007	ABC1	$1.33	FM	Pr	1718900003
6	23JAN2007	ABC1	$1.12	FM	Co	1718900003
7	26JAN2007	ABC1	$1.25	TH	Pr	1719200004
8	26JAN2007	ABC1	$1.12	TH	Co	1719200004
9	29JAN2007	DEF1	$5.31	TH	Pr	1719500005
10	29JAN2007	DEF1	$2.27	TH	Co	1719500005

Figure 4: Transaction data split between cost and profit.

Note that the sequence number is repeated on adjacent records, despite our having redefined it in our code. The issue here is that _N_ is actually a count of the number of times the data step iterates, and is not a count of the number of records. We need to retain a sequence number that we increment manually each time we output a record. This example will remind us that it is the SET statement that defines the data step iteration in these program excerpts.

To confirm, we can return to our original code and select some records that we'll report to the log. Adding the following code to the YTD data step will help.

```
If CODE Eq: "T"  Then
  Put "In record " _N_ SOLDBY " earned " COST " for the company.";
```

```
In record 3 FM  earned $2.45  for the company.
In record 5 TH  earned $7.58  for the company.
In record 8 FM  earned $2.37  for the company.
```

If we now place a "Where" clause in the data step to select only the sales by FM, we'll see the change in the _N_ number reported. It will no longer include records 3 and 8. Include a suitable "Where" statement in your code and note the different message.

We'll return to some special implementations of the "Set" statement later in this workshop.

II. MERGE

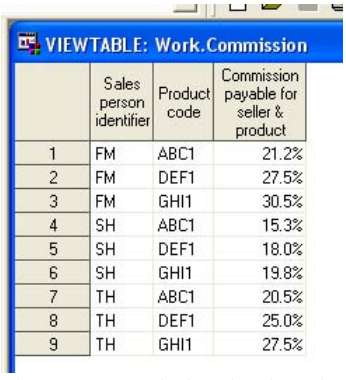
EXAMPLE A. MATCH MERGING.

SCENARIO

Our profitability report overstates our earnings, because it does not subtract the commission paid to our sales team. Reproduce the COSTING records, and add a third record for the commission "Cn", making sure the profit is also reduced. Note that commission varies by sales person and by item sold.

In our previous example, we added simple costs from a format. We could use a similar approach for commission, except that this is dependent on two terms and not on one. We could build a "foreign-key" which concatenates our sales person id to the item code and then generate a format, but this approach only works where a single term is being added to a data step. Adding multiple terms from a reference table would call for multiple formats, so let's look at a simpler approach, which is to match records between two tables using a key value.

Our commission data comes from a table called COMMISSION, which looks like this.



VIEWTABLE: Work.Commission

	Sales person identifier	Product code	Commission payable for seller & product
1	FM	ABC1	21.2%
2	FM	DEF1	27.5%
3	FM	GHI1	30.5%
4	SH	ABC1	15.3%
5	SH	DEF1	18.0%
6	SH	GHI1	19.8%
7	TH	ABC1	20.5%
8	TH	DEF1	25.0%
9	TH	GHI1	27.5%

Figure 5: Commission data by salesperson and product

The data is ordered by SOLDBY and PRODUCT, and matches commission rates according to the experience level of the Sales person and the profit margin on the product.

To match our YTD data, we want to associate the commission rate for a given sales person and product with the sale record that matches this value. This is called a "matching key". To use the matching key we need to have both tables ordered by this key, and then specify the key in our matching data step. We start by ordering the transaction data according to the matching key as shown below.

```
Proc Sort Data = YTD;
  By  SOLDBY PRODUCT;
Run;
```

Note that the key is identified by using it in a "By" statement on the Sort procedure. To use the same key on the match merge, we specify the same key.

To tell SAS that the data step will match merge the source tables, we use the "Merge" statement in place of the "Set" statement. That part of the code looks like the following.

```
  Merge YTD
        COMMISSION;
  By  SOLDBY PRODUCT;
```

Then within the data step we calculate our component costs for each sale and write out the unbundled transaction records.

```
Data COSTING;
  Merge YTD
        COMMISSION;
  By  SOLDBY PRODUCT;
  /* Calculate the component costs of the sale */
  WHOLESALE = Input( Put( PRODUCT, $PrdCost.), 8.2);
  COMMPAID  = Round( ( COST - WHOLESALE) * COMMISSION, 0.01);

  /* Unbundle the profit from the sale */
  COST      = COST - WHOLESALE - COMMPAID;
  CODE      = "Pr";
  Output;

  COST      = COMMPAID;
  CODE      = "Cn";
  Output;

  COST      = WHOLESALE;
  CODE      = "Co";
  Output;
Run;
```

The data looks like the following table image.

VIEWTABLE: COSTING

	Transaction date	Product code	Price paid in sale	Sales person identifier	Transaction code	Commission payable for seller & product	WHOLESALE	COMMPAID
1	23JAN2007	ABC1	$1.05	FM	Pr	21.2%	1.12	0.28
2	23JAN2007	ABC1	$0.28	FM	Cn	21.2%	1.12	0.28
3	23JAN2007	ABC1	$1.12	FM	Co	21.2%	1.12	0.28
4	17FEB2007	ABC1	$0.98	FM	Pr	21.2%	1.12	0.27
5	17FEB2007	ABC1	$0.27	FM	Cn	21.2%	1.12	0.27
6	17FEB2007	ABC1	$1.12	FM	Co	21.2%	1.12	0.27

Figure 6: Transactions split into Cost, commission and profit components.

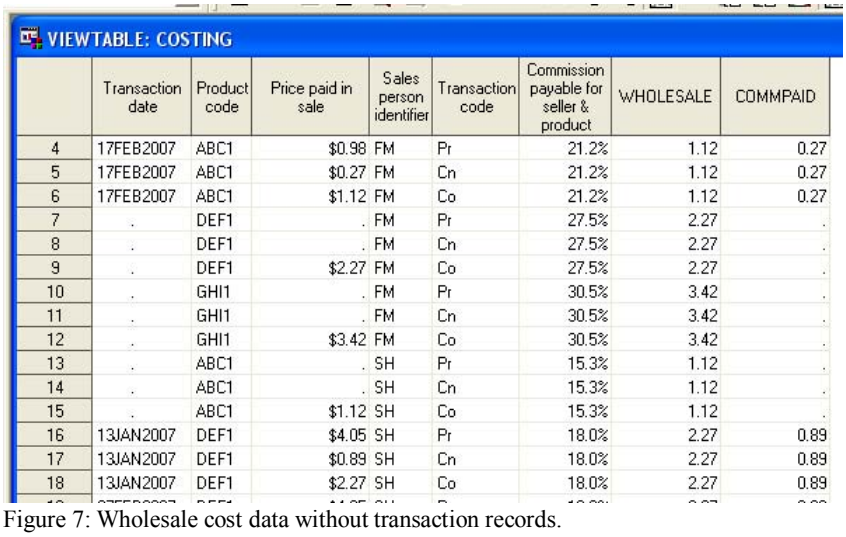
As we can see, the wholesale price remains fixed ($1.12) while the profit and commission values change according to the price that the seller negotiated with our sales person.

When we look in the log however, we see the following.

```
NOTE: Missing values were generated as a result of performing an operation
on missing values.
      Each place is given by: (Number of times) at (Line):(Column).
      4 at 316:15   4 at 316:29   4 at 316:42   4 at 319:20
NOTE: There were 10 observations read from the data set WORK.YTD.
NOTE: There were 9 observations read from the data set WORK.COMMISSION.
NOTE: The data set WORK.COSTING has 42 observations and 8 variables.
```

The “missing values” note should immediately alert us that something is wrong. If every sale has a cost, (we saw that it did) every item has a wholesale price (we expect that it should) and every sales person has a commission rate for each product, then we should not have missing values.

We might take a second clue from the record counts. The YTD table contained 10 sales records, each of which will be split into three component parts. Therefore, the COSTING table should contain 30 transaction records. However, the log notes that it contains 42. Let’s look further down the data.



VIEWTABLE: COSTING

	Transaction date	Product code	Price paid in sale	Sales person identifier	Transaction code	Commission payable for seller & product	WHOLESALE	COMMPAID
4	17FEB2007	ABC1	$0.98	FM	Pr	21.2%	1.12	0.27
5	17FEB2007	ABC1	$0.27	FM	Cn	21.2%	1.12	0.27
6	17FEB2007	ABC1	$1.12	FM	Co	21.2%	1.12	0.27
7	.	DEF1	.	FM	Pr	27.5%	2.27	.
8	.	DEF1	.	FM	Cn	27.5%	2.27	.
9	.	DEF1	$2.27	FM	Co	27.5%	2.27	.
10	.	GHI1	.	FM	Pr	30.5%	3.42	.
11	.	GHI1	.	FM	Cn	30.5%	3.42	.
12	.	GHI1	$3.42	FM	Co	30.5%	3.42	.
13	.	ABC1	.	SH	Pr	15.3%	1.12	.
14	.	ABC1	.	SH	Cn	15.3%	1.12	.
15	.	ABC1	$1.12	SH	Co	15.3%	1.12	.
16	13JAN2007	DEF1	$4.05	SH	Pr	18.0%	2.27	0.89
17	13JAN2007	DEF1	$0.89	SH	Cn	18.0%	2.27	0.89
18	13JAN2007	DEF1	$2.27	SH	Co	18.0%	2.27	0.89

Figure 7: Wholesale cost data without transaction records.

We can see that some records have wholesale cost data (Transaction code = “Co”) but we have no matching transaction date. Our problem is that we have read all our commission records, including those that do not have a matching transaction. What we really needed to do was read ONLY those commission records that matched transactions.

We do that by setting a flag on our transaction table (YTD). The flag will be true when a record is read from the YTD table, and false when it is not. So we can then only output records to our COSTING table where the YTD transaction flag is true. We need to modify our code as follows.

```
Merge YTD( In = Tr)
      COMMISSION;
By  SOLDBY PRODUCT;
If Tr;
```

This will create a new temporary variable on the PDV called "Tr". Since it is a new variable, the name must not be the same as any variable defined on the input tables, or required on output tables. Otherwise it can be any legal SAS variable name. Now modify your code, and check the log to ensure the missing values have gone, and the correct number of records has been output.

EXAMPLE B. INCOMPLETE MATCHING DATA

SCENARIO

When our report is reviewed, we are told that something is wrong with some of the records. On closer examination, we discover that an incorrect salesperson identifier has been recorded in the system.

Let's change the SOLDBY value on our possibly fraudulent record from "SHR" to an unknown sales person id. This will give us some data to replicate the problem. The following code will achieve this.

```
Data YTD;
  Set YTD;
  If CODE Eq "S9"  Then  SOLDBY = "XXX";
Run;
```

When we rerun the merge step we find the following records.

VIEWTABLE: COSTING

	Transaction date	Product code	Price paid in sale	Sales person identifier	Transaction code	Commission payable for seller & product	WHOLESALE	COMMPAID
22	23FEB2007	ABC1	$0.99	TH	Pr	20.5%	1.12	0.26
23	23FEB2007	ABC1	$0.26	TH	Cn	20.5%	1.12	0.26
24	23FEB2007	ABC1	$1.12	TH	Co	20.5%	1.12	0.26
25	29JAN2007	DEF1	$3.98	TH	Pr	25.0%	2.27	1.33
26	29JAN2007	DEF1	$1.33	TH	Cn	25.0%	2.27	1.33
27	29JAN2007	DEF1	$2.27	TH	Co	25.0%	2.27	1.33
28	25FEB2007	GHI1	.	XX	Pr	.	3.42	.
29	25FEB2007	GHI1	.	XX	Cn	.	3.42	.
30	25FEB2007	GHI1	$3.42	XX	Co	.	3.42	.

Figure 8: Transaction records without commission data.

Now we have a transaction record, as evidenced by the transaction date, and we derived a wholesale price based on our format table. However, we have lost the remaining cash flow because there is no commission value.

Our solution will be to take the following steps:

- Add a second table ("MISMATCH") to the "Data" statement to store mismatched records.
- Put a flag on the COMMISSION table to identify a record has been retrieved from this table.
- Where the flag is false, write the record to the MISMATCH table.
- Otherwise continue to do the other calculations and output.

Be sure that the flag on the COMMISSION table has a different name to the flag on the YTD table.

The existing output statements will also need to be changed to put the data in the COSTING table.

When you get the right result, your COSTING table will only hold records that are complete in all parts, and your

MISMATCH table will only hold records that are incomplete. Now remove the table name from the "Output" statement and observe the results on both tables. What does an "Output" statement do when an output table is not specified?

Finally, restore the table name to the 'Output" statement and then use the same flag name on each input table and observe the results. Do you recognize the problem, and understand why it is occurring?

EXAMPLE C. MULTIPLE MATCHES.

SCENARIO

The spreadsheet was updated to reflect some new commission levels, but the reports for this month don't show the new commission rates. What is wrong?

This situation is all too common where data is being retrieved from unstructured sources like Excel spreadsheets that are maintained by people with a speciality in business, but limited knowledge about data structure and good programming practices. If we browse the table NEWCOMMISSION we can see the changes that were made.

VIEWTABLE: NEWCOMMISSION

	Sales person identifier	Product code	Commission payable for seller & product
1	FM	ABC1	21.2%
2	FM	ABC1	23.5%
3	FM	DEF1	27.5%
4	FM	DEF1	28.5%
5	FM	GHI1	30.5%
6	FM	GHI1	31.5%
7	SH	ABC1	15.3%
8	SH	DEF1	18.0%
9	SH	GHI1	19.8%
10	TH	ABC1	20.5%
11	TH	DEF1	25.0%
12	TH	GHI1	27.5%

Figure 9: Commission data with new rates for sales person FMS

Our sales person "FMS" now has two commission rates for each of the products. Comparing this table to the previous table, we can see that a new deal providing higher commission rates has been struck. Unfortunately, the new rates have simply been added to the table, and we have no identifying data to show whether the rates have been increased or decreased. Here is the transaction data produced from using this new commission table.

VIEWTABLE: COSTING

	Transaction date	Product code	Price paid in sale	Sales person identifier	Transaction code	Commission payable for seller & product	WHOLESALE	COMMPAID
1	23JAN2007	ABC1	$1.05	FM	Pr	21.2%	1.12	0.28
2	23JAN2007	ABC1	$0.28	FM	Cn	21.2%	1.12	0.28
3	23JAN2007	ABC1	$1.12	FM	Co	21.2%	1.12	0.28
4	17FEB2007	ABC1	$0.96	FM	Pr	23.5%	1.12	0.29
5	17FEB2007	ABC1	$0.29	FM	Cn	23.5%	1.12	0.29
6	17FEB2007	ABC1	$1.12	FM	Co	23.5%	1.12	0.29

Figure 10: Costing table produced from duplicated commission data

The same sales person, "FMS", for the same product "ABC1" is paid commission for one transaction at the rate of 21.2% and another at the rate of 23.5%. While we might expect the sales person is closely monitoring the commissions paid, and will be quick to identify underpayment, we may not hear so quickly about overpayments. Our concern is whether there is any way we can identify this situation, and our answer is to be found in the log.

```
NOTE: MERGE statement has more than one data set with repeats of BY
values.
NOTE: There were 10 observations read from the data set WORK.YTD.
NOTE: There were 12 observations read from the data set
WORK.NEWCOMMISSION.
```

```
NOTE: The data set WORK.COSTING has 30 observations and 8 variables.
```

We can see that the right number of records was read, and the right number was written. There are no errors or warnings in the log, and we might not recognise the significance of the first "NOTE:" in the log. It is a significant message and this example underscores the need to carefully read the SAS log.

For this Salesperson, and this product, we have two transactions. We also have two commission rates. The question is: which commission rate should apply to each transaction? SAS has matched one transaction to one commission rate, and one to the other commission rate. We haven't lost any data, but our calculations are questionable.

Ideally we would like the commission table to be unique for each combination of sales person and product, but if it weren't then we would ask that an effective date be added to allow us to select the data.

For this exercise however, take the NEWCOMMISSION table, assume the correct commission rate is found on the last record for each combination of sales person and product, and only keep those records. Then match the commission rates to the YTD data and produce a new COSTING table. The expected results will be provided during the presentation. You should also have randomly selected some values and manually calculated the expected results so you can verify your own calculations.

In finishing with the "Merge" statement, let's remind ourselves that a match merge will match all records of a given key value on one table with all records of the same key value on another table. That is, if you have a table with three sales persons details in it, and you match merge a transaction table that contains three transactions for each sales person, then you expect to have a total of nine records at the end of the data step.

EXAMPLE D. MISMATCHED KEY VARIABLES.

SCENARIO

Earlier in this tutorial we looked at the problem of different column definitions on incoming data sets with the "Set" statement. While this situation did not give us any warnings, it caused us problems with our final data set. Let's now examine the situation where we are matching tables and have different data lengths on the key column for our two data sets. Write the following code, which will define translations for our product codes.

```
Data PRODUCTNAMES;
  Attrib PRODUCT       Length = $5  Label = "Product code value"
         DESCRIPTION Length = $32  Label = "Description of product";
  Input #1 PRODUCT $5.
         #2 DESCRIPTION $Char32.
  Cards;
ABC1
Dr Seuss calendar
DEF1
Channel sink plug
GHI1
Golf hole reamer
;
Run;

Data DESCRIBED;
  Merge COSTING( In = Co)
           PRODUCTNAMES;
  By PRODUCT;
  If Co;
Run;
```

Carefully read the log, and observe the messages created by this match merge. If you browse the two input tables, you'll realise that the product code values match, and that the match merge should be correct. Indeed, if you browse the output table you will find that the descriptions have been correctly applied.

The issue lies in the definition of the tables. In the COSTING data, which we read from Excel, the product code is a 4-byte character string, but we have defined it in our PRODUCTNAMES as a 5-byte character string. This is a common problem, especially where data is retrieved from unstructured sources like Excel. It may be that our corporate system defines the product code as a 5-byte string, and the PRODUCTNAMES table is correct. In such a case, we need to correct the data structure from our uncontrolled source.

While we could use an attrib or length statement to force the definition, if we need to do this with multiple data sets there is the chance we will make an error in our program changes, or miss a key source. Where we have a "Model" table, which is a table that has all our data defined correctly, it is often better to use this structure to force our unstructured data set to comply. Write the following code, and use it prior to the merge data step, and then rerun the merge.

```
Data COSTING;
  Set PRODUCTNAMES( Obs = 0)
      COSTING;
Run;
```

Do you see that the structure is now corrected, and the data merges without a warning in the log? We are utilising the order of the data sets on the Set statement to define the structure of data from the first set that contains our columns, and force subsequent sets to use the same definition. In this case, we have correctly forced the length of the Product code to be correct. The authors use this technique quite often for cleansing data from multiple unstructured sources, and the technique has then benefit of applying format and label characteristics to the data as well as the column lengths.

EXAMPLE E. EXISTING COLUMNS ON MASTER DATA SET.

SCENARIO

There is however a problem with the merge. If you look at the merged data, you will see that the DESCRIPTION is only populated on the first instance of the PRODUCT. This is the way the Merge statement is designed. Where a column doesn't exist on the primary data set, it is added and populated from the second data set. However, if the variable exists, then the replacement will only occur for the number of matching records that exist in both tables.

There are a number of cures for this problem; one is to use the update statement in preference to the merge statement. If you think about this, you will realise that when the variable already exists on the table, then you really want to replace all instances of the matching variable, which is an update process, not a match merge. A match merge by definition looks for a match, and when the variable exists, it will only match once for each instance of the by variable in the matching table.

If however we expect our matching table to have all possible values for the by variable then we should retrieve the values from our matching table. As stated earlier, this is either by using the update statement (which we examine in the next section), or we can drop the column from our matched table and retrieve the value only from the matching table. Add a drop statement to your merge data step and prove the description is correctly added to all rows.

III. UPDATE

EXAMPLE A. MODIFY A RECORD IN PLACE

Commission rates are now stored in a table rather than a spreadsheet. Sales person "SHR" has completed his probationary period and is to be paid additional percentage points on all sales. He has also completed some product training, and is to be paid additional percentage points for the product line he has now been certified to sell. Update the rate table.

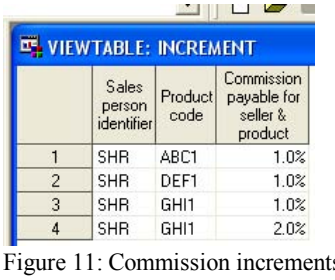
In the match merge steps we walked through previously, we saw that two records matched to one would give you two output records. In this case, we want to have a single commission record for each product for this sales person. We want to add 1% for completion of probation, and 2% for training completion in the GHI1 line.
The following table shows what we expect.

Product	Current commission	Probation completion	GHI1 certification	New commission
ABC1	15.3%	1%		16.3%
DEF1	18.0%	1%		19.0%
GHI1	19.8%	1%	2%	22.8%

We could simply create a new table and merge the data, but in a transaction-based system, we need to keep a record of all the changes that are made. The "Update" statement allows us to make these changes. The "Update" statement differs from the "Merge" statement in a number of significant ways.

Process	Merge outcome	Update outcome
Combine three tables	The combination of the data on the three tables	* Not possible *
Match one person to three transaction records.	Three person/transaction records	One person-record with the cumulative effect of applying three transactions in sequence.

The input data we have is shown below.



VIEWTABLE: INCREMENT

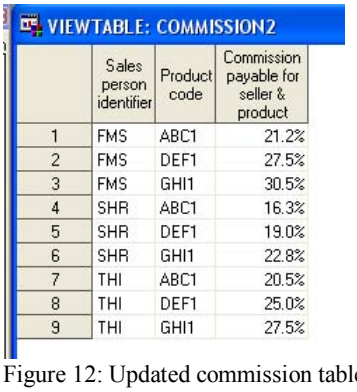
	Sales person identifier	Product code	Commission payable for seller & product
1	SHR	ABC1	1.0%
2	SHR	DEF1	1.0%
3	SHR	GHI1	1.0%
4	SHR	GHI1	2.0%

Figure 11: Commission increments for sales person

Take the code that created the COSTING table as the basis of a new step. Write a table called COMMISSION2 using the COMMISSION and INCREMENT tables as input tables.

Add the INCREMENT to the COMMISSION column to change the commission rate.

At the end of this step, browse the data and verify that the data matches these values.



VIEWTABLE: COMMISSION2

	Sales person identifier	Product code	Commission payable for seller & product
1	FMS	ABC1	21.2%
2	FMS	DEF1	27.5%
3	FMS	GHI1	30.5%
4	SHR	ABC1	16.3%
5	SHR	DEF1	19.0%
6	SHR	GHI1	22.8%
7	THI	ABC1	20.5%
8	THI	DEF1	25.0%
9	THI	GHI1	27.5%

Figure 12: Updated commission table.

EXAMPLE B. UPDATE WITH CUMULATIVE AND NON-CUMULATIVE DATA.

SCENARIO

In Update example A, we used transaction data to increment a term on our commission table. This is a common approach, where a transaction may hold debit and credit information against a key value of client identifier. When we use the client identifier as a matching variable, as in the following table, we then use the debit and credit transactions to increment our debit and credit measures. Here is some example code that creates sample client and transaction data with a key value, and then accumulates a

credit summary.

```
/* Create five client records, with a zero credit balance. */
Data CLIENTS;
  Do CLIENTID = 10001 to 10005  By  1;
    CREDBALANCE = 0;
    Output;
  End;
Run;

/* Create transactions for each client: at least one and randomly
   up to ten, with random values. */
Data TRANSACTIONS( Drop = LOOP);
  Do CLIENTID = 10001 to 10005  By  1;
    Do LOOP = 1  To  10  By  1;
      CREDBALANCE = RanUni( CLIENTID) * 1000;
      Output;
      If RanUni( LOOP) > 0.75  Then  LOOP = 10;
    End;
  End;
Run;

/* Produce a test print of the transaction total to verify the
   update process. */
Proc Summary Data = TRANSACTIONS  NWay  Sum  Print;
  Class CLIENTID;
  Var CREDBALANCE;
Run;

/* Apply the transactions, incrementing the credit balance value. */
Data BALANCES( Drop = CREDIT);
  Update CLIENTS
         TRANSACTIONS( Rename = (CREDBALANCE = CREDIT) );
  By CLIENTID;
  CREDBALANCE = Sum( CREDBALANCE, CREDIT);
Run;
```

When you have run the code, compare the transaction totals in the output screen from the summary procedure against the total accumulated in the "BALANCES" table.

Change the transaction table to also include some simple debit values, and modify the update step to increment these values as well.

When you have done this, add a line of code like the following to hold an update sequence in the transaction table. Then use this in the update step to hold the latest update number. Note that while this is a count of update transactions, it will need to be treated differently to the cumulative debit and credit columns.

```
Retain UPDATESEQ;
UPDATESEQ = ( Date() * 10000) ;
/* Transaction statements */
UPDATESEQ ++ 1;
```

This example should highlight that there are multiple ways to deal with transactions on an update step, and that it is vital to understand the nature and purpose of the data we are working with before we try to produce analytics.

Verify that the UPDATESEQ is unique for each transaction, and there is no break in the sequence. If there is, then go back to the code and try to find the error in the statements you have added.

IV. MODIFY

For completeness of the subject, let's briefly look at the Modify statement. It was introduced in SAS version 6.07 and is thoroughly treated in the Change and Enhancement book P-222 that is detailed at the end of this paper.

To understand how the modify process works, we need to examine the underlying process for the previous three statements. Consider the following code.

```
Data STOCK;
  Set STOCK;
  HOUSEDCOST = HOUSEDCOST + 0.23;
Run;
```

At the beginning of a new month, we add $0.23 to the cost of all items in stock to reflect its cost to us in terms of storage space, stock management, security and frozen capital. We are replacing the stock records, because we want these to be used for all future transactions and calculation of profit and loss.

When we start this process, there is a SAS data file called STOCK.SAS7BDAT stored on one of our disks.

When SAS starts to update the data, it creates a new file called STOCK.SAS7BDAT.LCK and copies each record (with cost updates) from STOCK.SAS7BDAT into this file.

When the SAS data step is finished, STOCK.SAS7BDAT is deleted, and STOCK.SAS7BDAT.LCK is renamed to STOCK.SAS7BDAT.

If the process is aborted before completion because our computer suddenly shuts down, then we may have STOCK.SAS7BDAT and STOCK.SAS7BDAT.LCK still on our hard drive and we can restart the update when our computer is on and stable again.

For any process with critical data, or large tables, this is the process most SAS professionals will advise. It is safe, and recoverable in the event we have a computer problem.

However, if we are processing a large table, then at some point we will have two equally sized large tables on our disks, just as we had with STOCK.SAS7BDAT and STOCK.SAS7BDAT.LCK. Where we are short of disk space, we may not be able to complete this process. So SAS has a process that will modify a table in place. SAS also warns that performing this process may leave you with a corrupted and unreadable data set if your computer operating system crashes or has a problem writing the altered data to your storage.

Here is how we can perform the same update in place on the STOCK table.

```
Data STOCK;
  Modify STOCK;
  HOUSEDCOST = HOUSEDCOST + 0.23;
Run;
```

Note that we are changing the value of an existing variable in the data set. This is because we cannot add any new columns, or delete any existing columns in a data set when we use the Modify statement.

We can however add records to the table. We can also mark records as deleted, and that is done with the "Remove" statement. Here is an example of the removal of the first three records from a stock data set that has stock values incrementing from 1 to 10.

```
49   Data STOCK;
50     Do LOOP = 1  To  10  By  1;
51       HOUSEDCOST = LOOP;
52       Output;
53     End;
54   Run;

NOTE: The data set WORK.STOCK has 10 observations and 2 variables.

55
56   Data STOCK;
```

```
57      Modify STOCK;
58      If HOUSEDCOST < 4  Then  Remove;
59      Else Replace;
60    Run;

NOTE: There were 10 observations read from the data set WORK.STOCK.
NOTE: The data set WORK.STOCK has been updated.  There were 7 observations
rewritten, 0 observations added and 3 observations deleted.
```

Note however that when we explicitly control the records to remove (in line 58), we must also specify the records we will keep (in line 59). The "Replace" statement instructs to "replace this record in the output data set".

While the "Remove" statement appears to be the equivalent of the "Delete" statement in other data steps, the "Replace" statement is NOT the equivalent of the "Output" statement. The "Output" statement is valid for "Modify", and will cause a record to be written. The result is that a record is added to the end of the output table, and is then read again in turn through the modify statement. This behaviour of adding a record and then rereading it when the record pointer reaches the end of the original table can cause the data step to go into an infinite loop that will only stop when the user intervenes or there is no remaining disk capacity for the table.

There are a number of other unexpected behaviours in this statement, so the authors suggest careful reading of the online documentation before this statement is used. It should also not be used where an interruption may occur within the data step, such as disk access, or termination of the SAS session, the operating system or the computer.

V. KEYED TABLE READ IN THE SET STATEMENT

In our earlier match merge, we associated transactions with commission rates using two key fields. A requirement to complete this match correctly was that both tables had to be sorted by the key value. This means we may need to perform three sort procedures, once each on the input tables to facilitate the merge, and one on the output data to restore the original order.

This may be the best approach to take. After all, the Sort procedure and simple data steps are subjects from an introduction to SAS. As consultants, we produce code every SAS programmer should be able to easily understand and maintain. This has a clear benefit for the delivery of a project. When we weigh the cost of understanding something less simple against the cost of performing many manipulations of large data sets, we may choose the simpler code. For very large tables though, the time to delivery of a result must also be important and that is when we favour something more efficient. This example shows a more efficient approach.

Earlier on we explored the use of a format to add data to an unsorted table, and commented that if we had a compound term, or multiple columns to add, then this became a more complex solution. Suppose however that we want to match a small table like our COMMISSION data with the larger transaction (YTD) data. We can use a technique called "keyed reading" of the small table to match its data to the transaction table. To accomplish this, we first apply an index to our COMMISSION table with the following code.

```
Proc DataSets  Lib = WORK  NoList;
  Modify COMMISSION;
    Index Create SOLDPROD = ( SOLDBY PRODUCT) / Unique;
Quit;
```

Now if we could read our two data sets side by side, using the index to select data from the COMMISSION table and match that to the YTD table, then we'd only have to process the YTD table once, and could do so in its natural order.

Try this code:

```
Options Errors = 3;

Data COSTING
     UNMATCH;
  Length ERRMSG $200;
  Set YTD;
  Set COMMISSION Key = SOLDPROD / Unique;
  If _IORC_ = 0  Then Output COSTING;
  Else Do;
    ERRMSG = IORCMSG();
    ERROR_ = 0;
    Put "Pgm Wrn: Mismatch error "  _IORC_  ERRMSG;
    Output UNMATCH;
  End;
  Drop ERRMSG;
Run;
```

The COMMISSION table has a unique index on the combinations of salesperson and product.

The "Key" option "Unique" indicates that there will be a single record in the COMMISSION table matching the YTD data. For this example to work correctly, we need to match each YTD record to no more than one record from the COMMISSION table. (Bear in mind that we earlier resolved an issue where a salesperson might have been paid two different commission rates on a given product. In this example we are assuming that the Commission rate is fixed, and not changing during the period we are reporting.)

The _IORC_ (literally Input / Output Return Code) will be 0, where the record has been found without error. So, where a match exists, we output the matched data to our COSTING table. In all other cases of the _IORC_ value, an error has occurred and we assume the record has not matched correctly and we write the data to the UNMATCH table for further investigation. (Note that in our earlier Merge example we used the "In =" operator to control the output of matched records. This technique produces a similar result.)

This is a very important part of the data step. When we use this technique, we will implicitly retain values from the previous iteration of the data step until a match is found and new data is written from COMMISSION to the PDV. In the exercise that follows, we will see that a record can be output with incorrect matching data.

The "non-zero" return code value on _IORC_ will normally set the data step error flag and produce an error message in the log. Since we are managing matching of records ourselves, we will reset this error flag to prevent the message being produced. Then we produce our own log message to indicate that the match wasn't made.

In this exercise, a table called SUPPLIERS holds product information. The table is ordered, but not indexed by its unique PRODUCT values. Match the SUPPLIERS table to the COSTING table to produce a SUPPLY table. Use the previous code as a template.

Let's examine the code you should have written. When a record is retrieved for the COSTING table, a matching product value is sought from the SUPPLIER table using the PRODUCT index. We should only have cost data for goods we advertise, stock or sold. We expect then that we have a supplier for each of the goods, and a supplier record should be found on the PRODUCT key. We also expect that the product key should be unique for a supplier, so that "Derwent HB pencils" have a different product code to "Staedtler HB pencils". If this isn't the case then we will have a problem with the match, and also a problem with the way we are storing and identifying the data.

In our example above we used a direct test of the I/O Return Code value to identify whether a match was found. Replace the clause "`If _IORC_ = 0  Then`" with the clause "`If %SysRc( _SOK)  Then`" and observe that you get the same result. This clause uses a SAS supplied macro to translate the mnemonic for the return code "Search OK" into a numeric value to match to the system return code for the indexed search. The mnemonic value "_DSENOM" will indicate that the search got to the end of the data set index without finding a match i.e. "Data Set End NO Match". So change your "Else" statement to first test for this value and write the record to your UNMATCH table. Then add a third "Else… Do" block for any other return code as an error trap.

When you test explicitly for "_DSENOM" we don't need to produce any message to the log, so we can remove the log message from this "Do loop" and then report only those return code values that are unexpected.

If it is important that our SUPPLY table contains all PRODUCT records with correct data, then we can null the values that were retained from the previous match to the COSTING table and still write those records to the SUPPLY table as well as writing them to the UNMATCH table for verification and correction of our reference tables.

We have split our data into records we'll process normally, and those that require verification and updating of system tables. Note that the UNMATCH table contains a record for product "RRC1" which is identified as "diLithium crystals" from "Coridan Mining Inc". If you look at the SUPPLY table you'll note that "DLC9" is associated with the same product and supplier.

Once the SUPPLIERS table has been updated, we can run the process again and our alarmingly inexpensive diLithium crystals become a "Road Runner catcher Mk. 1" from "Acme Industries" of Wiley.

The IORC return codes are documented in the SAS online help at “”Mnemonics for Warning and error conditions” in the Online help at “SAS Products / Base Sas / SAS macro reference / Macro language dictionary / %Sysrc Autocall macro”.

CONCLUSION

The Set, Merge and Update statements provide very powerful functionality that is sometimes hidden by the simplicity of the statements. At times they may appear to offer overlapping or interchangeable functionality, but taking the time to thoroughly understand their functionality will pay dividends in more efficient and robust code that delivers the right answers from the correct choice for the task.

REFERENCES

“SAS OnLine Documentation”
“SAS Technical Report P-222: Changes and Enhancements to Base SAS Software, Release 6.07,”

ACKNOWLEDGMENTS

Andrew Kuligowski and Peter Eberhardt for their encouragement and support.
Students and colleagues who haven’t been afraid to ask “simple” questions that required care and thought to answer properly.

RECOMMENDED READING

“Describing and Retrieving Data with SAS(r) Formats” Proceedings of SAS User Group International 28.

CONTACT INFORMATION

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

David H. Johnson
DKV-J Consultancies
Moorabbin, Vic Australia 3189
(Tel) +61 408 83 5355
(Fax) +44 7005 98 0829
sgf07@dkvj.com
www.dkvj.com

Wendy B Dickinson, Coordinator of Mathematics
Ringling School of Art and Design
Sarasota, Florida, 34234
+1 941 359 7521
+1 941 953 5071
wdickins@ringling.edu