Paper 138-2010

PROC DATASETS;

The Swiss Army Knife of SAS® Procedures

Michael A. Raithel, Westat, Rockville, MD

ABSTRACT

The DATASETS procedure provides the most diverse selection of capabilities and features of any of the SAS procedures. It is the prime tool that programmers can use to manage SAS data sets, indexes, catalogs, etc. Many SAS programmers are only familiar with a few of PROC DATASETS's many capabilities. Most often, they only use the data set updating, deleting, and renaming capabilities. However, there are many more features and uses that should be in a SAS programmer's toolkit.

This paper highlights many of the major capabilities of PROC DATASETS. It discusses how it can be used as a tool to update variable information in a SAS data set; provide information on data set and catalog contents; delete data sets, catalogs, and indexes; repair damaged SAS data sets; rename files; create and manage audit trails; add, delete, and modify passwords; add and delete integrity constraints; and more. The paper contains examples of the various uses of PROC DATASETS that programmers can cut and paste into their own programs as a starting point. After reading this paper, a SAS programmer will have practical knowledge of the many different facets of this important SAS procedure.

INTRODUCTION

Most people have some familiarity with the Swiss Army knife (www.swissarmy.com). Swiss Army knives resemble ordinary pocket knives, and usually have the two knife blades that common pocket knives have. So, you can use a Swiss Army knife to perform normal tasks such as cutting or whittling. But, Swiss Army knives frequently also include a plethora of additional fold-out gadgets such as a screwdriver, scissors, can opener, corkscrew, saw, etc. You can fix a loose screw, snip string or paper, open a can, open a wine bottle, or saw something into pieces; as well as cut or whittle. So, Swiss Army knives provide much more functionality and utility than ordinary pocket knives do.

The same holds true for the DATASETS procedure. PROC DATASETS allows you to perform the basic functions of renaming, copying, deleting, aging, and repairing SAS data sets. But, it provides features and facilities for doing much, much more. Some of the features are very specialized and obscure, so you are not likely to use them very often. Others are more mainstream and will become a part of your normal programming tool set. Whether obscure or mainstream, it is good for you to know that the DATASETS procedure has a wide range of utilities that you can bring to bear on a variety of tasks related to SAS data sets.

There are many ways that one could go about organizing the functions provided by PROC DATASETS. The way that this paper is organized is to divide the DATASETS procedure's functionality into four main categories:

1. **Obtaining SAS Library Information**. The CONTENTS statement provides you with the means to list the files in a SAS library and determine their characteristics. Executing the CONTENTS statement is a good starting point for understanding the nature of the files in a SAS library before considering how you might modify them.

2. **Modifying Attributes of SAS Variables**. This PROC DATASETS capability allows you to make changes to SAS data set metadata at very little cost in terms of computer resources. This is one of the more popular uses for the DATASETS procedure, and one that you will definitely want to have in your SAS toolkit.

3. **Modifying Attributes of SAS Data Sets**. This group of PROC DATASETS statements allows you to perform tasks that directly affect the structure and functionality of SAS data sets. Many of these statements involve more advanced data set structures, so you may not find yourself using them very often. However, you should be aware that the DATASETS procedure can perform these tasks when you need to accomplish them in your SAS programs. You can use these statements to:

    - Concatenate SAS data sets using the APPEND statement

- Manage audit trails using the AUDIT statement
- Manage integrity constraints using the IC statements
- Manage indexes using the index statements
- Change file attributes using the MODIFY statement
- Recover indexes and integrity constraints using the REBUILD statement

4. **Managing Files in SAS Libraries**. This collection of DATASETS procedure statements facilitates the processing of all types of files within SAS data libraries. Some of these actions, such as COPY-ing and DELETE-ing will be very familiar to many SAS programmers because they are widely used. Others, such as EXCHANGE-ing and SAVE-ing, are less frequently used, but are good to have when you need them. This group of DATASETS procedure statements permit you to:

- Cascade file renames using the AGE statement
- Rename SAS files using the CHANGE statement
- Copy files using the COPY, SELECT, and EXCLUDE statements
- Permanently remove files using the DELETE statement
- Swap file names using the EXCHANGE statement
- Fix damaged files using the REPAIR statement
- Keep files during a delete operation using the SAVE statement

One thing that you should note is that the DATASETS procedure only acts upon existing SAS files. It can manage the metadata of an *existing* SAS data set, manage features of *existing* SAS data set files, or manage *existing* SAS files in *existing* data libraries. Consequently, PROC DATASETS is used after-the-fact; after a SAS file has been created in a DATA step, with PROC SQL, or with some other SAS procedure. Except for COPY-ing, PROC DATASETS does not produce new SAS data sets. So, your use of the DATASETS procedure will primarily be to modify the features of existing SAS data sets or other members of data libraries.

The following sections provide the information that you need to make the DATASETS procedure an integral part of your SAS programming repertoire.

BRIEF OVERVIEW OF PROC DATASETS SYNTAX

Before looking at the many ways you can use the DATASETS procedure, let's take a look at its basic syntax. PROC DATASETS takes the following basic form:

```
proc datasets <option-1> <…option-n>;

<<PROC DATASETS Statements>>

quit;
```

The PROC DATASETS statement identifies the SAS data library containing the SAS files you want to modify. It is followed by one or more "RUN groups", and a "QUIT" statement that ends the execution of the procedure.

A "RUN group" is a series of PROC DATASETS sub-statements that perform a particular function. Each RUN group executes separately, in the order in which it appears, and completes its work before the next RUN group is executed. All RUN groups begin with a particular statement and some—but not all—end with a RUN statement. You can have multiple RUN groups within a particular invocation of PROC DATASETS.

Here is an example of several RUN groups within a single invocation of PROC DATASETS:

```
proc datasets library=sgflib;

modify snacks;
        format price dollar6.2 ;
        informat date mmddyy10.;
run;

append base=snacks data=newsnacks;

change newsnacks = oldsnacks;
```

```
copy out=archive;
        select oldsnacks / memtype =data;
run;

quit;
```

In the example above, there are five RUN groups. The first RUN group is the PROC DATASETS statement, which executes immediately. The second begins with the MODIFY statement, the third with the APPEND statement, the fourth with the CHANGE statement, and the fifth with the COPY statement. Each RUN group performs a specific function, and note that only two of them (MODIFY and COPY) end with an actual RUN statement.

PROC DATASETS considers the following PROC DATASETS statements to be RUN groups:

- The PROC DATASETS statement itself
- The MODIFY statement and its subordinate statements
- The APPEND, CONTENTS, and COPY statements—each being its own RUN group
- The AGE, CHANGE, DELETE, EXCHANGE, REPAIR, and SAVE statements—SAS treats multiple consecutive occurrences of any of these statements as a single RUN group

So, when coded, each of these RUN groups executes separately, in sequence, and performs the specified tasks to one or more SAS files in a particular SAS data library. For more information on DATASETS procedure RUN groups, refer to the SAS procedures guide reference specified at the end of this paper.

There are twelve options that may be used in the PROC DATASETS statement:

- **ALTER** – You can use this to specify an alter password for alter-protected files in the library.
- **DETAILS | NODETAILS** – These options specify whether SAS is to write the following to the SAS log:
  - **Obs, Entries, or Indexes –** For SAS data sets, catalogs, and indexes, respectively
  - **Vars –** The number of variables in a data set, view, or audit file
  - **Label –** SAS data set labels
- **FORCE –** Forces RUN groups to run even if there are errors in some of the statements. Also, if the APPEND statement is executed, it forces the concatenation of the two data sets when there are discrepancies in the variables.
- **GENNUM = ALL | HIST | REVERT | *integer* –** This specifies that processing is to be for specific files in generation group. See the DELETE statement in a subsequent section for a more detailed explanation of the possible values of this option.
- **KILL –** This option deletes all files in the SAS data library. Its behavior can be modified via the MEMTYPE option to only delete all of a certain type of file. *Be very, very careful when using this option!*
- **LIBRARY –** This option is used to indicate the SAS library that is going to have its files processed. If it is not specified, the WORK or USER library is used.
- **MEMTYPE –** The MEMTYPE option designates the type of SAS file that is to be processed by the procedure. The default is ALL file types.
  NOLIST – Stops SAS from printing a directory list of all of the library's files in the log. Since directory information is easily obtainable via other means, many programmers specify NOLIST to have a cleaner SAS log.
- **NOWARN –** This option suppresses errors and warnings from the CHANGE, COPY, DELETE, EXCHANGE, REPAIR and SAVE statements. It is dangerous to use this option, because if you do not get the results that you want, you will not be able to refer back to the SAS log to see exactly what happened.
- **PW –** Specifies an ACCESS, READ, or WRITE password. See the section on the PASSWORD statements, later in this paper, for more information about the various types of passwords.
- **READ –** Provides the READ password for files protected with a READ password.

It is not practical to cover all of the many nuances of PROC DATASETS's options in this paper, so the simple explanations above will have to suffice. For more detailed information, refer to the PROC DATASETS chapter in the **Base SAS 9.2 Procedures Guide**, listed in the References section of this paper.

OBTAINING SAS LIBRARY INFORMATION

The CONTENTS statement, like the CONTENTS procedure, can be used to list the directory of a SAS library, or list specific information for one or more SAS data sets. The basic format of the CONTENTS statement is:

CONTENTS <option-1 <…option-n>>;

There are over a dozen options that may be specified for the CONTENTS statement, so it is not practical to go into detail on each one of them. Instead, we will look at the ones that are most commonly used. Should the need arise, you can look up the rest of them in the **CONTENTS Procedure** chapter of the **Base SAS 9.2 Procedures Guide**, listed in the References section of this paper. Some of the more useful options are:

- **DATA** – Identifies the SAS data set that you want information on
- **OUT** – Only used if you want to write the output to a data set
- **DETAILS | NODETAILS** – Specifies whether the library section of the output includes data set labels, as well as the number of observations, variables, and indexes
- **DIRECTORY** – Output a list of all of the SAS files in the SAS data library
- **MEMTYPE** – Allows you to only output information for a specific SAS file type
- **NODS** – Stops the output of information on individual files
- **SHORT** – Creates an abbreviated output

Here is an example of the CONTENTS statement:

```
proc datasets library=sgflib;

contents data=bweight details varnum memtype=data;

run;
quit;
```

This example creates a listing for the BWEIGHT data set, ordering the list of variables by their position within observations. It also creates a detailed list of the SGFLIB SAS library directory, showing the number of entries or observations in SAS files, file labels, file sizes, indexes, etc.

The CONTENTS statement in the DATASETS procedure provides an alternative to the CONTENTS procedure that you may find convenient to use.

MODIFYING ATTRIBUTES OF SAS VARIABLES

It is not uncommon for SAS programmers to come across a SAS data set that needs to have changes made to one or more variable's formats, informats, labels, or even names. Perhaps the SAS data set was created by somebody else, or perhaps the programmer created the SAS data set at a time when that particular information was not available. Whatever the reason, once the proper values for formats, informats, labels, and variable names are known, changes must be made to the SAS data set to reflect those values.

Beginning SAS programmers often make the mistake of re-creating the entire SAS data set, just to change the value of one or more formats, informats, labels, or variable names. Such a program might look like this:

```
data sgflib.snacks;
set  sgflib.snacks;

       format price dollar6.2
              date  worddate.
              ;
       informat date mmddyy10.;
       label    product = "Snack Name"
                date    = "Sale Date"
                ;
       rename Holiday = Holiday_Sale;

run;
```

Though the program above *does* fix issues with the formats, informats, labels, and names for the variables in the SNACKS SAS data set, it is not very efficient to run. It is inefficient because it reads the entire SNACKS SAS data set and creates a new copy of it, simply to fix data set metadata. If SNACKS is a small data set, then not much I/O,

CPU time, and wallclock time are consumed. However, if SNACKS is big, then a lot of computer resources are consumed for several simple metadata changes.

SAS stores all of the metadata for a particular SAS data set in the descriptor portion of the data set, which is commonly stored in the first physical page of the SAS data set file. The DATASETS procedure can be used to update this information by reading only the data set's descriptor page. So, instead of reading the entire SAS data set, it only reads the first page, updates the format, informat, label, or variable name information, and saves that first page. Consequently, it is much more efficient to use PROC DATASETS to update such information.

You can use the DATASETS procedure to execute the following statements that modify SAS data set metadata:

- **ATTRIB** – This statement allows you to specify the format, informat, or label statements for one or more variables.
- **FORMAT** – This statement lets you to assign formats to variables.
- **INFORMAT** – This statement permits to you assign informats to variables.
- **LABEL** – This statement allows you to create variable labels.
- **RENAME** – This statement lets you rename variables.

Here is an example of using PROC DATASETS to update the same information updated in the DATA step above.

```
proc datasets library=sgflib;
modify snacks;
        format price dollar6.2
               date  worddate.
               ;
        informat date mmddyy10.;
        label product = "Snack Name"
              date    = "Sale Date"
              ;
        rename Holiday = Holiday_Sale;

run;
quit;
```

The first line specifies the DATASETS procedure and specifies the SAS data library SGFLIB, where the data set (SNACKS) that is to be modified can be found. The MODIFY statement specifies that the SNACKS data set will have some of its metadata modified. Thereafter the FORMAT, INFORMAT, LABEL, and RENAME statements are executed to modify the attributes of the PRICE, DATE, PRODUCT, and HOLIDAY variables, respectively.

The ATTRIB statement can be used to modify the FORMAT, INFORMAT, or LABELs for *multiple* variables. Here is an example:

```
proc datasets library=sgflib;
modify snacks;

        attrib QtySold Price Advertised label="";

run;
quit;
```

In this example, the labels for the QTYSOLD, PRICE, and ADVERTISED variables have been removed. The ATTRIB statement is a good tool for modifying the attributes of multiple variables with a single statement.

You can remove the FORMATS, INFORMATS, and LABELS from all variables in a data set with an ATTRIB statement. Here is an example:

```
proc datasets library=sgflib;
modify snacks;

        attrib _all_ format=;
        attrib _all_ informat=;
        attrib _all_ label="";
```

```
run;
quit;
```

In the example, above, all FORMATs, INFORMATs, and LABELS were removed from the SNACKS SAS data set. This is obviously a powerful tool that you need to use carefully!

MODIFYING ATTRIBUTES OF SAS DATA SETS

The DATASETS procedure provides over a half-dozen tools that you can use to modify the structure and functionality of individual SAS data sets. Several of these, such as the AUDIT statement, the IC (integrity constraint) statements, and the INDEX statements, create, activate, or delete additional SAS files that are closely associated with the original SAS data set. Others, such as the APPEND and MODIFY statements, actually change the contents of the SAS data set and change the attributes of the data set, respectively. We will look at the APPEND, AUDIT, MODIFY, and REBUILD statements separately, and the IC and INDEX statements together as groups of statements.

Concatenating SAS Data Sets with the APPEND Statement

The APPEND statement in PROC DATASETS performs the same function that the APPEND procedure does. It concatenates one SAS data set to the "bottom" of another. Like PROC APPEND, you must specify the BASE= SAS data set—the one being appended to—and the DATA= SAS data set—the one whose observations are being appended. After the DATASETS procedure has completed a successful execution of the APPEND statement, the BASE= SAS data set has been modified so that all of the observations in the DATA= SAS data set are now concatenated to the bottom of it. Consequently, the BASE= SAS data set contains both its original observations plus those from the appended SAS data set, while the DATA= SAS data set—whose observations were appended--remains unchanged by the procedure.

Appending one data set to another is more efficient than using a DATA step to concatenate two data sets. During the append, the observations in the BASE= SAS data set do not need to be read. Instead, SAS reads the observations from the DATA= SAS data set and writes them at the end of the BASE= SAS data set. This updates the BASE= SAS data set *in place*, avoiding the computer resources that would otherwise be used in reading the BASE= SAS data set.

Here is an example of the APPEND statement in its simplest form:

```
proc datasets library=sgflib;

append base=snacks
       data=snacktran;
quit;
```

In the example, above, the SNACKTRAN SAS data set is appended to the SNACKS SAS data set. The log for this example looks like this:

```
NOTE: Appending SGFLIB.SNACKTRAN to SGFLIB.SNACKS.
NOTE: There were 3066 observations read from the data set SGFLIB.SNACKTRAN.
NOTE: 3066 observations added.
NOTE: The data set SGFLIB.SNACKS has 44968 observations and 6 variables.
```

The append operation completed successfully because both the BASE= and the DATA= SAS data sets had the same variables with same data types and the same lengths. If there were conflicts between some of these data set attributes, the append may not have worked and you would have received a message such as:

```
NOTE: Appending SGFLIB.SNACKTRAN to SGFLIB.SNACKS.
WARNING: Variable newprod was not found on BASE file. The variable will not be added to
the BASE file.
ERROR: No appending done because of anomalies listed above.
       Use FORCE option to append these files.
NOTE: 0 observations added.
NOTE: The data set SGFLIB.SNACKS has 44968 observations and 6 variables.
NOTE: Statements not processed because of errors noted above.
NOTE: The SAS System stopped processing this step because of errors.
```

When the appending SAS data set contains variables not found in the BASE= data set or variables of different lengths or data types, the append operation does not take place. You can overcome some of these issues by using

the FORCE option, which is described below. It is impractical for this paper to cover every way in which the two data sets may be different and what would happen to a particular attempt to append. For more information, refer to the APPEND procedure in the SAS Online Documentation.

The APPEND statement has four options:

- **APPENDVAR=V6** – This specifies that SAS is to append one observation at a time to the BASE= SAS data set instead of appending blocks of data at a time (using the *"block I/O method*") that came into being with SAS v7 and later. Generally, you do not want to specify this option, as it leads to slower append execution times. It is often used in circumstances where data is being appended to an indexed SAS data set that has a unique index and the appending data might have non-unique key variable values. In such a case, SAS rejects observations with non-unique key variable values and does not append them. Refer to the aforementioned PROC APPEND documentation for more guidance on this option.

- **FORCE** – This option tells SAS to append a data set containing variables that are either not in the BASE= data set, do not have the same type as ones in the BASE= data set, or have lengths longer than those in the BASE= data set. Note that the characteristics of the BASE= data set trump those of the data set being appended. So, variables in the appending data set:

    - that are not found in the BASE= data set get dropped
    - that have different data types get set to missing.
    - that have longer lengths get truncated

- **GETSORT** – In cases where you are appending a sorted SAS data set to a BASE= SAS data set *with zero observations*, this option copies the sort information (that PROC SORT stored in the appending SAS data set) to the BASE= data set. So, a subsequent CONTENTS of the updated BASE= SAS data set will show that the data set is sorted.

- **NOWARN** – This option suppresses warnings in the SAS log when you use the FORCE option and variables in the two data sets have different characteristics. It is best to not use this option, so that you are aware of mismatched variable characteristics.

Here is an example of the previous DATASETS procedure with all of the options specified:

```
proc datasets library=sgflib;

append base=sgflib.snacks
       data=sgflib.snacktran
       appendver=v6 force getsort nowarn;
quit;
```

This example is for illustrative purposes only; you would not really want to specify these options in this circumstances. APPENDVAR is not needed since the SNACKS SAS data set is not indexed. The FORCE option is not needed since both data sets have the same variables with the same data types and lengths in them. The GETSORT option will not work because the SNACKS data set does not contain zero observations, it contains 35,770 observations. And, we do not want to specify the NOWARN option because we want warning messages written to the SAS log.

Managing Audit Trails with the AUDIT Statement

The AUDIT statement is used to facilitate using an audit trail for a particular SAS data set. An audit trail is a special SAS file that you can create to keep track of which observations are added, deleted, or modified in a SAS data set. By creating an audit file, you can determine who modified the data set, when it was altered, and what was changed. You can use audit trails for data security purposes, to review past changes made to data, and to roll changes back to previous values.

When you create an audit trail for a SAS data set, SAS automatically creates a new file with the same name as the original SAS data set, but with the file extension of ***.sas7baud***. For example, if you create an audit trail for the SNACKS SAS data set, the audit trail file will be named SNACKS.sas7baud. The audit trail file is created in the same directory as the original SAS data set. It remains there until you use the TERMINATE option in the DATASETS procedure, at which time it is deleted and auditing ceases for the specified SAS data set.

You can use the AUDIT statement to create, suspend, resume, or terminate an audit trail. Here is an example of creating an audit file for the SNACKS SAS data set:

```
proc datasets library=sgflib nolist;

        audit snacks;
               initiate;
               log admin_image=yes
                   before_image=yes
                   data_image=no
                   error image=yes;
               user_var update_reason $15;
        run;

        quit;
```

In this example, the LOG option was used to specify the four possible audit settings:

- **admin_image** – States whether or not the SUSPEND and RESUME administrative actions are logged to the audit file.
- **before_image** – States whether the before-image of updated observations are recorded to the audit file.
- **Data_image** – States whether the after-image of added, updated, and deleted observations are recorded to the audit file.
- **Error_image** – States whether the error images are recorded in the audit file.

You may specify a *YES* or *NO* for any one of the LOG option images above. However, note that all of them default to YES. So, if you decide not to code the LOG option, all of the images will automatically be set to YES.

The USER_VAR option allows you to create a new variable that is stored in the audit trail file. Most programmers use such variables to record *why* a change was made to a particular observation. For example the following PROC SQL code inserts a new row into the SNACKS table:

```
        proc sql;
           insert into sgflib.snacks
               set product = 'Snake Snacks',
                   qtysold = 20890,
                   price = 2.5,
                  advertised=0,
                  holiday=0,
                  date=18379,
                 update_reason = "Add new product";
        quit;
```

The variables PRODUCT, QTYSOLD, PRICE, ADVERTISED, HOLIDAY, and DATE exist in the SNACKS table and values are provided for the new row. However, the variable UPDATE_REASON only exists in the SNACKS audit trail data set. When the new row is added to the SNACKS table, the UPDATE_REASON for adding the new row will be saved in the audit trail file. After inserting the row, you can print the resulting audit file entry with the SQL procedure. Here is an example:

```
        options linesize=150;

        proc sql;
        select product,
                 update_reason,
                 _atopcode_,
                 _atuserid_ format=$9.,
                 _atdatetime_
               from sgflib.snacks(type=audit);
        quit;
```

This example prints several, but not all, audit file variables, resulting in the following listing:

```
Product            update_reason     _ATOPCODE_  _ATUSERID_      _ATDATETIME_
-------------------------------------------------------------------------------
Snake Snacks       Add new product    DA         RAITHEL_M    09JAN2010:15:43:52
```

You can see that the UPDATE_REASON supplied in the previous PROC SQL step was recorded in the audit trail file. So was the userid of the person making the change and the date/time that the change was made. The _ATOPCODE_ value of DA specifies that the record was added to the SNACKS data set. You can find all possible

values for _ATOPCODE_ in the section **Understanding Audit Trails** in the **SAS 9.2 Language Reference: Concepts** online documentation cited in the References section of this paper.

To print all fields and all records in the audit file, simply execute the following:

```
proc sql;
  select * from sgflib.snacks(type=audit);
quit;
```

You can determine which fields are in a SAS audit trail data set via the CONTENTS procedure. Here is an example:

```
proc contents data=sgflib.snacks(type=audit);
run;
```

Here is the *Alphabetic List of Variables and Attributes* from the CONTENTS output:

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Format
3	Advertised	Num	8	
5	Date	Num	8	
4	Holiday	Num	8	
2	Price	Num	8	
6	Product	Char	40	
1	QtySold	Num	8	
8	_ATDATETIME_	Num	8	DATETIME19.
13	_ATMESSAGE_	Char	8	
9	_ATOBSNO_	Num	8	
12	_ATOPCODE_	Char	2	
10	_ATRETURNCODE_	Num	8	
11	_ATUSERID_	Char	32	
7	update_reason	Char	15	

For more information on the set of *"_AT…"* variables found in an audit file data set, refer to the section **Understanding Audit Trails** in the **SAS 9.2 Language Reference: Concepts** online documentation cited in the References section of this paper.

There are several other important AUDIT options that you should be aware of:

- **audit_all** – States whether audit log settings may be changed and whether auditing may be suspended in the future. Specifying YES means that you cannot use the SUSPEND option in the future, nor can you use the LOG option to turn off logging for various images. It is best to use the default of NO and not specify this option unless you have a good reason for specifying YES.
- **suspend** – Stops SAS from logging changes to the audit file.
- **resume** – Directs SAS to resume logging changes to the audit file. It is generally used after a SUSPEND option has stopped logging changes.
- **terminate** – Terminates logging to an audit file and deletes the audit file. Be careful when using the TERMINATE option. If you might want to inspect the audit file at a future time, it is best to use the SUSPEND option, which merely suspends use of auditing, and keeps the audit file. The TERMINATE option deletes the audit file and there is no way for SAS to recover it.

Though audit files can be great for security, for understanding the history of changes, and even for rolling back changes, they do carry a price. When a change is made to the original SAS data set additional computer resources are needed to update the audit file. So, it takes longer, requires more I/O's, and consumes more CPU time to make updates, deletes, and adds to SAS data sets with audit trails enabled. But, if keeping track of the changes made to critical SAS data sets is important to your organization, SAS audit trails are a great tool to use.

Managing Integrity Constraint with the IC Statements

The three IC statements, IC CREATE, IC DELETE, and IC REACTIVATE, are used to facilitate the use of *integrity constraints* on SAS data sets. Integrity constraints are built-in data set validation rules that have their roots in the world of SQL programming. They are a set of rules used to restrict the values stored in variables in SAS data sets.

You can create integrity constraints (rules) that limit the values that can be stored in variables in a SAS data set. SAS then enforces those rules whenever observations are added, modified, or deleted from the data set.

There are two major categories of integrity constraints:

1. **General integrity constraints**. General integrity constraints exist for the variables within a single file. They consist of the following four constraints:

   - **check** – Limits the values in a variable to a range, set, or list of values. You can also limit the values of a variable depending upon the value of another variable in the same observation. For instance, if Gender equals "Male", then Pregnant must equal "No".
   - **not null** – Specifies that a variable cannot contain missing values
   - **primary key** – States that all occurrences of this variable in the SAS data set must be unique. There can only be one *primary key* variable for a given SAS data set. Customer ID, Part Number, and Social Security Number are all examples of typical *primary key*s.
   - **unique** – Specifies that all occurrences of this variable must have unique values within the data set. This is similar to the *primary key* constraint. However, there may be many variables with the *unique* constraint, but only one with the *primary key* constraint.

2. **Referential integrity constraints**. Referential integrity constraints exist between two or more SAS data sets. This happens when a *primary key* integrity constraint in one data set is referenced by a *foreign key* integrity constraint in another data set. Three types of referential integrity constraint actions can be defined for either update or delete operations:

   - **cascade** – Allows primary key variables to be updated, which results in automatically updating the values of the corresponding foreign *key* variables to the same values in the matching *foreign key* data file.
   - **restrict** – This action stops *primary key* variables from being updated or deleted if there is a matching value in a *foreign key* variable in the matching *foreign key* data file.
   - **set null** – Allows *primary key* variables to be updated or deleted and sets the values of the *foreign key* variables in the matching *foreign key* data file to missing values (null).

When you specify a *unique*, *primary key*, or *foreign key* integrity constraint, SAS automatically creates an index file to store and keep track of the values. The index file has the same name as the original SAS data set, but with the file extension of ***.sas7bndx***. For example, for the SNACKS SAS data set, the index file would be named SNACKS.sas7bndx. The index file is created in the same directory as the original SAS data set. If an index file already exists for a data set because it has indexes, integrity constraint entries are simply inserted into it when the integrity constraints are defined. SAS automatically updates the integrity constraint entries in the index file as observations are added, modified, or deleted. The index file is automatically deleted by SAS when all integrity constraints (and all indexes) for a particular SAS data set have been removed.

There is a lot more to know about integrity constraints, and this paper cannot do more than provide some very basic concepts. For more information, refer to the section **Understanding integrity constraints** in the **SAS 9.2 Language Reference: Concepts** online documentation cited in the References section of this paper.

Creating Integrity Constraints

The IC CREATE statement can be used to create integrity constraints. The format of the IC CREATE statement is:

IC CREATE <*constraint-name*=> constraint <MESSAGE='*message string*' <MSGTYPE=USER>>

In the form above:

- ***<constraint-name=>*** – You may decide to provide a constraint-name to a constraint if you wish

- **Constraint** – This can be any one of the following:

  - **NOT NULL**(*variable*) – Variable cannot contain any type of SAS missing value
  - **UNIQUE***(variables)* or **DISTINCT**(*variables*) – Values of the specified variable or variables must be unique throughout the entire SAS data set
  - **CHECK**(WHERE-*expression*) – Values must adhere to a specific range, list, or set specified by the WHERE expression
  - **PRIMARY KEY***(variables)* – Values of the variable or variables must be unique for the entire SAS data set. This can only be specified once per SAS data set.

  - **FOREIGN KEY***(variables)* REFERENCES *table-name* <ON DELETE *referential-action*> <ON UPDATE *referential-action*> – For *foreign key*s, you need to specify:
    - **variables** – The variables in the current table that are *primary key*s in the table that you want SAS to link to
    - **table-name** – The name of the other table that you want SAS to link to
    - **<ON DELETE *referential-action*>** - The action to take when the corresponding Primary Key is deleted from the other table. There are three referential actions: RESTRICT, CASCADE, SET NULL – See explanation in the previous section.
    - **<ON UPDATE *referential-action*>** – The action to take when the corresponding Primary Key is updated in the other table. There are three referential actions: RESTRICT, CASCADE, SET NULL – See explanation in the previous section.

- **<MESSAGE=*'message string*' <MSGTYPE=USER>>** – You may provide text that will be written into the SAS error message when the integrity constraint is violated. By coding the optional MSGTYPE=USER, you will suppress the SAS error message, so that only the integrity constraint violation message that you have specified is output.

Here is an example of creating integrity constraints for the SHOES SAS data set:

```
proc datasets library=sgflib nolist;

modify shoeregions;
       ic create primkey = primary key (region);
run;

modify shoes;
       ic create pkey = primary key (sequenceno);
       ic create regprodsub = distinct (region product subsidiary)
                  message = "Region, Product, Subsidiary combination must be unique";
       ic create storelimit = check(where=(stores < 50))
                  message = "Limit of 50 stores";
       ic create returnsales = check(where=(returns+sales < inventory))
                  message = "Returns + Sales cannot exceed Inventory";
       ic create fkey = foreign key (region) references sgflib.shoeregions
                        on update cascade on delete set null;
run;

quit;
```

In the example, above, we first create a *primary key* (REGION) integrity constraint in the SHOEREGIONS SAS data set. This will only work if all values of REGION are unique in that data set. Secondly, we create five integrity constraints for the SHOES SAS data set. Note that we provided a name for each of them. The five integrity constraints are:

- **pkey** – States that variable SEQUENCENO is the primary key for the SHOES SAS data set. So, all values of SEQUENCENO must be unique within the data set.
- **regprodub** – Specifies that the combined values of REGION, PRODUCT, and DISTRICT must be unique (DISTINCT) for all observations within the SHOES SAS data set. Note, that we could have used the UNIQUE keyword instead of the DISTINCT keyword.
- **storelimit** – Limits the value of STORES to less than fifty for all observations in the data set.
- **returnsales** – Limits the value of RETURNS plus SALES to be less than the value of INVENTORY for each observation.
- **fkey** – Specifies that REGION is a *foreign key* for the SHOEREGIONS data set. When REGION is updated in SHOEREGION, that value is changed ("cascaded") in the SHOES data set. If an observation is deleted from SHOEREGIONS, then the corresponding value of REGION is set to missing values ("null") in the SHOES data set.

The log for the program above looks like this:

```
18   proc datasets library=sgflib nolist;
19
20   modify shoeregions;
21
22   ic create primkey = primary key (region);
```

```
NOTE: Integrity constraint primkey defined.
23
24   run;
NOTE: MODIFY was successful for SGFLIB.SHOEREGIONS.DATA.
25
26   modify shoes;
27
28   ic create pkey = primary key (sequenceno);
NOTE: Integrity constraint pkey defined.
29   ic create regprodsub = distinct (region product subsidiary)
30              message = "Region, Product, Subsidiary combination must be unique";
NOTE: Integrity constraint regprodsub defined.
31   ic create storelimit = check(where=(stores < 50))
32              message = "Limit of 50 stores";
NOTE: Integrity constraint storelimit defined.
33   ic create returnsales = check(where=(returns+sales < inventory))
34              message = "Returns + Sales cannot exceed Inventory";
NOTE: Integrity constraint returnsales defined.
35   ic create fkey = foreign key (region) references sgflib.shoeregions
36            on update cascade on delete set null;
NOTE: Integrity constraint fkey defined.
37
38   run;
NOTE: MODIFY was successful for SGFLIB.SHOES.DATA.
39
40   quit;
```

The log shows that all integrity constraints were successfully created.

Removing Integrity Constraints

The IC DELETE statement is used to remove one or more integrity constraints. Here is how all of the integrity constraints from the previous example can be removed:

```
proc datasets library=sgflib nolist;

modify shoes;
ic delete _all_;
run;

modify shoeregions;
ic delete primkey;
run;

quit;
```

In this example, all of the integrity constraints in the SHOES SAS data set are deleted. Then, the primary key for the SHOEREGIONS data set is deleted. Since there are no indexes for either data set, and no additional integrity constraints for the SHOEREGIONS data set, the index files (SHOES.sas7bndx and SHOEREGIONS.sas7bndx) are both removed by SAS during the execution of the DATASETS procedure.

Reactivating Integrity Constraints

The IC REACTIVATE statement is used to reactivate an inactive *foreign key* integrity constraint. Foreign key integrity constraints can become inactive when SAS data sets are moved via the COPY, CPORT, CIMPORT, UPLOAD, or DOWNLOAD procedures.

Here is an example of reactivating the FKEY *foreign key* integrity constraint in the SHOES SAS data set:

```
proc datasets library=sgflib nolist;
   modify shoes;
      ic reactivate fkey references sgflib;
   run;
quit;
```

After this has executed, the link between the SHOES SAS data set and the SHOEREGION SAS data set via a *foreign key* built from the REGION variable (see previous examples) will be reestablished.

Managing Indexes with the Index Statements

The three INDEX statements—INDEX CREATE, INDEX DELETE, and INDEX CENTILES—are used to facilitate the use of *Indexes* on SAS data sets. An index is a tool that allows quick access to subsets of observations within a SAS data set. Programmers use indexes to retrieve subsets of SAS data sets without having to read the entire SAS data set, thereby saving time and computer resources. There are two types of indexes:

- **simple** – Created from a single variable, such as PatientID, SocSecNum, or PartNumber.
- **composite** – Created from two or more variables, such as Hospital and Patientid, or Region and State and Cityname. When you create a composite index, you give it a name, such as Hosp_PatID or Region_State_Cityname.

Both types of indexes can be created, deleted, and have their centiles updated by the DATASETS procedure.

When you generate the first index for a SAS data set, SAS automatically creates an index file to store the index values. The index file has the same name as the original SAS data set, but with the file extension of ***.sas7bndx***. For example, the index file for the SNACKS SAS data set would be named ***SNACKS.sas7bndx***. The index file is created in the same directory as the original SAS data set. All indexes for a SAS data set are stored together in the same index file. SAS automatically updates entries in the index file as observations are added, modified, or deleted. The index file is automatically deleted by SAS when all indexes (and integrity constraints) for a particular SAS data set have been removed.

The word "*centiles*" is short for "cumulative percentiles". The index header page holds 21 centiles that represent the lowest index value, the highest index value, the 5th percentile value, the 10th percentile value, etc. SAS uses centiles in its algorithm that determines whether or not to use a particular SAS index to retrieve data. SAS automatically refreshes the 21 centile values when 5 percent of the values of the indexed variable or variables have changed. You can use the DATASETS procedure to set and change the value at which centiles are updated, or to have SAS immediately refresh the centiles.

Creating Indexes

The INDEX CREATE statement creates either a simple or a composite index for a SAS data set. This is the format of the statement:

INDEX CREATE *index-specification*
</ <NOMISS> <UNIQUE> <UPDATECENTILES= ALWAYS | NEVER | *integer*>>;

Here are the details for the format of the INDEX CREATE statement:

- *index-specification1* – The index specification can be either for a simple or composite index:
  - *variable* – The name of the variable if this is a simple index
  - *indexname = (var-1 var-2*... var-n) – The name of the index followed by a list of variables within parenthesis for a composite index.
- NOMISS – Specifies that SAS is not to create index entries for observations with missing values in the index key variable(s)
- UNIQUE – States that the values in the index variable(s) are unique throughout the SAS data set
- UPDATECENTILES – Specifies when SAS can update centiles. Acceptable values are:
  - ALWAYS – Update centiles after any update of the SAS data set
  - NEVER – Never update centiles
  - *Integer* – Update centiles after *Integer* percent of the index values has been updated.

Here is an example of creating indexes for a SAS data set:

```
proc datasets library=sgflib nolist;

     modify shoes;

     index create sequenceno / unique updatecentiles=always;
     index create reg_sub_prod = (region subsidiary product) / nomiss;

run;
```

```
quit;
```

In the example, two indexes are being created: a simple index from SEQUENCENO and a composite index, named REG_SUB_PROD, that is created from the REGION, SUBSIDIARY, and PRODUCT variables. For the SEQUENCENO index, we specify that all values will be unique and that SAS is to update the centiles whenever an update is made to the data set. For the REG_SUB_PROD index, we specified that SAS is not to create index entries for observations where the values of REGION, SUBSIDIARY, and PRODUCT are all missing. The log for the execution of this proc looks like this:

```
236  proc datasets library=sgflib nolist;
237
238  modify shoes;
239
240  index create sequenceno / unique nomiss updatecentiles=always;
NOTE: Simple index SequenceNo has been defined.
241  index create reg_sub_prod = (region subsidiary product);
NOTE: Composite index reg_sub_prod has been defined.
242
243  run;
NOTE: MODIFY was successful for SGFLIB.SHOES.DATA.
244
245  quit;
```

As you can see both indexes were successfully created. Since these were the first indexes created for the SHOES data set, SAS created an index file named SHOES.sas7bndx in the same directory that SHOES resides in.

Deleting Indexes

The INDEX DELETE statement deletes one or more simple or composite indexes for a SAS data set. This is the format of the statement:

INDEX DELETE *index-1 index-n* | *_ALL_*;

You can list the indexes to be deleted in any order, or use the _ALL_ argument to delete all indexes for a particular SAS data set. When deleting composite indexes, you must specify the name that was supplied when the composite index was first created. Here is how we would delete the indexes created in the CREATE INDEX example:

```
proc datasets library=sgflib nolist;

modify shoes;
index delete sequenceno reg_sub_prod;
run;

quit;
```

The code, above, produces the following log:

```
246  proc datasets library=sgflib nolist;
247
248  modify shoes;
249  index delete sequenceno reg_sub_prod;
NOTE: All indexes defined on SGFLIB.SHOES.DATA have been deleted.
250  run;
NOTE: MODIFY was successful for SGFLIB.SHOES.DATA.
251
252  quit;
```

Note that we could have also have used “INDEX DELETE _ALL_;” since we were deleting all indexes from the SHOES data set. Or, we could have used two INDEX DELETE statements: one for SEQUENCENO and one for REG_SUB_PROD. There is not really an efficiency advantage to any particular method; all will work equally well.

Managing Centiles for Indexes

The INDEX CENTILES statement is used to either refresh (update) the centiles for an index, or to reset the value at which an index's centiles will be updated. This statement is only valid for existing SAS indexes. Here is the format of the statement:

INDEX CENTILES *index-1 <index-n>* **/** *<REFRESH> <UPDATECENTILES= ALWAYS | NEVER | integer>>*;

The REFRESH option causes SAS to immediately update the centiles for the specified index(es). The UPDATECENTILES option resets the percentage of index variable updates that must occur before SAS refreshes the centiles.

Here is an example:

```
proc datasets library=sgflib nolist;

modify shoes;
index centiles sequenceno / refresh;
index centiles reg_sub_prod / updatecentiles=20;
run;

quit;
```

In this example, the centiles for SEQUENCENO are refreshed immediately. Centiles for the REG_SUB_PROD composite index will be refreshed when 20% of the observations in the SHOES data set have been updated. The log looks like this:

```
271  proc datasets library=sgflib nolist;
272
273  modify shoes;
274  index centiles sequenceno / refresh;
NOTE: Index sequenceno centiles refreshed.
275  index centiles reg_sub_prod / updatecentiles=20;
NOTE: Index reg_sub_prod centiles update percent changed to 20.
276  run;
NOTE: MODIFY was successful for SGFLIB.SHOES.DATA.
277
278  quit;
```

You can see that both actions completed successfully.

For more information on SAS Indexes, refer to the book, **The Complete Guide on SAS Indexes**, cited in the References section of this paper.

Changing File Attributes with the MODIFY Statement

In a previous section, we saw how the MODIFY statement could be used to change the attributes of *variables* within a SAS data set. This section discusses how the MODIFY statement can be used to modify the attributes of SAS *files*. We can group these features of the MODIFY statement into three categories: changing basic data set attributes, modifying passwords, and changing generation groups. We will look at each of these in turn after taking a look at the format of the MODIFY statement.

MODIFY *SAS-file* <(*option-1 <...option-n>*)> </ <CORRECTENCODING=*encoding-value*>
<DTC=*SAS-date-time*> <GENNUM=*integer*> <MEMTYPE=*mtype*>>;

MEMTYPE is an overarching option of the MODIFY statement that restricts the specified operation to a specific SAS file type, such as DATA, CATALOG, etc. If it is specified, then only members of the SAS data library with the specified MEMTYPE are changed by the MODIFY command. If it is not specified, the default is that only SAS files with MEMTYPE=DATA (SAS data sets) are modified. This option can be useful when you have more than one file type with a given name. For example, if you had both a data set and a catalog named SHOES, then specifying MEMTYPE=CATALOG would inform SAS that you want the operation to work for the SHOES catalog.

Changing Data Set Attributes

The following options can be used to change attributes of SAS data sets:

- **CORRECTENCODING** – This option can be used to change the character encoding indicator to reflect the actual encoding of data within the file. For a list of acceptable values, refer to the CORRECTENCODING option in the MODIFY statement in the SAS National Language Support (NLS): Reference Guide.
- **DTC** – This option lets you modify the file's creation date-time stamp. Note that you can only reset the file's creating date-time to a value *earlier* than when the file was originally created.
- **LABEL** – This option allows you to specify a label for a SAS data set. It is useful for documenting SAS data sets that were not originally labeled by the creating SAS program.
- **SORTEDBY** – This option can be used to specify how the data are sorted. SAS simply records this information in the SAS data set header without checking its veracity. There are two sub-options:
  - **By-clause** – A BY statement followed by one or more variables in the SAS data set.
  - **_NULL_** – This sub-option removes the sort information from the SAS data set.
- **TYPE** – This option is used to assign a special type to a SAS data set. This is rarely used because most SAS files do not have a "type". PROC CORR can create a number of SAS data sets with different *types*. You can use the TYPE statement to change the type to one of them. See the documentation for PROC CORR for more details.

Here is an example of using all of these options, except TYPE, within the MODIFY statement:

```
proc datasets library=sgflib nolist;

modify shoes(label = "Shoe Sales for First Quarter of 2009" sortedby = region) /
                     correctencoding = wlatin1 dtc = "31MAR09:07:45:00"dt;
run;
quit;
```

In this example, we set SHOES encoding to WLATIN1 (the default in the US), set the creation date-time to March 31st 2009 at 7:45 am, set the label to "Shoe Sales for First Quarter of 2009", and specify that SHOES is sorted by the REGION variable. The SAS log looks like this:

```
8    proc datasets library=work nolist;
9
10   modify shoes(label = "Shoe Sales for First Quarter of 2009" sortedby = region) /
11               correctencoding = wlatin1 dtc = "31MAR09:07:45:00"dt;
12   run;

NOTE: MODIFY was successful for WORK.SHOES.DATA.
13   quit;
```

Modifying Passwords

The MODIFY statement can also be used to create, change and remove passwords. There are three *types* of passwords that may be specified for a SAS file:

- **ALTER** – This type restricts who may delete the file, update variable attributes, or create and delete indexes.
- **READ** – This type restricts who may read the SAS file.
- **WRITE** – This type restricts who may update the data in the file. For SAS data sets, it restricts who can add, modify, and delete observations.

These are the MODIFY statement options used to control passwords:

- **ALTER** – Is used to assign, change or remove an ALTER Password.
- **READ** – Is used to assign, change, or remove a READ password.
- **WRITE** – Is used to assign, change, or remove a WRITE password.
- **PW** – Is used to assign, change, or remove a single password that is used by all programs that need to ALTER, READ, or WRITE to a SAS file.

Here is an example of each of these options used:

```
proc datasets library=sgflib nolist;
```

```
/* Assign passwords */
modify shoes(alter=rock read=paper write=scissors);
modify snacks(pw=skynet);

/* Alter passwords */
modify shoes(alter=rock/hamlet read=paper/macbeth write=scissors/othello);
modify snacks(pw=skynet/cyberdyn);

/* Remove passwords */
modify shoes(alter=hamlet/ read=macbeth/ write=othello/);
modify snacks(pw=cyberdyn/);

run;

quit;
```

In the example above, individual ALTER, READ, and WRITE passwords are created for the SHOES data set, via the respective options. Then, a single password ALTER, READ, and WRITE password is created for the SNACKS data set with the PW option. Next, the three SHOES data set passwords are changed via the respective options, and the SNACKS data set password is changed via the PW option. Finally, the SHOES passwords are removed by specifying the individual ALTER, READ, and WRITE options, followed by the passwords and a slash ("/"), and the SNACKS passwords are deleted via the PW option, followed by the password and a slash.

The log for this program looks like this:

```
37   proc datasets library=sgflib nolist;
38
39   /* Assign passwords */
40   modify shoes(alter=XXXX read=XXXXX write=XXXXXXXX);
NOTE: MODIFY was successful for SGFLIB.SHOES.DATA.
41   modify snacks(pw=XXXXXX);
NOTE: MODIFY was successful for SGFLIB.SNACKS.DATA.
42
43   /* Alter passwords */
44   modify shoes(alter=XXXX/XXXXXX read=XXXXX/XXXXXXX write=XXXXXXXX/XXXXXXX);
NOTE: MODIFY was successful for SGFLIB.SHOES.DATA.
45   modify snacks(pw=XXXXXX/XXXXXXXX);
NOTE: MODIFY was successful for SGFLIB.SNACKS.DATA.
46
47   /* Remove passwords */
48   modify shoes(alter=XXXXXX/ read=XXXXXXX/ write=XXXXXXX/);
NOTE: MODIFY was successful for SGFLIB.SHOES.DATA.
49   modify snacks(pw=XXXXXXXX/);
51
52   run;
NOTE: MODIFY was successful for SGFLIB.SNACKS.DATA.
53  quit;
```

Note that SAS does not write the passwords to the SAS log. For more information on SAS passwords, refer to the *File Protection* chapter in the **SAS 9.2 Language Reference: Concepts**.

Modifying Generation Groups

You can have SAS automatically keep historic versions of a SAS data set by creating a *generation group* for it. You specify the number of versions of the data set that SAS is to save when you create the *generation group*. Once a *generation group* is created, the old copy is saved as a member of the *generation group* every time you replace the data set. Its name is affixed with a pound sign and a 3-digit number indicating which generation data set it is. For example, if you specified a *generation group* of four for the SHOES data set and updated it three times, the current data set would be named SHOES, the next most current would be SHOES#003, the next most current would be SHOES#002, and the oldest version would be SHOES#001. Generation data sets within a *generation group* are automatically stored in the same SAS directory. G*eneration group*s are a good way to keep track of past versions of important SAS data sets.

When you have a *generation group*, SAS processes the base file—the most recent file, without the generation numbers—by default. If you want one of the generations other than the base file to be processed, you must specify it using the GENNUM option. For instance:

```
proc print data=shoes(gennum=2);
run;
```

Also, SAS will not allow you to update historic data sets in a *generation group*.

Here is an example of changing the number of generation groups:

```
proc datasets library=sgflib nolist;

modify shoes(genmax=10);
modify snacks(genmax=0);

run;
quit;
```

In the example, the *generation group* for the SHOES data set is set to ten. If there was previously not a *generation group* for SHOES, SAS creates one. If there was, and the number was less than ten, SAS increases the number of generation data sets SHOES may have. If the number of generations was greater than ten and there were more than ten generation data sets, SAS deletes the older ones so that only ten remain.

Also in the example, the *generation group* for the SNACKS data set is removed by specifying it as zero. Once that executes, SAS deletes all SNACKS generation data sets except for the most recent. So, be careful that when you use this option you do not accidentally delete previous versions of the generations you wanted to keep.

You can get more information about SAS *generation group*s from the chapter **Understanding Generation Data Sets** in the **SAS 9.2 Language Reference: Concepts**, cited in the References section of this paper.

Recovering Indexes and Integrity Constraints with the REBUILD Statement

The REBUILD statement is a tool for rebuilding or deleting disabled indexes or integrity constraints. When SAS comes across a damaged data set containing indexes or integrity constraints and the DLDMGACTION data set or system option is set to *NOINDEX*, it deletes the index, repairs the file, changes the file so that it can only be opened in READ ONLY mode, and writes a warning in the SAS log that the REBUILD statement must be run. Running the REBUILD statement will rebuild the data set's indexes or integrity constraints and finish the work of recovering the damaged SAS data set. Thereafter, the data set can be opened in any mode.

Here is the format of the REBUILD statement:

REBUILD *SAS-file* </ ALTER=*password* GENNUM=*n* MEMTYPE=*mtype* NOINDEX>;

NOINDEX is the most noteworthy option. Normally, you would choose to rebuild the data sets indexes or integrity constraints. However, NOINDEX specifies that SAS is to drop—not rebuild—all indexes or integrity constraints. So, if you have a reason for the indexes to *not* be rebuilt, you would use the NOINDEX option to have them permanently deleted and the data set freed-up to be opened in any mode.

Here is an example of the REBUILD statement:

```
proc datasets library=scratlib nolist;

rebuild shoes;

run;
quit;
```

In this example, we are rebuilding the two indexes for the SHOES data set. The log looks like this:

```
NOTE: Rebuilding SCRATLIB.SHOES (memtype=DATA).
NOTE: File SCRATLIB.SHOES.INDEX does not exist.
NOTE: Indexes recreated:
```

```
 1 Simple indexes
 1 Composite indexes
318  quit;
```

Consequently, the simple and composite indexes for SHOES were rebuilt, and the data set is now back to normal.

MANAGING FILES IN SAS LIBRARIES

The DATASETS procedure has about a dozen statements that you can use to manage the files within a SAS library. Although the COPY statement mirrors another SAS procedure, the functionality of the rest of the statements is unique to PROC DATASETS. The following sections describe each of the PROC DATASETS statements that you can use to manage a SAS library.

Cascading File Renames with the AGE Statement

The AGE statement is used to rename a group of SAS data sets—one to another's name—and delete the last SAS data set named in the statement. This statement is handy for cases where you want to maintain your own historical SAS data sets without using *generation group*s. The format of the statement is:

AGE current-file-name related-SAS-file-1 <…current-file-name related-SAS-file-n>
</ <ALTER=alter-password> <MEMTYPE=memtype>>;

If the SAS file is protected by an ALTER password, then the ALTER option must be used. If the AGE statement specifies a SAS file whose name has more than one member type, the MEMTYPE defaults to "DATA". If you are aging some other file type, then you must specify it in the MEMTYPE statement.

There are several caveats concerning the files listed in the AGE statement that you should know. The first file named in the AGE statement ***must*** exist, but the subsequent files do not have to exist. SAS will simply age the files as best it can and write warning messages to the log. Also, you can only age entire *generation group*s from one name to another. That is; you cannot age the individual generation data Sets within a *generation group* using the AGE statement.

Here is an example of the AGE statement:

```
proc datasets library=work nolist;

age newshoes shoes oldshoes oldestshoes;

run;
quit;
```

This log shows what happens when this example is run:

```
266  proc datasets library=work nolist;
267    age newshoes shoes oldshoes oldestshoes;
268  run;
NOTE: Deleting WORK.OLDESTSHOES (memtype=DATA).
NOTE: Aging the name WORK.OLDSHOES to WORK.OLDESTSHOES (memtype=DATA).
NOTE: Aging the name WORK.SHOES to WORK.OLDSHOES (memtype=DATA).
NOTE: Aging the name WORK.NEWSHOES to WORK.SHOES (memtype=DATA).
269   quit;
```

Since the last data set named in the AGE statement is deleted, you need to be careful in using this statement.

Renaming SAS Files with the CHANGE Statement

You can use the CHANGE statement to rename one or more files in a SAS data library. The CHANGE statement is handy in cases where the initial name of a file does not adhere to your organization's naming standards, or a file does not have a meaningful name. The format of the CHANGE statement is:

CHANGE *old-name-1 = new-name-1 < old-name-n = new-name-n*> < /
<ALTER=*alter-password*> <GENNUM=ALL | *integer*> <MEMTYPE=*mtype*>>

You are familiar with the ALTER and MEMTYPE options from previous statements. The GENNUM option allows you to change the name of an entire *generation group* or to change the name of individual members, based on the generation number.

Here is an example of the CHANGE statement:

```
proc datasets library=work nolist;

        change bweight = BodyWeight shoes = ShoeSales;

run;
quit;
```

In this example, we changed the name of the BWEIGHT data set to BODYWEIGHT and the name of all of the files in the SHOES *generation group* to SHOESALES. Here is what the SAS log looks like:

```
43   proc datasets library=work nolist;
44     change bweight = BodyWeight shoes = ShoeSales;
45   run;
NOTE: Changing the name WORK.BWEIGHT to WORK.BODYWEIGHT (memtype=DATA).
NOTE: Changing the name WORK.SHOES to WORK.SHOESALES (memtype=DATA gennum=ALL).
46   quit;
```

The second note in the log states that all of the members of the SHOES *generation group* were renamed to SHOESALES.

Copying Files Using the COPY, SELECT, and EXCLUDE Statements

The COPY statement copies one or more SAS files from one SAS library to another. It is analogous to the COPY procedure. The default behavior is for *all* SAS files in one library to be copied to the other. You can specify individual files to be copied via the SELECT statement, or you can use the EXCLUDE statement to have all files, *except* the ones named in the EXCLUDE statement, copied between libraries. The SELECT and EXCLUDE statements are described in detail later in this section of the paper.

Here is the basic format of the COPY statement:

COPY OUT=*libref-1* <ALTER=*alter-password*> <CLONE|NOCLONE> <CONSTRAINT=YES|NO> <DATECOPY> <FORCE> <IN=*libref-2*> <INDEX=YES|NO> <MEMTYPE=(*mtype-1 <...mtype-n>*)> <MOVE>> ;

These are the options for the COPY statement:

- **OUT** – Specifies the libref of the SAS library the files are to be copied to.
- **IN** – Names the libref of the SAS library of the files that are to be copied. The default is the libref specified by the LIBRARY option on the PROC DATASETS statement.
- **ALTER** – Is used when you are using the MOVE option to move files that are have an ALTER password.
- **CLONE | NOCLONE** – Specifies whether to copy the following file characteristics to the file that is copied or moved:
  - Input/output buffer size
  - Data set compression
  - Whether free space is reused in compressed data sets
  - Whether compressed data sets can be accessed by observation number
  - Data representation
  - Encoding value
- **CONSTRAINT = YES | NO** – Copy existing integrity constraints when copying a data set.
- **DATECOPY** – Copy the original file's creation and last updated date/times to the new copy of the file.
- **FORCE** – Must be used to MOVE a SAS data set that has an audit trail file.
- **INDEX = YES | NO** – Copy indexes for indexed data sets that are copied or moved.
- **MEMTYPE = (*mtype-1 <...mtype-n>*)** – Specifies the type(s) of file that should be copied or moved.
- **MOVE** – States that the file or files should be physically moved from one SAS library to the other.

Here is an example of the COPY statement:

```
proc datasets library=work nolist;

copy out=bkuplib clone datecopy;
       select shoesales;
run;

copy out=bkuplib clone datecopy force constraint=yes index=yes move;
       Select bodyweight;
run;
quit;
```

In this example, we are *copying* the SHOESALES data set and *moving* the BODYWEIGHT data set to another library. We used the CLONE and DATECOPY options on both copies so that the original data sets' page size and create/updated dates will be copied along with them to the new SAS library. Since the BODYWEIGHT data set has an integrity constraint, an audit file, and an index, the CONSTRAINT, FORCE, and INDEX options were used, respectively, to have those copied to the new SAS data library. Note that the MOVE statement tells SAS to *move*—not *copy*—the BODYWEIGHT data set and its attendant files to the other library.

Here is the log from this program:

```
274  proc datasets library=work nolist;
275  copy out=bkuplib clone datecopy;
276      select shoesales;
277  run;
NOTE: Copying WORK.SHOESALES to BKUPLIB.SHOESALES (memtype=DATA).
INFO: Engine's block-read method is in use.
INFO: Engine's block-write  method is in use.
NOTE: There were 33 observations read from the data set WORK.SHOESALES.
NOTE: The data set BKUPLIB.SHOESALES has 33 observations and 9 variables.
NOTE: Copying WORK.SHOESALES (gennum=3) to BKUPLIB.SHOESALES (memtype=DATA gennum=3).
INFO: Engine's block-read method is in use.
INFO: Engine's block-write  method is in use.
NOTE: There were 33 observations read from the data set WORK.SHOESALES (gennum=3).
NOTE: The data set BKUPLIB.SHOESALES (gennum=3) has 33 observations and 9 variables.
NOTE: Copying WORK.SHOESALES (gennum=2) to BKUPLIB.SHOESALES (memtype=DATA gennum=2).
INFO: Engine's block-read method is in use.
INFO: Engine's block-write  method is in use.
NOTE: There were 33 observations read from the data set WORK.SHOESALES (gennum=2).
NOTE: The data set BKUPLIB.SHOESALES (gennum=2) has 33 observations and 9 variables.
NOTE: Copying WORK.SHOESALES (gennum=1) to BKUPLIB.SHOESALES (memtype=DATA gennum=1).
INFO: Engine's block-read method is in use.
INFO: Engine's block-write  method is in use.
NOTE: There were 33 observations read from the data set WORK.SHOESALES (gennum=1).
NOTE: The data set BKUPLIB.SHOESALES (gennum=1) has 33 observations and 9 variables.
278  copy out=bkuplib clone datecopy constraint=yes force index=yes move;
279      Select bodyweight;
280  run;
NOTE: Moving WORK.BODYWEIGHT to BKUPLIB.BODYWEIGHT (memtype=DATA).
WARNING: WORK.BODYWEIGHT.DATA has an AUDIT file associated with it. AUDIT files will not
be copied.
INFO: Engine's block-read method is in use.
INFO: Engine's block-write  method is in use.
NOTE: Simple index married has been defined.
NOTE: Integrity constraint marriedic defined.
NOTE: There were 50000 observations read from the data set WORK.BODYWEIGHT.
NOTE: The data set BKUPLIB.BODYWEIGHT has 50000 observations and 10 variables.
81  quit;
```

In the first part of the log, you can see that SAS copies each of the generation data sets in the SHOESALES *generation group*. The second part of the log shows that the audit file for BODYWEIGHT was not copied—even though the FORCE option was used. Note that the FORCE option simply allows SAS data sets with audit files to be copied or moved; the audit files themselves are *never* copied or moved. The log also shows that both the index and the integrity constraint have been defined for the moved BODYWEIGHT data set. SAS does not actually copy/move indexes or integrity constraints between libraries. It simply rebuilds them in the new library.

The SELECT statement is only used as a modifier to a COPY statement. The format of the SELECT statement is:

SELECT *SAS-file-1* <*...SAS-file-n*> </ <ALTER=*alter-password*> <MEMTYPE= *mtype*>>;

You may select one or more files for SAS to copy via the SELECT statement. A SELECT statement and an EXCLUDE statement cannot be used for the same COPY. Refer to the example above for an instance of the SELECT statement being used.

The EXCLUDE statement is the other modifier for the COPY statement. The format of the EXCLUDE statement is:

EXCLUDE *SAS-file-1* <*...SAS-file-n*> </ <MEMTYPE= *mtype*>>;

Files listed in the EXCLUDE statement will not be copied between SAS libraries. When you have a lot of files to copy, and only want to exclude a few, it is easier to list the ones to be excluded that it is to list all of the files that should be copied. As mentioned earlier, an EXCLUDE statement and a SELECT statement cannot be used for the same COPY.

Permanently Removing Files with the DELETE Statement

The DELETE statement is used to do exactly what you would imagine: delete SAS files. It is a great tool to use to clean up SAS data libraries. The format of the DELETE statement is:

DELETE *SAS-file-1* <*...SAS-file-n*> </ <ALTER=*alter-password*> <GENNUM=ALL|HIST|REVERT|*integer*> <MEMTYPE=*mtype*>>;

The ALTER and MEMTYPE options have the same function as in earlier examples. GENNUM has three possible values:

- **ALL** – Delete base version and all historical versions of the *generation group*
- **HIST** – Delete *only* the historical versions of the *generation group*
- **REVERT** – Delete the base version and rename the most recent historical version to the base version
- **Integer** – The specific version of the *generation group* to delete. A positive integer refers to the generation number concatenated after the data set's name, e.g. 3 refers to SHOES#003. A negative number refers to the relative number, going from most recent to oldest; e.g. for GENMAX=4, -3 refers to the oldest generation data set.

Here is an example of deleting SAS files:

```
proc datasets library=work nolist;

delete shoes / gennum=hist;
delete bodyweight / memtype=data;

run;
quit;
```

In the example above, all historical versions of the SHOES data set (SHOES#002, SHOES#003, and SHOES#004) are deleted, but the base data set, SHOES, is kept. The BODYWEIGHT SAS data set is deleted. Note that the MEMTYPE option was used because there is also a BODYWEIGHT catalog in the WORK data library.

When the DELETE statement is executed, SAS deletes the specified files immediately without prompting you for your confirmation. Any associated index files are deleted too, unless they contain *foreign key* integrity constraints or a *primary key* with *foreign key* references. So, be sure that you really want to delete the files that you specify, and check the log to be sure that they were deleted.

Swapping File Names with the EXCHANGE Statement

The EXCHANGE statement is one of the most unique statements in the SAS language. It swaps the names of two SAS data sets. So, if it were executed against NEWFILE and OLDFILE, the name of NEWFILE would be changed to "OLDFILE" and the name of OLDFILE would be changed to "NEWFILE". Consequently, the names of the two files would be *exchanged*. Index files and *generation group*s are also renamed if they exist.

Here is the form of the EXCHANGE statement:

> **EXCHANGE** *name-1=other-name-1* <*...name-n=other-name-n*> </ <ALTER=*alter-password*> <MEMTYPE=*mtype*>>;

The ALTER, GENNUM and MEMTYPE options have the same function as in earlier examples. Note that you can exchange the names of multiple SAS files within the same execution of the EXCHANGE statement. Be careful when exchanging multiple names, as SAS will exchange names in the order that they appear in the directory, not the order that the exchanges appear in the EXCHANGE statement.

Here is an example:

```
proc datasets library=work nolist;

exchange shoes  = shoes_backup
         snacks = snacks_current;

run;
quit;
```

In this example, we exchange the names of the SHOES and SHOES_BACKUP data sets, and the names of the SNACKS and the SNACKS_CURRENT data sets. Since the original SHOES SAS data set had an index, after the exchange, we find that there is a SHOES_BKUP.sas7bdat and a SHOES_BKUP.sas7bndx (index) file, while there is simply a SHOES.sas7bdat file.

Fixing Damaged Files with the REPAIR Statement

It is usually bad news when you have to execute the REPAIR statement, because it means that you are attempting to restore a damaged SAS data set or catalog. Data sets and catalogs get damaged for a variety of reasons, the most common of which are I/O errors while data are being written to the file, a system failure, or running out of space on the media the file is being written to. When such an error occurs and SAS attempts to process a damaged file, it writes error messages to the SAS log and processing of the DATA step or procedure ends. When you use the REPAIR statement to repair a:

- **SAS data set** – SAS tries to restore the data set to a usable condition, and rebuilds any indexes and integrity constraints (unless the DLDMGACTION option has been set to NOINDEX).
- **SAS catalog** – SAS tries to restore the damaged catalog entry. If the entire catalog is damaged, SAS tries to restore all entries in the catalog.

In either of these cases, SAS provides information in the SAS log regarding the success or failure of the REPAIR.

This is the format of the REPAIR statement:

> **REPAIR** *SAS-file-1* <*...SAS-file-n*> </ <ALTER=*alter-password*> <GENNUM=*integer*>
> <MEMTYPE=*mtype*>>;

Here is an example of a REPAIR statement:

```
proc datasets library=scratlib nolist;

repair shoes;

run;
quit;
```

In this example, the damaged SHOES data set is being repaired. The log looks like this:

```
NOTE: Repairing SCRATLIB.SHOES (memtype=DATA).
NOTE: File SCRATLIB.SHOES.INDEX does not exist.
NOTE: Indexes recreated:
 1 Simple indexes
 1 Composite indexes
330  quit;
```

Note that this log looks similar to the one from the REBUILD statement. The difference is that the REPAIR statement repairs problems with the SAS data set or catalog, and rebuilds indexes and integrity constraints. The REBUILD option simply rebuilds disabled indexes and integrity constraints for SAS data sets.

Keeping Files During a Delete Operation using the SAVE Statement

The SAVE statement should be used with extreme caution, because it causes every other file in the SAS data library, *except* those listed in the SAVE statement, to be deleted. Consequently, it is useful when you have to clean up a SAS library with many files in it and only want to keep (*save*) a handful of files.

The format of the SAVE statement is:

SAVE *SAS-file-1* <*...SAS-file-n*> </ MEMTYPE=*mtype*>;

Here is an example of the SAVE statement:

```
proc datasets library=work nolist;

save shoes_backup;

run;
quit;
```

The example cleans the WORK library of all SAS files except the SHOES_BACKUP data set. The NOTES in the log look like this:

```
NOTE: Saving WORK.SHOES_BACKUP (memtype=DATA).
NOTE: Deleting WORK.SHOES (memtype=DATA).
NOTE: Deleting WORK.SNACKS (memtype=DATA).
NOTE: Deleting WORK.SNACKS_CURRENT (memtype=DATA).
```

You can see that SHOES_BACKUP was saved, but that three other files were deleted.

CONCLUSION

Like a Swiss Army knife, the DATASETS procedure is a multi-faceted tool. It has a plethora of facilities for modifying the attributes of SAS variables, SAS data set files, and other SAS files. Some of the facilities mirror the functions of other SAS procedures such as PROC APPEND and PROC COPY. Others, such as INDEX CREATE and IC CREATE, can be accomplished using other SAS tools such as the DATA step and PROC SQL, respectively. Still other capabilities—including the CHANGE, REBUILD, and SAVE statements—are unique only to the DATASETS procedure. Whatever the case, PROC DATASETS is the most versatile and function-rich procedure available in Base SAS. So, you should become well-acquainted with this *Swiss Army knife* of SAS procedures and make it an integral part of your SAS programming repertoire.

DISCLAIMER

The contents of this paper are the work of the author and do not necessarily represent the opinions, recommendations, or practices of Westat.

REFERENCES

SAS Institute Inc. 2009. Base SAS® 9.2 Procedures Guide. Cary, NC: SAS Institute Inc. Available: http://support.sas.com/documentation/cdl/en/proc/61895/PDF/default/proc.pdf

SAS Institute Inc. 2009. SAS® 9.2 Language Reference: Concepts. Cary, NC: SAS Institute Inc.
Available: http://support.sas.com/documentation/cdl/en/lrcon/61722/PDF/default/lrcon.pdf

SAS Institute Inc. 2009.SAS® 9.2 National Language Support (NLS): Reference Guide. Cary, NC: SAS Institute
Available: http://support.sas.com/documentation/cdl/en/nlsref/61893/PDF/default/nlsref.pdf

Raithel, Michael, and Rhoads, Mike. 2009. "You May Be A SAS Energy Hog If…" SAS Institute Inc. 2009. Proceedings of the SAS® Global Forum 2009 Conference. Cary, NC: SAS Institute Inc.
Available: http://support.sas.com/resources/papers/proceedings09/041-2009.pdf

Raithel, Michael A. 2006. *The Complete Guide to SAS Indexes*: Cary, NC: SAS Institute, Inc.
Available*:* http://www.sas.com/apps/pubscat/bookdetails.jsp?catid=1&pc=60409

ACKNOWLEDGMENTS

The author would like to thank Westat management for supporting his participation in SAS Global Forum 2010. He would also like to thank Westat Vice President Mike Rhoads for the title of his paper. During the presentation of their joint SAS Global Forum 2009 paper, *You May Be A SAS Energy Hog If…* Mike referred to PROC DATASETS as "the Swiss Army knife of SAS procedures", which seemed like a *perfect* title for this paper.

CONTACT INFORMATION

I would love to get your feedback on this paper; especially if you found it helpful. I would also like to know about your own unique uses for PROC DATASETS. You can contact me at the following email address:

michaelraithel@westat.com