Paper 117-2013

A Day in the Life of Data – Part 2

Harry Droogendyk, Stratia Consulting Inc.

ABSTRACT

As a new SAS® programmer, you may be overwhelmed with the variety of tricks and techniques that you see from experienced SAS programmers; as you try to piece together some of these techniques you get frustrated and perhaps confused because the data showing these techniques are inconsistent. That is, you read several papers and each uses different data. This series of four papers is different. They will step you through several techniques but all four papers will be using the same data. The authors will show how value is added to the data at each of the four major steps: Input, Data Manipulation, Data and Program Management, and Graphics and Reporting.

INTRODUCTION

The first paper (116-2013) in this series of four has demonstrated how to import data from a variety of sources. This paper, the second in the series, will illustrate how data is to be manipulated by joining tables, transforming and summarizing the data to make it suitable for reporting and visualization. Various techniques will be explored outlining the advantages and disadvantages of each method.

JOINING DATA

In a typical enterprise setting we deal with millions of rows and thousands of columns of data. Not all this information is going to be available in a single table or dataset. Proper database design demands that the data be normalized as much as practically possible. Normalization is the process that attempts to ensure each data item is stored in only one table. This usually results in multiple tables as the data is broken into components or categories (e.g. personal data, address data, transaction data etc…) to ensure uniqueness. The end result greatly minimizes the potential for data anomalies which may occur by storing the same information in multiple tables. However, that also means that most often tables must be joined to provide data stores suitable for reporting and visualization.

When joining tables, care must be taken to ensure the relationships between the tables do not introduce unwanted results. For instance, typically relationships between tables result in a one-to-one or a One-to-Many Relationship. A One-to-One Relationship would result when the join criteria results in a single row from one table being joined to a single row in another table, e.g. an Employee Master table joined to an Employee Salary table. A one-to-many relationship would result when joining Employee Master to an Employee Hierarchy table where a single manager has multiple reporting employees. A many-to-many relationship would most probably exist between Customer and Product – many customers can order the same product, and many products may be ordered by a single customer. Sometimes the product of a Many-to-Many Relationship is the desired result, but if not, the result set will be much larger than anticipated and results will be incorrect. The examples below will illustrate each of these relationships.

This paper will introduce the two most common techniques for joining tables in SAS is via the data step MERGE and PROC SQL joins. For a fuller treatment of MERGE and PROC SQL usage, see the 178-2008 SUGI paper.

Before moving on, note the following diagram which illustrates which rows from the contributing tables are represented in the final result set for the four different types of joins.

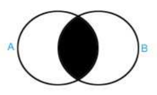
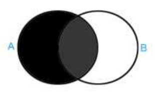
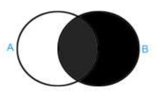
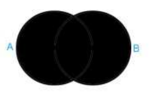
Inner (or Equi-) Join	**Left Outer Join**	**Right Outer Join**	**Fuller Outer Join**
			
Match rows only	**All rows from A, matching rows from B**	**All rows from B, matching rows from A**	**All rows from both tables**

Figure 1: illustrating types of joins

DATA STEP MERGE

Some databases provide an SQL MERGE statement to insert into and update a target table from a source table. The data step MERGE is a quite different in that it allows multiple tables to be brought together with the opportunity to fully control the process with data step logic. Most often the MERGE join criteria is regulated by specifying a common set of BY variables that define the relationship(s) between the tables.

Before joining tables using the data step MERGE, SAS requires that the tables be sorted in BY variable order, or an index must exist that includes the BY variables specified.

One-to-One Relationship

```
data employee_master;
   emp_id = 32;  name = 'George'; hire_dt = '03Feb2012'd;gender = 'M';output;
   emp_id = 13;  name = 'Susan';  hire_dt = '23Nov1999'd;gender = 'F';output;
   emp_id = 7;   name = 'Peter';  hire_dt = '12Apr1998'd;gender = 'M';output;
   emp_id = 45;  name = 'Egbert'; hire_dt = '31Dec2011'd;gender = 'O';output;
   format hire_dt yymmddd10.;
run;

data employee_salary;
   emp_id = 7;   salary = 52000;  increase_dt = '03Feb2012'd;   output;
   emp_id = 13;  salary = 70500;  increase_dt = '14Nov2012'd;   output;
   emp_id = 32;  salary = 67800;  increase_dt = '03May2011'd;   output;
   emp_id = 45;  salary = 43200;  increase_dt = '02jan2012'd;   output;
   format salary dollar10. increase_dt yymmddd10.;
run;

data employee_data;
   merge  employee_master
          employee_salary;
      by  emp_id;
run;
```

```
150  data employee_data;
151      merge employee_master
152            employee_salary;
153         by emp_id;
154  run;

ERROR: BY variables are not properly sorted on data set WORK.EMPLOYEE_MASTER.
emp_id=32 name=George hire_dt=2012-02-03 gender=M salary=67800 increase_dt=18750 FIRST.emp_id=1
LAST.emp_id=1 _ERROR_=1 _N_=3
NOTE: The SAS System stopped processing this step because of errors.
NOTE: There were 2 observations read from the data set WORK.EMPLOYEE_MASTER.
NOTE: There were 4 observations read from the data set WORK.EMPLOYEE_SALARY.
```

Figure 2: results of MERGE with incorrect BY variable order

When tables are not in BY variable order, the MERGE will fail. Before joining these tables, they must both be in EMP_ID order – Employee Salary is in the correct order but Employee Master must be sorted. Figure 2 shows the successful code and partial log.

```
proc sort data = employee_master;
          by   emp_id;
run;

data employee_data;
   merge  employee_master
          employee_salary;
      by  emp_id;
run;
```

```
155  proc sort data = employee_master;
156             by   emp_id;
157  run;

NOTE: There were 4 observations read from the data set WORK.EMPLOYEE_MASTER.
NOTE: The data set WORK.EMPLOYEE_MASTER has 4 observations and 4 variables.

158
159  data employee_data;
160      merge employee_master
161            employee_salary;
162         by emp_id;
163  run;

NOTE: There were 4 observations read from the data set WORK.EMPLOYEE_MASTER.
NOTE: There were 4 observations read from the data set WORK.EMPLOYEE_SALARY.
NOTE: The data set WORK.EMPLOYEE_DATA has 4 observations and 6 variables.
```

```
title 'Employee Data';
proc print data = employee_data noobs;
run;
```

```
Employee Data
                                                             increase_
emp_id     name        hire_dt     gender         salary            dt

   7      Peter      1998-04-12      M          $52,000     2012-02-03
  13      Susan      1999-11-23      F          $70,500     2012-11-14
  32      George     2012-02-03      M          $67,800     2011-05-03
  45      Egbert     2011-12-31      O          $43,200     2012-01-02
```
Figure 3: output of successfully merged data

One-to-Many Relationship

The next example introduces a gender table which will be used to supply a description for the gender code stored on the Employee Master table. Since the Employee Master table created in the previous example has multiple rows with the same gender code, the tables have a one-to-many relationship. There is a wrinkle… each table has a gender code that is not found on the other table. The *type of join* we elect to use in this example will affect our results.

Rather than sorting the Employee Master table again, we've elected to create an index on the table using the gender code. Create the Gender dimension table – adding a code for Unknown and merge the two tables on gender code.

```
proc sql;
   create index gender on employee_master;
quit;

data gender;
   gender = 'F';gender_desc = 'Female ';   output;
   gender = 'M';gender_desc = 'Male    ';   output;
   gender = 'U';gender_desc = 'Unknown';   output;
run;

data employee_gender;
   merge  employee_master
          gender;
       by gender;
run;
```

```
data employee_gender;
    merge employee_master   ( in = m )
          gender            ( in = g );
        by gender;
run;

NOTE: There were 4 observations read from the data set WORK.EMPLOYEE_MASTER.
NOTE: There were 3 observations read from the data set WORK.GENDER.
NOTE: The data set WORK.EMPLOYEE_GENDER has 5 observations and 5 variables.
```

Figure 4: full join of gender data

We have added no additional data step logic to deal with the mismatched data – a full outer join will result. Since the Gender table code value of ‘U’ did not exist in the Employee Master table, an extra row has been created. If a subsequent step was summarizing this data, the additional record would be counted and the number of employees would be overstated, probably not what we want to occur. See the PRINT output in figure 5.

```
title 'Employee Gender';
proc print data = employee_gender noobs;
run;
```

```
Employee Gender

                                                gender_
emp_id     name        hire_dt    gender         desc

  13       Susan     1999-11-23     F          Female
   7       Peter     1998-04-12     M          Male
  32       George    2012-02-03     M          Male
  45       Egbert    2011-12-31     O
   .                          .     U          Unknown
```

Figure 5: full join of gender data – extra row generated with no employee_master data

In conjunction with the IN data set option, the data step allows us to specify logic to ensure extra rows are not introduced into the employee data in cases where the dimension table has values not found in the employee master. In addition, if employee gender code values are not found in the dimension table, a gender_desc value can be supplied.

```
data employee_gender;
   merge employee_master   ( in = m )
         gender            ( in = g );
       by gender;

   if m;          * subsetting if, need employee_master rows;
   if ^g then gender_desc = 'Invalid';
run;

NOTE: There were 4 observations read from the data set WORK.EMPLOYEE_MASTER.
NOTE: There were 3 observations read from the data set WORK.GENDER.
NOTE: The data set WORK.EMPLOYEE_GENDER has 4 observations and 5 variables.
```

Figure 6: employee_master left joined to gender dimension table

The IN= data set option supplies a variable which will be set to true if the data set is contributing to the current observation (that variable is automatically dropped from the output dataset). The “`if m;`” is a *subsetting if* statement that causes the data step to continue processing the current observation only if the condition is true. If it is false, SAS does not output the current observation but immediately returns to the top of the data step and resumes processing with the next observation. In this case, if the Employee Master has not contributed an observation (i.e. we’ve read a row from the Gender dimension table with a gender code that does not exist on the employee_master) , we do not want to output this observation.

In similar fashion, the “`if ^g then gender_desc = 'Invalid';`” statement assigns a value to gender_desc if the gender table does not have a row for the employee_master gender code value.

The end result is what's known as a "left join" in SQL nomenclature. As the log snippet shows, the output table contains 4 observations, the same number as the Employee Master table. The gender code not found on the Gender dimension table now has a value as well.

```
title 'Employee Gender';
proc print data = employee_gender noobs;    run;
```

```
Employee Gender
                                            gender_
emp_id     name        hire_dt    gender     desc

  13       Susan     1999-11-23     F       Female
   7       Peter     1998-04-12     M       Male
  32       George    2012-02-03     M       Male
  45       Egbert    2011-12-31     O       Invalid
```

Figure 7: results of employee_master left joined to gender dimension table

Many-to-Many Relationship

Unfortunately another user didn't care for "Female" / "Male" gender descriptions and decided "Woman" / "Man" might be more suitable. Rather than replacing the existing Gender table, the new values were appended to the existing data. The end result? A Gender table with duplicate values, two records each for the codes F, M and U. If we join Employee Master to the new Gender table, the result set will not be what we want. We're specifying the IN= data set options to supervise the join to ensure we don't create extra observations. Unfortunately those options don't help us with the many-to-many relationship.

```
data gender;
   gender = 'F'; gender_desc = 'Female ';   output;
   gender = 'F'; gender_desc = 'Woman  ';   output;
   gender = 'M'; gender_desc = 'Male   ';   output;
   gender = 'M'; gender_desc = 'Man    ';   output;
   gender = 'U'; gender_desc = 'Unknown';   output;
   gender = 'U'; gender_desc = 'Unknown';   output;
run;

data employee_gender;
   merge employee_master   ( in = m )
         gender            ( in = g );
       by gender;

   if m;  * subsetting if, need employee_master rows;
   if ^g then gender_desc = 'Invalid';
run;
```

```
NOTE: MERGE statement has more than one data set with repeats of BY values.
NOTE: There were 4 observations read from the data set WORK.EMPLOYEE_MASTER.
NOTE: There were 6 observations read from the data set WORK.GENDER.
NOTE: The data set WORK.EMPLOYEE_GENDER has 5 observations and 5 variables.
```

Figure 8: many-to-many MERGE

Once again an extra observation has been introduced – though this time it's not because of the gender code value of 'U'. When the Employee Gender table is printed, the results are somewhat surprising. Susan now appears twice but the extra "M" row in the Gender table has had a different effect - Peter or George now have different gender description values.

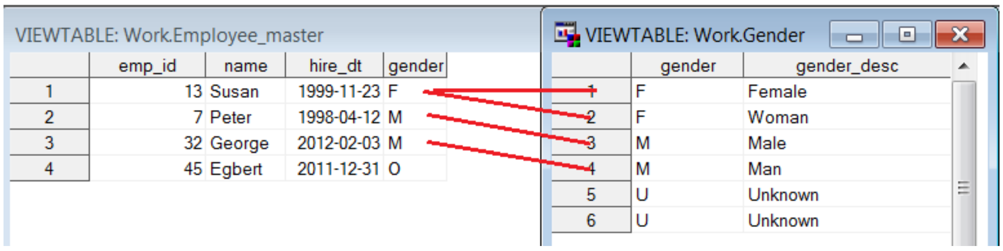
```
Employee Gender

                                              gender_
emp_id     name        hire_dt    gender       desc

  13      Susan     1999-11-23      F         Female
  13      Susan     1999-11-23      F         Woman
   7      Peter     1998-04-12      M         Male
  32      George    2012-02-03      M         Man
  45      Egbert    2011-12-31      O         Invalid
```

Figure 9: many-to-many MERGE results

See Figure 10 for a visual representation of the actions of the MERGE statement when it encounters a many-to-many situation – very unlike what we'll see from PROC SQL.



VIEWTABLE: Work.Employee_master

	emp_id	name	hire_dt	gender
1	13	Susan	1999-11-23	F
2	7	Peter	1998-04-12	M
3	32	George	2012-02-03	M
4	45	Egbert	2011-12-31	O

VIEWTABLE: Work.Gender

	gender	gender_desc
1	F	Female
2	F	Woman
3	M	Male
4	M	Man
5	U	Unknown
6	U	Unknown

Figure 10: many-to-many MERGE matching action

MERGE does not produce a Cartesian product when a many-to-many relationship is encountered. Instead, based on the BY variables values, the first row from the first table will be joined to the first matching row from the second table. If both tables have additional matching rows, the second row from each will be joined and so on. When one table no longer has matching rows, the final matching row will be joined to the rest of the matching rows from the other table. Yeah, I know, I'm confused too. ☺ See the SAS Online Docs for a fuller description of MERGE behavior.

MERGE Advantages

- allows use of familiar data step syntax to govern join logic and data manipulation
- simpler for novice users

MERGE Disadvantages

- requires contributing tables to be sorted or indexed by join variables
- variable names and type must be identical in contributing tables
- many to many joins do not create Cartesian products
- post join summaries must be coded as a separate step

PROC SQL JOIN

PROC SQL allows the programmer to code table joins with SQL syntax very similar or identical to RDBMS SQL syntax for the same purpose. The full suite of joins found in most any database are available in PROC SQL, i.e. inner (or equi-), left, right, full and cross joins. While SQL joins have their "gotchas", it is typically easier to prevent these by the use of sub-queries and syntax such as DISTINCT to ensure undesired results are not generated.

The SQL examples below use the same input data sets as those for the previous MERGE examples.

One-to-One Relationship

Note the specification of the SQL **_METHOD** option which generates LOG output outlining the execution method chosen to do the join. Deciphering the rather cryptic output will tell you if SAS has sorted the tables first behind the scenes or used the more efficient hash method. To encourage SAS to use the hash method, specify a large **BUFFERSIZE=** option as well. In this case, SAS has elected to use the hash method, i.e. see sqxjhsh in the LOG. Listing tables in the FROM clause separated by commas results in an inner or equi-join.

```
proc sql _method buffersize=1e6;
   create table employee_data_sql as
      select m.*, s.salary, s.increase_dt
        from employee_master  m,
             employee_salary  s
       where m.emp_id = s.emp_id
   ;
quit;
```

```
248
249  proc sql _method buffersize=1e6;
250      create table employee_data_sql as
251          select m.*, s.salary, s.increase_dt
252            from employee_master  m,
253                 employee_salary  s
254           where m.emp_id = s.emp_id
255      ;

NOTE: SQL execution methods chosen are:

      sqxcrta
          sqxjhsh
              sqxsrc( WORK.EMPLOYEE_SALARY(alias = S) )
              sqxsrc( WORK.EMPLOYEE_MASTER(alias = M) )
NOTE: Table WORK.EMPLOYEE_DATA_SQL created, with 4 rows and 6 columns.
```

Figure 11: one-to-one SQL equi-join

One-to-Many Relationship

Because the initial MERGE example in the One-to-Many Relationship did not govern the join by using the IN= variables, the output table was the product of a full join. In SQL, one must specify "full join" and use the ON clause to specify the join criteria to achieve the same result.

```
proc sql;
   create table employee_gender_sql as
      select m.*, g.gender_desc
        from employee_master  m
                 full join
             gender           g
          on  m.gender = g.gender
   ;
quit;
```

```
NOTE: Table WORK.EMPLOYEE_GENDER_SQL created, with 5 rows and 5 columns.
```

Figure 12: one-to-many SQL full join

Just as we saw in the MERGE example, the extra "U" Gender row has created an extra row in the output table – probably not what we want. In the MERGE example, the use of the IN= variable was used to ensure the output table only contained rows where the Employee Master table contributed. The same result can be obtained in SQL using the "left join" syntax. The "left" table, i.e. the table name to the left of the "left join" clause governs which rows are included. In cases where the Employee Master table has a gender code that is not found on the Gender table, that row will still be included. To ensure the gender_desc field has a value in those cases, the COALESCE function is used. COALESCE will use the first non-missing value in the argument list provided. If gender_desc is not available from the Gender table because no matching Gender row is found, the value "Invalid" will be used.

```
proc sql ;
   create table employee_gender_sql as
      select m.*, coalesce(g.gender_desc,'Invalid') as gender_desc
        from employee_master  m
                left join
            gender           g
          on  m.gender = g.gender
   ;
quit;
```

```
401
402  proc sql ;
403      create table employee_gender_sql as
404          select m.*, coalesce(g.gender_desc,'Invalid') as gender_desc
405            from employee_master  m
406                  left join
407                 gender           g
408               on  m.gender = g.gender;
NOTE: Table WORK.EMPLOYEE_GENDER_SQL created, with 4 rows and 5 columns.
```

```
Employee Gender SQL
                                            gender_
emp_id     name        hire_dt    gender     desc

  13      Susan     1999-11-23      F       Female
  32      George    2012-02-03      M       Male
   7      Peter     1998-04-12      M       Male
  45      Egbert    2011-12-31      O       Invalid
```

Figure 13: one-to-many SQL left join

Many-to-Many Relationship

Recall that the Gender table was modified incorrectly when another user wanted different gender_desc values for “F” and “M”. MERGE produced a peculiar result, joining matching rows one-to-one and did not produce the Cartesian product we expected – only 5 rows appeared in the output table. SQL creates a different result, matching all rows from the first table to all rows from the second table where the join criteria is satisfied netting 7 rows. Note the ORDER BY clause which sorts the output table.

```
proc sql ;
   create table employee_gender_sql as
      select m.*, coalesce(g.gender_desc,'Invalid') as gender_desc
        from employee_master  m
                left join
            gender           g
          on m.gender = g.gender
        order by m.name
   ;
quit;
```

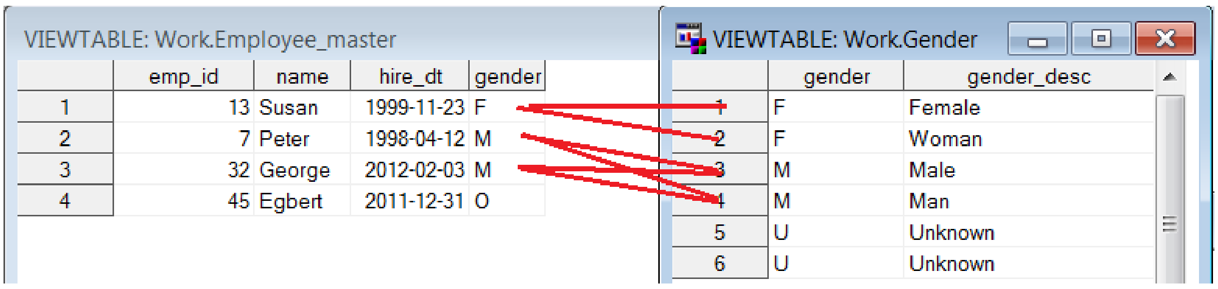
```
NOTE: Table WORK.EMPLOYEE_GENDER_SQL created, with 7 rows and 5 columns.
```

```
Employee Gender
                                            gender_
emp_id     name        hire_dt    gender     desc

  45      Egbert    2011-12-31      O       Invalid
  32      George    2012-02-03      M       Man
  32      George    2012-02-03      M       Male
   7      Peter     1998-04-12      M       Male
   7      Peter     1998-04-12      M       Man
  13       Susan      1999-11-23      F        Woman
  13       Susan      1999-11-23      F        Female
```

Figure 14: one-to-many SQL left join

As Figure 15 illustrates, each Employee Master row with a gender code value of “F” was matched with each “F” row from the Gender dimension table, same action for the “M” values. Rather than the 4 rows we expected to have, it now appears we have 7 employees. Using this table for reporting will lead to erroneous results.



	emp_id	name	hire_dt	gender
1	13	Susan	1999-11-23	F
2	7	Peter	1998-04-12	M
3	32	George	2012-02-03	M
4	45	Egbert	2011-12-31	O

	gender	gender_desc
1	F	Female
2	F	Woman
3	M	Male
4	M	Man
5	U	Unknown
6	U	Unknown

Figure 15: many-to-many SQL join matching action

Real World SQL Join Examples

In the preamble at the top of the JOIN DATA section, we discussed data mart normalization. Typically this results in many tables, thus the need to join different types or groups of data back together again when surfacing it in reports or graphs. Sometimes the join criteria isn’t straightforward and non-technical users find it difficult to make the proper associations to use the data easily. In those cases, it’s often helpful to create a view of the many tables and provide access to that view for those requiring non-normalized data. The example below accomplishes this for the Orion Gold data, joining the Order Fact table to the various dimension tables, creating a wide data store of all columns that might be required.

If we use the table_alias.* syntax to select all columns from each of the tables, we will end up with duplicate column names and SQL will complain (though it will execute successfully). Rather than have the warning appear in the LOG, we’d rather explicitly specify the non-join columns from Order Fact in the SELECT and use the * to pull all columns from the dimension tables. However, we’re lazy – we don’t actually want to type all the column names.

Thankfully SAS (and most databases) maintain meta data, or dictionary tables, which tells us things about the data in our repository. From the SAS meta data we can derive many things:

- librefs assigned to our session
- global and automatic macro variables
- tables and columns in the assigned librefs
- etc… see the views in the automatic library reference SASHELP, e.g. VCOLUMN

Rather than trying to remember which columns we need and type them in, we’re going to query the list of column names from the SAS meta data, send that list to a macro variable, and use the macro variable in the SELECT. Lazy programmers are often efficient programmers.

We want the list of Order Fact column names that are not in common with the key fields of the dimension tables. We’ll create a table of these columns, their data types and position within the table. A second query will take the resulting column names and create a macro variable which contains the SELECT syntax we need to select the Order Fact columns – specifying the table alias to be used for the Order Fact table, eg. ord.*column_name*.

```
proc sql noprint;
   create table order_fact_cols as
      select name, type, varnum
        from sashelp.vcolumn
       where upcase(libname) = 'GOLD'
         and upcase(memname) = 'ORDER_FACT'
         and upcase(name)    not in
                            ('STREET_ID','PRODUCT_ID','EMPLOYEE_ID','CUSTOMER_ID')
   ;
   select cats('ord.',name)
     into :order_fact_select separated by ', '
     from order_fact_cols
    order by varnum
   ;
quit;

%put Selecting ORDER_FACT columns: &order_fact_select;
```

Figure 16: querying meta data to derive column names

Note the syntax of the second query. The INTO : clause sends the results of the SELECT into a macro variable named order_fact_select. Because we're expecting multiple values (because Order Fact has many columns), the SEPARATED BY clause tells SQL to insert a comma between values. The end result is a comma-delimited list of qualified column names separated by commas – valid SQL syntax. Lazy programmers rock.

```
826  proc sql noprint;
827      create table order_fact_cols as
<snip>
833      ;
NOTE: Table WORK.ORDER_FACT_COLS created, with 8 rows and 3 columns.

834      select cats('ord.',name)
835        into :order_fact_select separated by ', '
836        from order_fact_cols
837       order by varnum
<snip>
841  %put Selecting ORDER_FACT columns: &order_fact_select;
Selecting ORDER_FACT columns: ord.Order_Date, ord.Delivery_Date, ord.Order_ID, ord.Order_Type,
ord.Quantity, ord.Total_Retail_Price, ord.CostPrice_Per_Unit, ord.Discount
```

Figure 17: LOG results from meta data query

Building the view is straightforward. Note the inclusion of &order_fact_select. Before the query runs, SAS will substitute the value of the macro variable from the global symbol table (for more macro stuff, see this paper from my website) and the SELECT will be complete. Because we've omitted the various *_ID variables from the macro variable list, we can use the * syntax to select all columns from the dimension tables and not generate nasty warning messages in the log.

```
proc sql;
   create view order_fact_all_dims as
      select &order_fact_select
         ,   cust.*
         ,   geo.*
         ,   org.*
         ,   prod.*
        from    gold.order_fact        ord
                left join
            gold.customer_dim   cust
         on  ord.customer_id    = cust.customer_id
                left join
            gold.geography_dim     geo
         on  ord.street_id      = geo.street_id
                left join
            gold.organization_dim  org
         on  ord.employee_id    = org.employee_id
                left join
            gold.product_dim    prod
         on  ord.product_id     = prod.product_id   ;
quit;
```

Figure 18: creating the view using the macro variable created from the meta data

In the next section, we'll use this view to summarize the Order Fact data, incorporating some of the dimension table columns in our summary definition.

A second real world example illustrates how SQL is also very helpful when creating more complex result sets involving multiple tables and join criteria that includes ranges rather than simple equality conditions. The Orion order data includes a DISCOUNT table which indicates when different products were marked down. We'd like to identify the full-price order items and the number of days by which the customer bought before or after the discount period started or ended. Since it's possible that a product be discounted more than once summary functions will be utilized to ensure we find the discount period closest to the order date.

```
proc sql;
   create table orders_no_discount_days as
      select o.order_id, i.order_item_num, i.product_id
         ,  o.order_date, o.delivery_date
         ,  min(case when o.order_date < d.start_date
                then d.start_date - o.order_date  else . end)  as days_before
         ,  min(case when o.order_date > d.end_date
                then o.order_date - d.end_date    else . end)  as days_after
        from orion.orders              o
                inner join
            orion.order_item           i
          on o.order_id     = i.order_id
                inner join
            orion.discount             d
         on i.product_id     = d.product_id
       where o.order_date     > '01dec2008'd
         and i.discount       = .
         and ( o.order_date < d.start_date  or  o.order_date > d.end_date )
       group by 1,2,3,4,5
      having calculated days_before <= 30 or calculated days_after <=30
       order by 1,2,3,4,5 ;
quit;
```

Figure 19: multiple table summary query

The query involves three equi-joins (or inner joins), bringing together three tables where the join criteria specified in the ON phrase is satisfied. The WHERE clause is filtering the results to ensure we're only considering orders after a certain date and without a discount. CASE statements are used to generate values only where specific conditions are met. Since we only want the discount period closest to the full-price order item, the MIN() summary function is being employed. When using summary functions, grouping columns must be defined, hence the GROUP BY. The HAVING clause is like a where clause, but it is applied to the data *after* grouping and summarization has occurred.

How many data and sort steps would be required to accomplish what one SQL query has generated?

PROC SQL Advantages

- contributing data sets do not need to be sorted or indexed by the joining variables, SQL will sort automatically or use hash tables behind the scenes
- variables used to join can have different names on each contributing table
- variable types can be modified in the join criteria where necessary to make them match
- multi-table joins where join criteria is different for each relationship can be performed in a single query
- syntax is portable and may be used with RDBMS queries if tables are subsequently hosted on a different platform
- ranges can be used when joining, e.g. when a.date between b.start_dt and b.end_dt
- sub-queries can be used create sets of distinct values to avoid Cartesian products
- where a Many-to-Many Relationship is required, the correct Cartesian product is created
- summarization can occur in the same SQL statement as the join

PROC SQL Disadvantages

- data manipulation is less flexible since conditional processing is more difficult
- SELECT variable lists must often be coded explicitly

SUMMARIZING DATA

Most reporting we do requires summarized data. While a very granular report is sometimes helpful, it's often most useful to generate summary reports and only drill-down into the details when warranted.

SAS provides a number of ways to summarize data. As seen in the second real world example, one of those ways is SQL. However, given the choice, if SAS has created a PROC to do something, usually it's most efficient to use the PROC to do the heavy lifting – that's what it's been designed to do.

Using the view created in the previous section, a summary table is to be created by Organization Group, Job Title and Month. Separate summaries are to be created for the NWAY (i.e. all combinations of summary variables), Org_Group and Month, and by Month. While PROC SUMMARY will not be discussed in detail, the code below does show some typical summarization work that is often required when readying data reporting and visualization.

```
/*  Summarize the order data by Org Group, Job Title and Month  */

proc summary data = order_fact_all_dims missing chartype;
   class org_group job_title order_date;
   var quantity total_retail_price;
   output out = monthly_org_job_sales
      ( rename = ( _freq_ = sales_cnt )
         where = ( _type_ in ( '111','101','001' ))
      ) sum=;
   format order_date yymmn6. ;
run;

/*  Join sales summary data to monthly targets  */

proc sql;
   create table monthly_sales_targets as
      select s.*, t.sales_target, s.quantity - t.sales_target as variance
        from monthly_org_job_sales_me      s
                full join
            sales_target_data              t
         on t.org_group      = s.org_group
        and t.job_title      = s.job_title
         and t.target_date   = intnx('month',s.order_date,0,'end')
   ;
quit;
```

Figure 20: multiple summary table join to monthly targets

The PROC SUMMARY is creating three separate summary points in one pass through the data – something PROC SQL cannot do. The _type_ variable dictates which summary combinations of the CLASS variables are be kept in the summary output data set. In this case, the summary points included are org_group * job_title * month, org_group * month and month. The number of rows at each summary point is the number of order items rolled up to that summary point, hence the output data set option renaming the automatic variable _freq_ to sales_cnt. The inclusion of the FORMAT statement indicates the ORDER_DATE values are to be summarized at the formatted value, i.e. year & month and not the raw order_date values (scattered through the month as they likely are). The actual order_date value stored in the summarized output dataset will be the *minimum* order_date value in the group.

After summarizing the sales figures, the target data is merged in to determine sales target variances. Since the sales target data uses month-end dates, the join criteria is advancing the order_date value to the end of the month using the INTNX function to line-up with the month-end values of the target dates.

The resulting MONTHLY_SALES_TARGETS table is now ready for the reporting group. Using the reporting and graphing capabilities of SAS, the summarized data will be presented in an easily digestible manner, allowing the data to tell their story.

CONCLUSION

Our enterprises are accumulating vast amounts of data every day – much of it stored in data warehouses. Gaining insight into the data and leveraging the usefulness of it usually requires combinations of tables, data transformation and summarization. SAS is a toolbox that often provides multiple means of solving business problems. Data step MERGE and PROC SQL both have their strengths and can be employed as the situation warrants. However, it is imperative that one know their data and ensure table joins do not "multiple" rows through careless join criteria, non-distinct key values etc..

ACKNOWLEDGMENTS

Thanks to Peter Eberhardt for creating this series of papers – great idea!

RECOMMENDED READING

Bee, Brian "A Day in the Life of Data – Part 1", P*roceedings of the 2013 SAS Global Forum 2013*

Crawford, Peter "A Day in the Life of Data – Part 2", P*roceedings of the 2013 SAS Global Forum 2013*

Matange, Sanjay "A Day in the Life of Data – Part 4", P*roceedings of the 2013 SAS Global Forum 2013*

CONTACT INFORMATION

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

Harry Droogendyk
Stratia Consulting Inc.
PO Box 145
Lynden, ON L0R 1T0
conf@stratia.ca
www.stratia.ca

To request the Orion and Orion Gold data used in this paper, send an email to dayinlife@fernwood.ca to obtain the download address.