# Joining Data: Data Step Merge or SQL?

Harry Droogendyk, Stratia Consulting Inc., Lynden, ON
Faisal Dosani, RBC Royal Bank, Toronto, ON

## ABSTRACT

This paper explores the joining of datasets / tables using both the data step MERGE and PROC SQL. Similarities between the two methods are identified and occasions when one method might be preferred over the other are discussed. Specific issues relating to INNER, OUTER and FULL joins are covered, as are the vagaries of the SQL ON vs. WHERE clauses. An array of examples will illustrate exactly how joins are accomplished in both data step and SQL environments. Included in the presentation is a method of displaying SQL's inner workings providing hints for query optimization.

## INTRODUCTION

We all want to be able to extract data and manage it as efficiently as possible with a minimum of coding effort. Understanding how your data relates from a business perspective is key, but the different choices from a technical perspective can help avoid frustration when using SAS®.

One of the key elements to data management is comprehending how to properly join 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 opportunity for data anomalies which may occur by storing the same information in multiple tables, and, it saves disk space.

It is helpful to consider distinct entities when seeking to understand the role normalization plays in good database design. A customer is a good example of an entity. Customers have attributes which can be translated into columns within a table. A customer can have a given name, surname, age, address and a variety of other attributes which uniquely describe them. However, customers also have transactions. Consider the maintenance issues that would arise if the customer's demographic data was stored with their transaction data. If the customer's demographic data were to change, we'd have to update each and every record since the mailing address appeared on each transaction. Separating customer demographic data and transaction data will reduce data issues / maintenance and also save space.

| Given Name | Surname | Mailing Addr | City | Transaction | Date | Amount |
|---|---|---|---|---|---|---|
| George | Smith | 123 Main | Lynden | 123 | 2007-02-03 | $12.37 |
| George | Smith | 123 Main | Lynden | 4372 | 2007-04-25 | $32.98 |
| George | Smith | 123 Main | Lynden | 12234 | 2007-08-12 | $27.22 |
| George | Smith | 123 Main | Lynden | 14599 | 2007-10-31 | $76.14 |

Normalization also involves removing redundant data. Consider a table of family data containing names, ages and address information. If a family had five members, the same address information would be stored five times. With normalization the address would only be stored once and each family member would have a relationship with the one address record. The advantages of data normalization are especially evident when dealing with high-volume data stores, both in reduced maintenance and disk space savings.

When this paper uses the term "join", it refers to combining two or more tables which have a relationship or merging two or more datasets together. The paper will demonstrate the good, the bad and the ugly when it comes to this art form.

**THE DATA**

This paper will use two different sets of data to illustrate the join dynamics using data step MERGE and SQL JOIN. The initial data is a small subset ( found in the Appendix A ) used to illustrate the effect of the various joins and merges. The REPORTS section will use data of a more complex nature ( described immediately below and in Appendix B ) to demonstrate the possibilities of well normalized data and the joining mechanisms required to make use of that data.

The examples will center on a database designed for a store, something we are all familiar with. It will record data about the stores, sales people, the items available in each store, and the sales. The data tables described below have been normalized.

The first will be Stores, keeping track of the location and name of the store. The second will be the Items sold in each of the stores, storing the item name, description, price, etc. In addition to Stores and Items, it's important to record the sales people who work at the stores. Sales Person attributes include name, the store they're employed in and their employment status. Assume that one salesperson can only work at a single store. The final data entity will be the actual sale transaction which will record who made the sale, the store the sale occurred in, and the quantity sold.

The examples in this paper are simple to ensure data complexity does not obscure the illustrations that are really the point of the paper. In real life, sales systems can be very complex and sophisticated but here the intent is to clearly demonstrate join concepts and not how to build a comprehensive sales system.

A breakdown of the tables and their columns follows:.

*Stores:*

| Field Name | Description | Data Type |
|---|---|---|
| store_id | unique Identifier for the store | Numeric |
| store_name | name of the Store | Char |
| store_region | regional location of the store | Char |
| store_city | city which the store is located in | Char |

*Items:*

| Field Name | Description | Data Type |
|---|---|---|
| item_id | unique Identifier for the item | Numeric |
| desc | description of the item | Char |
| price | price of the item | Numeric |
| status | status of the item:<br>● I – In stock<br>● N – Not Available | Char |
| category_id | category of the item. eg Shoes, Shirts, Pants might be appropriate for a clothing store. Presupposes a Category table with category descriptions | Char |

*Salesperson:*

| Field Name | Description | Data Type |
|---|---|---|
| sales_id | unique Salesperson identifier. *( sales_id is intentionally different than salesperson_id from Sales table / dataset )* | Char |
| name | name of the Salesperson | Char |
| store_id | store_id where the salesperson works | Numeric |
| status | employment status of the salesperson:<br>● A – Active<br>● I – Inactive | Char |

*Sales:*

| Field Name | Description | Data Type |
|---|---|---|
| transaction | transaction ID | Numeric |
| item_id | item_id of the item sold | Numeric |
| qty | quantity of the item sold | Numeric |
| salesperson_id | sales_id of the salesperson | Numeric |
| store_id | store_id of the store where the sale occurred | Numeric |
| date | sale date | Date |
| time | sale time | Time |

The diagram in Figure 1 provides a graphical view of the tables and their relationships.
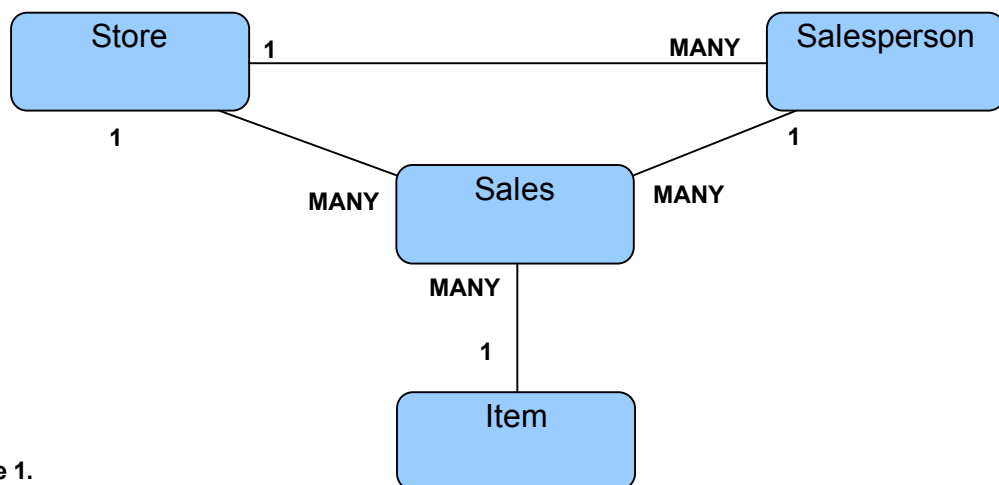


**Figure 1.**

## TYPES OF JOINS

Before embarking on the more complex reports possible using the store / purchase data, let's illustrate the different types of joins with a smaller, simpler subset of data found in Appendix A.

### *Salesperson:*

| Field Name | Description |
|---|---|
| sales_id | Salesperson identifier. |
| name | Name of the Salesperson |

### *Sales:*

| Field Name | Description |
|---|---|
| item_id | The item_id of the item sold |
| qty | Quantity of the item sold |
| salesperson_id | The sales_id of the salesperson |
| date | The date the sale happened |

### DATA DUMP OF DATA FROM APPENDIX 1.

| Salesperson name | sales_id | Sales transaction | item_id | qty | sales_id | date |
|---|---|---|---|---|---|---|
| Jim Goodnight | 1 | 1 | 37 | 4 | 5 | 2007-07-10 |
| Odious Herodias | 2 | 2 | 85 | 4 | 2 | 2007-01-26 |
| Nicole Richie | 3 | 3 | 10 | 1 | 5 | 2007-01-20 |
| Paris Hilton | 4 | 4 | 71 | 1 | 6 | 2007-08-06 |
| Caesar Augustus | 5 | 5 | 55 | 2 | 6 | 2007-06-16 |
| Daniel Flu | 6 | 6 | 38 | 4 | 2 | 2007-07-30 |
| Gord Nixon | 7 | 7 | 96 | 3 | 8 | 2007-03-09 |
| | | 8 | 17 | 4 | 4 | 2007-01-07 |
| | | 9 | 87 | 4 | 2 | 2007-08-06 |
| | | 10 | 79 | 4 | 3 | 2007-05-05 |

Upon careful examination of the sales data, it will be noted that Jim Goodnight and Gord Nixon had no sales. And, the 7th transaction is assigned to a non-existent sales_id, a value of "8". The "mismatch" of data will help us to clearly illustrate the nuances between the different join types.

Before we get to the code, the following diagram illustrates which rows from contributing tables are represented in the final data set in the different types of joins.
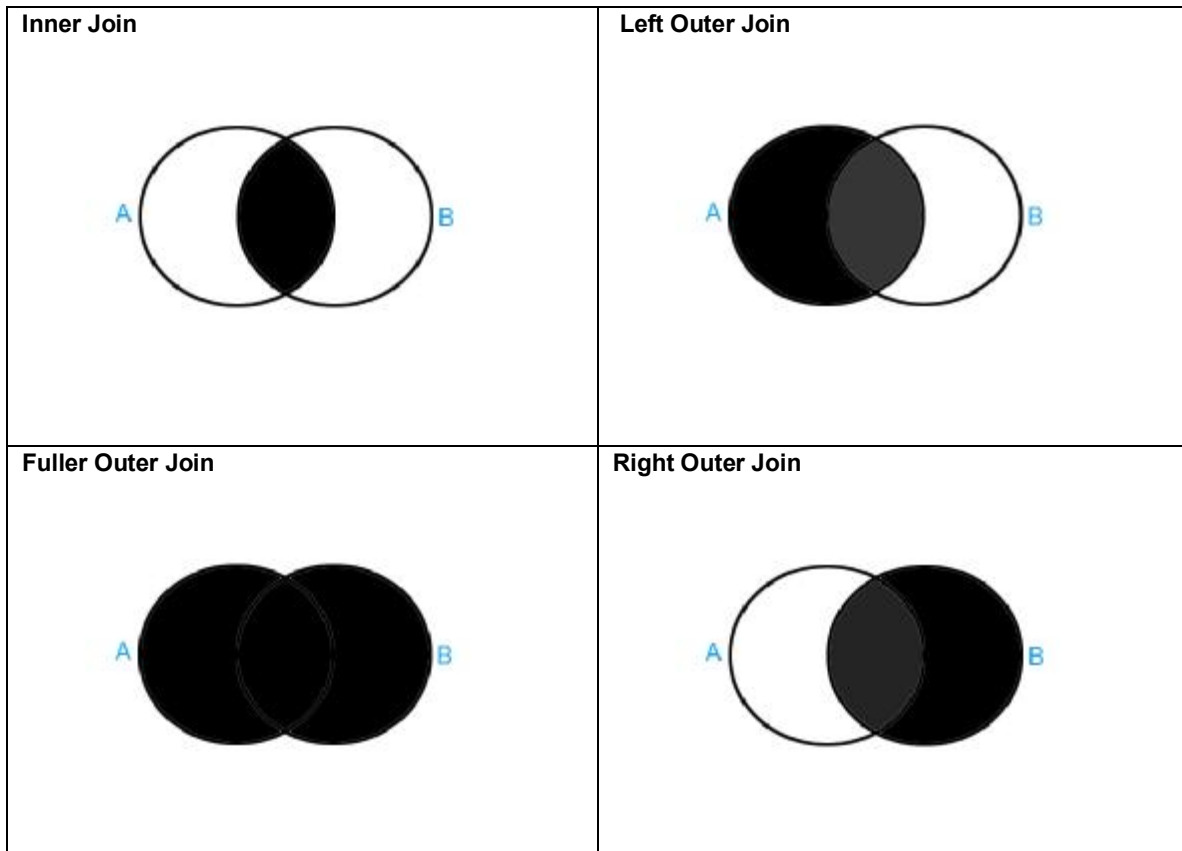
| Inner Join | Left Outer Join |
|---|---|
|  |  |
| **Fuller Outer Join** | **Right Outer Join** |
|  |  |

**Figure 2.**

### INNER JOINS

Inner joins, or equi-joins, are probably the most commonly used type of join.  Inner joins are used to connect two or more sets of information via a common value or set of common values known as keys. Typically a comparison operator, usually the equal sign (=), is used to make the connection between the keys.  To eliminate casting type errors in DATA or PROC SQL steps, the keys should be of the same data type, i.e. numeric to numeric, character to character.

Inner joins will return only the set of rows from the contributing tables that satisfy the join criteria.  In this case, this means where the same sales_id values are found on both tables.

```
proc sql;
      create table inner_sql as
        select b.transaction
           , b.item_id
           , b.qty
           , b.date
           , a.sales_id
           , a.name
          from sales      b,
               salesperson a
        where a.sales_id = b.sales_id
        order by sales_id;
quit;
```

```
proc sort data = sales;
           by   sales_id;
run;
proc sort data = salesperson;
           by   sales_id;
run;

data inner_join;
     merge sales          ( in = s )
                 salesperson ( in = p );
        by sales_id;

     if s and p;
run;
```

| transaction | item_id | qty | sales_id | date | name |
|---|---|---|---|---|---|
| 2 | 85 | 4 | 2 | 2007-01-26 | Odious Herodias |
| 6 | 38 | 4 | 2 | 2007-07-30 | Odious Herodias |
| 9 | 87 | 4 | 2 | 2007-08-06 | Odious Herodias |
| 10 | 79 | 4 | 3 | 2007-05-05 | Nicole Richie |
| 8 | 17 | 4 | 4 | 2007-01-07 | Paris Hilton |
| 1 | 37 | 4 | 5 | 2007-07-10 | Caesar Augustus |
| 3 | 10 | 1 | 5 | 2007-01-20 | Caesar Augustus |
| 4 | 71 | 1 | 6 | 2007-08-06 | Daniel Flu |
| 5 | 55 | 2 | 6 | 2007-06-16 | Daniel Flu |

**Figure 3.**

As an aside, PROC SQL syntax also allows the use of the INNER JOIN ON clause.  The results are exactly the same as using the WHERE clause but with minor syntactical differences.

## OUTER JOINS

Outer joins do **not** require a matching record from each of the contributing records.  The result set contains each record  - even if no other matching record exists - from the "driving" table(s).  Outer joins are further sub-divided into left, right and full outer joins, the left / right / full outer defining which table(s) is/are the "driving" table(s).

### LEFT OUTER JOIN

In the left join example below, the **Sales** table is the "driver", i.e. all sales will be included in the result set even if a corresponding row cannot be found in the **Salesperson** table on the right side of the join.  The result set below the code shows all sales, even the one sale attributed to sales_id 8 which is not a valid salesperson.  When one of the tables does not contribute to the join, missing values are assigned to the columns originating from that table.

```
proc sql;
     create table left_sql as
     select s.transaction
        , s.item_id
        , s.qty
        , s.date
        , p.sales_id
        , p.name
       from sales s
           left join
         salesperson p
     on    s.sales_id = p.sales_id
     order by sales_id;
quit;
```

```
proc sort data = sales;
            by  sales_id;
run;
proc sort data = salesperson;
            by  sales_id;
run;

data left_merge;
     merge sales        ( in = s )
           salesperson  ( in = p );
        by sales_id;

        if s ;
run;
```

| transaction | item_id | qty | sales_id | date | name |
|---|---|---|---|---|---|
| 2 | 85 | 4 | 2 | 2007-01-26 | Odious Herodias |
| 6 | 38 | 4 | 2 | 2007-07-30 | Odious Herodias |
| 9 | 87 | 4 | 2 | 2007-08-06 | Odious Herodias |
| 10 | 79 | 4 | 3 | 2007-05-05 | Nicole Richie |
| 8 | 17 | 4 | 4 | 2007-01-07 | Paris Hilton |
| 1 | 37 | 4 | 5 | 2007-07-10 | Caesar Augustus |
| 3 | 10 | 1 | 5 | 2007-01-20 | Caesar Augustus |
| 4 | 71 | 1 | 6 | 2007-08-06 | Daniel Flu |
| 5 | 55 | 2 | 6 | 2007-06-16 | Daniel Flu |
| 7 | 96 | 3 | 8 | 2007-03-09 | **Figure 4.** |

**RIGHT OUTER JOIN**
In the right join example below, the **Salesperson** table is the "driver", i.e. all salespersons will be represented in the result set, even if they did not make a sale.

```
proc sql;                                    proc sort data = sales;
      create table right_sql as                      by   sales_id;
      select p.sales_id                      run;
            , p.name                         proc sort data = salesperson;
            , s.transaction                          by   sales_id;
            , s.item_id                      run;
            , s.qty
            , s.date                         data right_merge;
         from sales          s                   merge sales          ( in = s )
              right join                              salesperson   ( in = p );
            salesperson  p                        by sales_id;
      on     s.sales_id = p.sales_id
      order by sales_id;                           if p ;
quit;                                        run;
```

| sales_id | name | transaction | item_id | qty | date |
|---|---|---|---|---|---|
| 1 | Jim Goodnight | . | . | . | . |
| 2 | Odious Herodias | 2 | 85 | 4 | 2007-01-26 |
| 2 | Odious Herodias | 6 | 38 | 4 | 2007-07-30 |
| 2 | Odious Herodias | 9 | 87 | 4 | 2007-08-06 |
| 3 | Nicole Richie | 10 | 79 | 4 | 2007-05-05 |
| 4 | Paris Hilton | 8 | 17 | 4 | 2007-01-07 |
| 5 | Caesar Augustus | 1 | 37 | 4 | 2007-07-10 |
| 5 | Caesar Augustus | 3 | 10 | 1 | 2007-01-20 |
| 6 | Daniel Flu | 4 | 71 | 1 | 2007-08-06 |
| 6 | Daniel Flu | 5 | 55 | 2 | 2007-06-16 |
| 7 | Gord Nixon | . | . | . | . |

**Figure 5.**

**FULL OUTER JOIN**

A full outer join result set includes all rows from all contributing tables. Where the join criteria are satisfied on all contributing tables, the result set will contain the values from all tables. As we've seen in the left and right join examples, where a table does not contribute a row to the row, the values from that table will be missing.

```
proc sql;
      create table full_sql as
      select p.sales_id
            , p.name
            , s.transaction
            , s.item_id
            , s.qty
            , s.date
         from sales  s
              full outer join
            salesperson p
      on     s.sales_id = p.sales_id
      order by sales_id;
quit;
```

```
proc sort data = sales;
             by   sales_id;
run;
proc sort data = salesperson;
             by   sales_id;
run;


data full_merge;
      merge sales           ( in = s )
            salesperson ( in = p );
            by sales_id;
run;
```

| sales_id | name | transaction | item_id | qty | date |
|----------|------|-------------|---------|-----|------|
| . | | 7 | 96 | 3 | 2007-03-09 |
| 1 | Jim Goodnight | . | . | . | . |
| 2 | Odious Herodias | 6 | 38 | 4 | 2007-07-30 |
| 2 | Odious Herodias | 2 | 85 | 4 | 2007-01-26 |
| 2 | Odious Herodias | 9 | 87 | 4 | 2007-08-06 |
| 3 | Nicole Richie | 10 | 79 | 4 | 2007-05-05 |
| 4 | Paris Hilton | 8 | 17 | 4 | 2007-01-07 |
| 5 | Caesar Augustus | 1 | 37 | 4 | 2007-07-10 |
| 5 | Caesar Augustus | 3 | 10 | 1 | 2007-01-20 |
| 6 | Daniel Flu | 5 | 55 | 2 | 2007-06-16 |
| 6 | Daniel Flu | 4 | 71 | 1 | 2007-08-06 |
| 7 | Gord Nixon | . | . | . | . |

**Figure 6.**

**CROSS JOINS**

Cross-joins, sometimes known as "Cartesian Joins", are most often a mistake. Cross joins are the set of rows created by matching every row from the contributing tables to every other row of the contributing tables. In most cases, a Cartesian product is created when the programmer neglects to specify the table join criteria correctly in SQL. The generation of a Cartesian product produces notes in the SAS log to identify the probable logic issue. Note how cross joining the two small tables in the example data results in 70 rows in the resulting dataset! Cross-joins can only be created in data step if one goes to extraordinary lengths. It is worth noting that omitting the BY statement on a MERGE will result in unexpected results as well. The use of the system option MERGENOBY=WARN is strongly encouraged.

```
proc sql;
      create table cross_sql as
      select p.sales_id
            , p.name
            , s.transaction
            , s.item_id
            , s.qty
            , s.date
         from sales        s,
            salesperson  p
      order by sales_id;
quit;
```

```
608  proc sql;
609      create table cross_sql as
610      select p.sales_id
611          , p.name
612          , s.transaction
613          , s.item_id
614          , s.qty
615          , s.date
616      from sales        s,
617              salesperson p
618      order by sales_id;
NOTE: The execution of this query involves performing one or more Cartesian product joins
that can not be optimized.
NOTE: Table WORK.CROSS_SQL created, with 70 rows and 6 columns.
```

## MERGE vs. SQL

Are there occasions when one ought to choose SQL over the data step MERGE or vice versa?  Aside from individual coding preferences and comfort levels with the different techniques, each method has pros and cons that ought to be considered when coding table joins.

| | Data Step | SQL |
|---|---|---|
| **Pros** | - Allows use of data step syntax, e.g. arrays, first. / last. processing<br>- Easier understanding and comprehension for novice users | - Sorts need not be explicitly sorted as SQL will sort behind the scenes<br>- Multiple tables can be joined on multiple keys ( even if variable types differ ) in one query<br>- Easier to deal with ranges on join, e.g. date between X and Y<br>- Join query can also summarize<br>- Pass-thru queries to RDBMS<br>- Use of efficient hash joins |
| **Cons** | - Data steps must be sorted or indexed before merging<br>- Multi-table joins on different keys requires multiple sorts/merges<br>- Post-merge summary generally requires additional steps<br>- BY variables must be same type ( char or num )<br>- Many to Many joins not handled properly | - Limitations of SQL apply, e.g. flexibility of data step syntax is lost |
| **Wash** | - Use of data set options allows variables to be dropped, renamed, kept etc.…<br>- the vast majority of SAS functions are available in both data step and SQL | |

**Figure 7.**

9

## THE REPORTS

Using the more complex data defined at the beginning of the paper, let's now look at the types of reports we can create.

To produce a report showing the number of sales at each store on December 14th 2007, it'll be necessary to list the store name, store city, item description, and quantity sold on that day, specifying the correct criteria to join the tables and limit the rows returned based on the report requirements.  In this example three tables must be joined to achieve the desired results (Stores, Sales, and Items).

| PROC SQL | DATA Step |
|---|---|
| <pre>proc sql;
  create table inner_join2 as
  select    a.store_name,
            a.store_city,
            b.desc,
            sum(c.qty) as sales
    from    stores      a,
            items       b,
            sales       c
    where   a.store_id = c.store_id
       and  b.item_id = c.item_id
       and  c.date = '14dec2007'd
    group by  1,2,3 ;
quit;

proc print data = inner_join2;
run;</pre> | <pre>proc sort data = stores;
         by   store_id;
run;

proc sort data = items;
         by   item_id;
run;

proc sort data = sales;
         by   store_id;
run;

data inner_join2;
  merge stores      (in=a)
        sales       (in=b);
    by store_id;
  if (a and b) and date = '14dec2007'd;
run;

proc sort data = inner_join2;
         by   item_id;
run;

data inner_join3
 (keep = store_name store_city
         desc qty);
  merge inner_join2 (in=a)
        items       (in=b);
     by item_id;
  if a and b;
run;


proc means data = inner_join3
  ( rename = ( qty = sales )) sum;
   class store_name store_city desc;
   var sales;
run;</pre> |

```
store_name      store_city     desc        sales

Store name 8    Hamilton    Item Desc 54       4
Store name 8    Hamilton    Item Desc 56       7
Store name 8    Hamilton    Item Desc 57       7
Store name 8    Hamilton    Item Desc 58       4
Store name 8    Hamilton    Item Desc 59       4
Store name 8    Hamilton    Item Desc 60       4
Store name 8    Hamilton    Item Desc 61      11
Store name 8    Hamilton    Item Desc 68       9
Store name 8    Hamilton    Item Desc 69       3
Store name 8    Hamilton    Item Desc 72       3
Store name 8    Hamilton    Item Desc 73       3
Store name 8    Hamilton    Item Desc 76      11
Store name 8    Hamilton    Item Desc 77       3
Store name 8    Hamilton    Item Desc 80       3
Store name 8    Hamilton    Item Desc 82       1
Store name 8    Hamilton    Item Desc 85       1
Store name 8    Hamilton    Item Desc 87       2
Store name 8    Hamilton    Item Desc 89       4
Store name 8    Hamilton    Item Desc 9        2
Store name 8    Hamilton    Item Desc 92       1
Store name 8    Hamilton    Item Desc 94       4
Store name 8    Hamilton    Item Desc 95       2
Store name 8    Hamilton    Item Desc 96       3
Store name 8    Hamilton    Item Desc 97       2
Store name 8    Hamilton    Item Desc 98       4
Store name 8    Hamilton    Item Desc 99       6
```

**Figure 8.**

In Figure 8 it is evident that extracting the data via PROC SQL is much more efficient.  Compared to the DATA step code, less code and fewer steps are required in SQL.  The DATA step solution required four sorts, two merges and a PROC MEANS step to summarize.  The PROC SQL method was complete in one step and a subsequent PROC PRINT.

A store manager might be interested in finding out which employees had no sales for a given month.   For this example we need to identify the store, region, city and name of the employees who had no sales during the month of August.  As seen a few sections earlier, outer joins help find relationships which do exist as well as ones that do not.  The query will include Stores, Salesperson and Sales tables to gather the required information.

| PROC SQL | DATA Step |
|---|---|
| <pre>proc sql;<br>  create table no_sales as<br><br>    select<br>      st.store_name,<br>      st.store_region,<br>      st.store_city,<br>      sp.name<br><br>     from<br>      Stores          st,<br>      Salesperson    sp<br>          left outer join<br>      Sales          s<br><br>    on input(sp.sales_id,5.) =<br>s.sales_id<br>      and s.date between '01AUG2007'd<br>and '31AUG2007'd<br><br>where st.store_id = sp.store_id<br>  and s.transaction is null<br>;<br>quit;</pre> | <pre>proc sort data = Stores;<br>            by   store_id;<br>run;<br><br>data salesperson_proper;<br>    set salesperson<br>   (rename=(sales_id=sales_id_txt));<br>    sales_id = input(sales_id_txt,5.);<br>run;<br><br>proc sort data = salesperson_proper;<br>            by   sales_id;<br>run;<br><br>proc sort data = Sales<br>            out = Sales_Aug;<br>            by   sales_id;<br>    where date >= '01AUG2007'd<br>    and date <= '31AUG2007'd;<br>run;<br><br>data left_join;<br>    merge<br>      salesperson_proper (IN=A)<br>      Sales_Aug          (IN=B);<br>      by sales_id;<br>    if (A and NOT B);<br>run;<br><br>proc sort data = left_join;<br>            by   store_id;<br>run;<br><br><br>data no_sales (keep=store_name<br>store_region store_city name);<br>    merge<br>      Left_join          (IN=A)<br>      Stores             (IN=B);<br>      if store_id;<br>    if A;<br>run;<br><br>/* Reorder columns  */<br>data new_no_sales;<br>  retain store_name store_region<br>store_city name;<br>  set no_sales;<br>run;</pre> |

```
   store_name      store_region    store_city      name

Store name 6     Area 4          Orlando         Ezra Levant
Store name 4     Area 2          San Antonio     Nicole Richie
Store name 5     Area 4          Atlanta         Paris Hilton
Store name 3     Area 2          Chicago         Richard Nixon
Store name 4     Area 2          San Antonio     Caesar Augustus
```

**Figure 9.**

From Figure 9 it is evident that Stores and Salesperson are combined using an inner join as noted in the WHERE clause. To identify all salespersons who did not make a sale, Salesperson must be joined to Sales using a left outer join. Remember that the outer join returns the matching and non matching rows of the "driving" table based on the conditions in the ON clause. The ON clause specified matches on sales_id which connects the two tables. Since we are only interested in August 2007, the between date range condition is necessary. If the date range condition was the only condition specified on the WHERE clause, the query would return all employees, both those who had sales in August and those who had no sales in August. The addition of the "transaction is null" condition eliminates rows where the salesperson *did* have a sale, thus returning only those salespersons who had no sales.

When the DATA Step version is compared to the SQL solution, the difference in the amount of code required is significant. DATA step requires multiple sorts, massaging the key fields to ensure variable names and types are properly matched up. Then some more sorting, a merge to bring it all back together and a final data step to re-order the variables to our liking. A little more involved and time consuming in comparison to the one step PROC SQL solution.

### INDEXES

SAS indexes provide another opportunity for efficiency when joining data. A SAS index is a separate SAS data store, akin to a Rolodex, automatically maintained by SAS. Indexes allow direct access to observations when extracting a subset of the data using the key or indexed variables. If appropriate indexes are defined to source datasets, it is not necessary to sort the datasets by the key variables before joining / merging data.

Indexes are most efficient when extracting a small subset from a large dataset. Without an index, datasets must be read sequentially to identify the observations required. Reading the entirety of a large dataset when relatively few observations are required unnecessarily consumes computing resources and results in longer run times. To realize the full efficiency of indexes, SAS recommends that they be used when the subset data is 15% or less of the entire dataset. Three ways to create a simple SAS index are illustrated below:

| Method | Syntax Example | Comments |
|---|---|---|
| PROC SQL | ```proc sql;     create index sales_id         on Salesperson(sales_id);     quit;``` | index is sales id |
| PROC Datasets | ```proc datasets library = sgf;     modify Items;     index create item_id / unique;     quit;``` | index is item id |
| DATA Step | ```data sgf.stores     (index =( store_id ));     *....;  run;``` | index is store id |

**Figure 10.**

It is also possible to define composite indexes on multiple variables.  The syntax to define composite indexes is very similar to that of simple indexes.  Once indexes are defined, SAS will use the index when joining datasets via the PROC SQL WHERE or DATA step BY statements.  See the references section at the end of this paper for an excellent treatment of the subject by Michael Raithel. His paper goes into much more detail and provides insight into this valuable feature.

## EXTRA FEATURES

### _METHOD

Have you ever wondered how PROC SQL performs joins or the algorithms being utilized in the back ground to perform the join?  The undocumented PROC SQL option **_method** provides this information by writing the SQL query plan to the log.  To use _method specify the option on the PROC SQL statement.

```
PROC SQL _method;
   <...Select statement...>;
QUIT;
```

The hierarchy of the query plan displayed in the log provides insight as to how SAS is managing the SQL query. The following table shows the most popular codes and their meaning.

| _method code | Description |
| --- | --- |
| sqxcrta | Select operation |
| sqxjsl | Step Loop Join |
| sqxjm | Merge Join |
| sqxjndx | Index Join |
| sqxjhsh | Hash Join |
| sqxsort | Sort operation |
| sqxsrc | Source Rows from table |

```
        sqxslct
            sqxjm
                sqxsort
                    sqxsrc
                sqxsort
                    sqxsrc
```

How is this hierarchy to be read and interpreted?

- typically read from right to left.
- two most right items are sqxscr operations, defining the **data source**
- sqxsort notation informs that the source data is **sorted**
- sqxjm means that the two sorted sets of data are joined via a **merge join**
- final notation at the top of the log is the **select** operation to display the results of the join.

### FEEDBACK

When writing complex queries or using embedded macro variables, it is often helpful to see the generated SQL that is executed.  This is true especially when you are receiving errors and looking to pin point an error. The **feedback** option displays the final expanded SQL statement which is ultimately executed. If asterisks (*) have been specified on the select, the actual variable names are substituted, macro variables are resolved, parentheses are added to display evaluation order, and underlying view information is expanded.  Feedback will become your best friend when debugging misbehaving SQL statements.

```
data a;                                   102
     do i = 1 to 1e5;                     103  %let lo = 3;
          j = i * 2;                      104  %let hi = 7;
          output;                         105
     end;                                 106  proc sql feedback ;
run;                                      107      select a.*, b.w
data b;                                   108        from a, b
     do i = 1 to 10;                      109       where a.i = b.i
          w = i * 33;                     110         and a.i between &lo and &hi
          output;                         111      ;
     end;                                 NOTE: Statement transforms to:
run;
                                                  select A.i, A.j, B.w
%let lo = 3;                                        from WORK.A, WORK.B
%let hi = 7;                                       where (A.i=B.i) and A.i between 3 and 7;
proc sql feedback ;
     select a.*, b.w
       from a, b                          112  quit;
      where a.i = b.i
        and a.i between &lo
        and &hi   ;
quit;
```

## BUFFERSIZE

In the **_method** example above, the query plan showed that SQL was sorting each dataset behind the scenes before performing the merge join. If the datasets are large, significant computing resources are consumed for each sort. However, it is possible to encourage SAS to forgo the sort operations in favour of a hash joins, the most efficient SAS joins. Hash joins are performed by loading at least one of the tables into memory, providing very fast access times. To influence SAS to use hash joins rather than sort/merge, another PROC SQL option, **buffersize,** can be employed. Buffersize specifies the memory allocated to the join operation. If one of the tables can be fit into memory, SAS creates a hash table of that table's keys and joins the other table(s) to it using the hash algorithm / keys. Significant time savings are realized because the tables need not be sorted. When the _method option is also specified, the log output indicates that a hash join was utilized resulting in significant execution / time savings.

```
765  proc sql _method buffersize=1e7;
766      create table master_hash as
767          select b.*, t.k
768            from backup          b,
769                 transaction     t
770           where b.i = t.i
771              ;

NOTE: SQL execution methods chosen are:

     sqxcrta
        sqxjhsh
            sqxsrc( WORK.BACKUP(alias = B) )
            sqxsrc( WORK.TRANSACTION(alias = T) )
NOTE: Table WORK.MASTER_HASH created, with 12255 rows and 3 columns.
```

## CONCLUSION

This paper has demonstrated the rationale for normalized data and the different types of joins required to bring the normalized data back together to produce meaningful results. A variety of PROC SQL and DATA step examples illustrating each type of join has identified the implementation methodologies required. From most of the examples, it's apparent that SQL joins have significant advantages over DATA step merges, in terms of coding effort, flexibility and ease of use. In almost every case, SQL requires less code and processing time.

## REFERENCES

SAS Institute Inc. 2006. **SAS OnlineDoc® 9.1.3.** Cary, NC: SAS Institute Inc.

Lafler, Kirk. 2002. "A Visual Introduction to SQL Joins" *The Twenty-Seventh Annual SAS® Users Group International Conference Proceedings*, Orlando, FL. http://www2.sas.com/proceedings/sugi27/p072-27.pdf (June 22, 2007).

Raithel, Michael A. 2005. "The Basics of Using SAS® Indexes" *Proceedings of the Thirtieth Annual SAS® Users Group International Conference*, Philadelphia, PA. http://www2.sas.com/proceedings/sugi30/247-30.pdf (January 2, 2008).

Dickstein, Craig; Pass, Ray; Davis, Michael L. 2007. "DATA Step vs. PROC SQL: What's a neophyte to do?" *Proceedings of the SAS® Global Forum 2007 Conference*, Orlando, FL. http://www2.sas.com/proceedings/forum2007/237-2007.pdf (January 5, 2008).

Cheng, Wei. 2004. "Helpful Undocumented Features in SAS[®]" *Proceedings of the Twenty-Ninth Annual SAS® Users Group International Conference*, Montreal, QC. http://www2.sas.com/proceedings/sugi29/040-29.pdf (January 5, 2008).

SAS Institute, Inc., SAS Technical Papers TS-553, Cary, NC: SAS Institute, Inc., http://support.sas.com/techsup/technote/ts553.html

## CONTACT

Your suggestions and questions are welcomed. Please contact the authors at:

Harry Droogendyk
Stratia Consulting Inc.
PO Box 145,
Lynden, ON
Canada
L0R 1T0
conf@stratia.ca
905-296-3595

Faisal Dosani
Senior Information Analyst
RBC Royal Bank
330 Front St W - 7th Floor
Toronto, ON
Canada
M5V 3B7
Faisal.Dosani@rbc.com
416-955-7654

## APPENDIX A - CODE TO CREATE SMALL TABLES FOR JOIN DEMONSTRATION

```sas
/*  Small data for Join demonstration  */

data salesperson ( drop = _: ) ;

      length name $20;
      array _name(7) $20 ('Jim Goodnight','Odious Herodias','Nicole
Richie','Paris Hilton','Caesar Augustus',
                                        'Daniel Flu','Gord Nixon') ;
      do sales_id = 1 to 7;
              name          = _name(sales_id);
              output;
      end;
run;

data sales ;
      do transaction  = 1 to 10;
              item_id       = ceil(ranuni(2)*100);
              qty           = ceil(ranuni(2)  * 4);
              sales_id      = ceil(ranuni(2)  * 5) + 1;
              date          = '02jan2007'd + ceil(ranuni(2) * 363 );
              if not(mod(transaction,7)) then sales_id = 8;  * special case ;
              output;
      end;
      format        qty    comma3.
                    date   yymmddd10.
                    ;
run;

title 'Salesperson';
proc print data = salesperson noobs;
run;

title 'Sales';
proc print data = sales noobs;
run;

proc freq data = sales;
      tables sales_id date / norow nocol nocum nopercent missing;
      format date yymmn6.;
run;
```

## APPENDIX B - CODE TO CREATE LARGE TABLES FOR REPORTS

```sas
%let num_stores = 8;

data salesperson ( drop = _: );

      length name $20;
      array _name(25) $20 ('Jim Goodnight','Odious Herodias','John Smith','Mark
Brown','Susan Jones','Harry Droogendyk',
                                        'Faisal Dosani','Lisa White','Mary
Thompson','Don Daniels','George Trifunovic','John Sharpe',
                                        'Steve Bilyea','Daniel Flu','Mats
Sundin','Darcy Tucker','Roger Clemens','Rob Bell',
                                        'Gord Sanderson','Richard Naves','Ezra
Levant','Nicole Richie','Paris Hilton',
                                            'Richard Nixon','Caesar Augustus');
```

```sas
        do _i = 1 to 25;
                sales_id      = put(_i,z5.-l);
                name          =  name(_i);
                store_id      =  'ST' || put(ceil(ranuni(1)*&num_stores),1.);
                if mod(_i,7) then status = 'A'; else status = 'I';

                output;
        end;
run;

data stores ( drop = _: );

        array _store_region(4) $6 ('Area 1','Area 2','Area 3','Area 4');
        array _store_city(8) $12 ('Toronto','Lynden','Chicago','San
Antonio','Atlanta','Orlando','Seattle','Hamilton');
        do _i = 1 to &num_stores;
                store_id          = 'ST' || put(_i,1.);
                store_name        = 'Store name ' || put(_i,1.);
                store_region = _store_region(ceil(ranuni(1)*4));
                store_city        = _store_city(_i);
                output;
        end;
run;

data items;
        do item_id = 1 to 100;
                desc = 'Item Desc ' || put(item_id,3.-l);
                price = ranuni(1) * 97;
                if mod(item_id,7) then status = 'A'; else status = 'N';
                category = 'Cat ' || put(ceil(ranuni(1)*9),1.);
                output;
        end;
        format price dollar12.2;
run;

/*  Missing 'Cat 5' to demonstrate left join  */
data categories ( drop = _: );
        do _i = 1 to 4, 6 to 9;
                category         = 'Cat ' || put(_i,1.);
                category_desc= 'Category Desc ' || put(_i,1.);
                output;
        end;
run;

/*  Have to get the right store_id for the sales_id by joining to salesperson
*/
data sales_int ;
        do transaction  = 1 to 1e5;
                item_id    = ceil(ranuni(2)*100);
                qty        = ceil(ranuni(2) * 4);
                sales_id   = ceil(ranuni(2) * 20);          * 5 less than
actual number sales persons, for outer join stuff;
                date       = '02jan2007'd + ceil(ranuni(2) * 363 );
                output;
        end;
        format      qty    comma3.
                    date   yymmddd10.
                    ;
run;

proc sql;
        create table sales as
                select s.*, store_id
```

```sas
                from sales_int          s
                        left join
                    salesperson         p
                  on s.sales_id = input(p.sales_id,5.)
        ;
quit;

proc print data = salesperson;
run;

proc print data = stores;
run;

proc print data = items;
run;

proc print data = categories;
run;

proc print data = sales ( obs = 100 );
run;

proc freq data = sales;
       tables store_id sales_id date / norow nocol nocum nopercent missing;
       format date yymmn6.;
run;

proc freq data = items;
       tables category*status / norow nocol nocum nopercent missing;
run;
```