

## SAS Advanced Programming Techniques

1. [Creating Samples and Indexes](#)
  2. [Combining Data Vertically](#)
  3. [Combining Data Horizontally](#)
  4. [Using Lookup Tables to Match Data](#)
  5. [Formatting Data](#)
  6. [Modifying SAS Data Sets and Tracking Changes](#)
  7. [Introduction to Efficient SAS Programming](#)
  8. [Controlling Memory Usage](#)
  9. [Controlling Data Storage Space](#)
  10. [Utilizing Best Practices](#)
  11. [Selecting Efficient Sorting Strategies](#)
  12. [Querying Data Efficiently](#)
- 
-

# Chapter 1

## Creating Samples and Indexes

### Introduction

Some of the SAS data sets that you work with might be quite large. Large data sets can take a relatively long time to process because, by default, SAS reads observations in a data set sequentially. For example, assume that your data set has five hundred observations. In order to read the five-hundredths observation, SAS first reads the observations numbered 1 through 499, and then reads observation number 500. Sometimes, you might want to make SAS access specific observations directly for greater speed and efficiency.

You will need to access specific observations directly when you want to create a **representative sample** of a large data set, which can be much easier to work with than the full data set. For example, if you are concerned about the accuracy of the data in a large data set, you could audit a small sample of the data in order to determine if a full audit is necessary. A representative sample is a **subset** of the full data set. The subset should contain observations that are taken from throughout the original data set so that the subset gives an accurate representation of the full data set. This lesson discusses two types of representative samples:

- **systematic samples**
- **random samples.**

**Indexes** can also make working with very large data sets easier. An index is a separate data structure that is associated with a data set, and that contains information about the specific location of observations in the data set according to the value of **key variables**. An index enables you to access a particular observation directly, without needing to read all of the observations that precede it in the data set. Indexes are useful in many instances, including WHERE and BY processing. This lesson discusses how to create and maintain both **simple and composite indexes**.



### Setting Up Filerefs for Practices in This Lesson:

Submit this SAS program to set up filerefs for practices. If you have already done this in the current SAS session, you can skip this task. (This program is the same for all lessons.)

### Objectives

In this lesson, you learn to

1. create a systematic sample from a known number of observations
2. create a systematic sample from an unknown number of observations
3. create a random sample with replacement
4. create a random sample without replacement
5. use indexes
6. create indexes in the DATA step
7. manage indexes with PROC DATASETS
8. manage indexes with PROC SQL
9. document and maintain indexes.

## Creating a Systematic Sample from a Known Number of Observations

One type of representative sample that you might want to create is a **systematic sample**. A systematic sample contains observations that are chosen from the original data set at **regular intervals**. For example, a systematic sample could contain every hundredth observation of a very large data set.

To create a systematic sample from a data set that has a known number of observations, you use the POINT= option in the SET statement.

### General form, SET statement with POINT= option:

**SET** *data-set-name* **POINT=** *point-variable*;

where *point-variable*

- names a temporary numeric variable whose value is the observation number of the observation to be read
- must be given a value before the execution of the SET statement
- must be a variable and not a constant value.

The value of the variable that is named by the POINT= option should be an integer that is greater than zero and less than or equal to the number of observations in the SAS data set. SAS uses the value to point to a specific observation in the SET statement. You must assign this value within the program so that the POINT= variable has a value when the SET statement begins execution. Also, in order for SAS to read different observations into the sample, the value of the POINT= variable must change during execution of the DATA step.

### Example

You can place the SET statement with the POINT= option inside a DO loop that will assign a different value to the POINT= variable on each iteration. In the following code sample, the DO loop assigns a value to the variable pickit, which is used by the POINT= option to select every tenth observation from **Sasuser.Sale2000**. Notice that the following example is not a complete step.

```
do pickit=1 to 142 by 10;
  set sasuser.sale2000 point=pickit;
  output;
end;
```



In general, samples are most useful when you are working with very large data sets. However, the sample data sets that are included in this lesson are relatively small. The basics of creating samples and indexes are the same no matter the size of the data set, and you can apply the techniques in this lesson to larger data sets.

In the following SET statement, what does this one name?

set original point=thisone;

**Q1.**

- A a data set
- B a variable
- C a value
- D an output location

By default, SAS reads a data set sequentially, beginning with the first observation. A DATA step stops processing when SAS reaches the **end-of-file marker** after reading the last observation in the data set.

The POINT= option uses **direct-access read mode**, which means that SAS only reads those observations that you direct it to read. In direct-access read mode, SAS does not detect the end-of-file marker. Therefore, when you use the POINT= option in a SET statement, you must use a **STOP statement** to prevent the DATA step from looping continuously.

The STOP statement causes SAS to stop processing the current DATA step immediately and resume processing statements after the end of the current DATA step.

#### General form, STOP statement:

```
STOP;
```

#### Example

The **Sasuser.Revenue** data set contains 142 observations. Suppose you want to select a ten-observation subset of the data set **Sasuser.Revenue** by reading every fifteenth observation. You can use the POINT= option in a SET statement inside a DO loop to create this sample.

In the following example, the DO loop assigns a value to the variable *pickit*, which is used by the POINT= option to select an observation from **Sasuser.Revenue**. The OUTPUT statement writes the program data vector values to the SAS data set, and the STOP statement stops the DATA step from continuing to execute after the ten observations have been selected.

```
data sasuser.subset;
  do pickit=1 to 142 by 15;
    set sasuser.revenue point=pickit;
    output;
  end;
  stop;
run;

proc print data=sasuser.subset;
run;
```

The program above creates the **Sasuser.Subset** data set, shown below.

Obs	Origin	Dest	FlightID	Date	Rev1st	RevBusiness	RevEcon
1	ANC	RDU	IA03400	02DEC1999	15829	28420	68688
2	CCU	PEK	IA09700	08DEC1999	19992	27832	36010
3	DEL	JRS	IA09001	27DEC1999	14434	16169	42066
4	FRA	RDU	IA00400	18DEC1999	22893	28824	87750
5	HKG	SYD	IA10100	24DEC1999	34074	39990	101898
6	HNL	SFO	IA03000	30DEC1999	10868	16825	46248
7	LHR	JED	IA08100	21DEC1999	21942	42330	63990
8	PEK	CCU	IA09801	11DEC1999	23324	27264	37949
9	RDU	LHR	IA00101	17DEC1999	17600	30520	79119
10	SIN	CCU	IA09401	23DEC1999	20061	24843	36556

## Creating a Systematic Sample from an Unknown Number of Observations

Sometimes you might not know how many observations are in the original data set from which you want to create a systematic sample. In order to make a systematic sample, you need to know the total number of observations in the original data set so that you can choose observations that are evenly distributed from it.

You can use the NOBS= option in the SET statement to determine how many observations there are in a SAS data set.

### General form, SET statement with NOBS= option:

`SET SAS-data-set NOBS=variable;`

where *variable* names a temporary numeric variable whose value is the number of observations in the input data set.

 If **multiple data sets** are listed in the SET statement, the value of the NOBS= variable is the **total number** of observations in all of the data sets that are listed.

The value of the NOBS= variable is assigned automatically during compilation when SAS reads the descriptor portion of the data file. Therefore, this value is available at any time during execution of the DATA step.

 The total that is used as a value for the NOBS= variable includes observations that have been marked for deletion but have not been physically removed from the data set.

You can use the NOBS= option in conjunction with the POINT= option to create a systematic sample of a data set if you do not know how many observations are in the data set.

### Example

Suppose you want to create a systematic sample of the **Sasuser.Revenue** data set, and you do not know how many observations are in it. In the following example, *totobs* is assigned the value of the total number of observations in the data set **Sasuser.Revenue** during compilation. Then, *totobs* is used as the upper limit for the DO loop that controls how many observations are chosen for the systematic sample.

```
data sasuser.subset;
do pickit=1 to totobs by 10;
  set sasuser.revenue point=pickit nobs=totobs;
  output;
end;
stop;
run;
```

The resulting **Sasuser.Subset** data set contains every tenth observation from the **Sasuser.Revenue** data set. Notice that when the program above is submitted, the DATA step iterates only once, and the DO loop iterates multiple times within the DATA step.

The variable that is created by the NOBS= option is assigned a value

**Q.2.**

- A automatically during compilation of the SET statement
- B automatically during execution of the SET statement
- C during compilation of the SET statement, by program statements
- D during execution of the SET statement, by program statements.

### Example

In the last example, you saw a DATA step that selects a systematic sample of the **Sasuser.Revenue** data set by reading every tenth observation beginning with observation number 1, without prior knowledge of the total number of observations in **Sasuser.Revenue**. Remember that *totobs* is assigned a value that is the total number of observations in **Sasuser.Revenue**. The variable *pickit* has an initial value of 1, and this value increases by 10 with each iteration of the DO loop until it reaches the value of *totobs*.

The following steps show the execution of the DATA step below.

1. This DATA step creates a systematic sample of the input data set **Sasuser.Revenue**. The output data set **Sasuser.Subset** will contain every tenth observation from the input data set.

```
data sasuser.subset;
do pickit=1 to totobs by 10;
  set sasuser.revenue point=pickit nobs=totobs;
  output;
end;
stop;
run;
```

2. During the compilation phase, SAS reads the descriptor portion of the input data set and creates the PDV. The variables *totobs* and *pickit* are also created in the PDV. SAS obtains the total number of observations from the descriptor portion of the input data set and assigns this value to the variable *totobs*.
3. Execution begins. All of the remaining variables are initialized. The DO loop begins and *pickit* is assigned an initial value of 1.
4. SAS reads the first observation from **Sasuser.Revenue** in direct-access mode and records the values in the PDV.
5. SAS writes the values from the PDV to the sample as the first observation in the **Sasuser.Subset** data set, except for the temporary variables *pickit* and *totobs*.
6. The value of *pickit* is increased by 10 and the DO loop iterates. The values for all other variables are retained in the PDV.
7. The DO loop continues to execute. SAS reads the eleventh observation from **Sasuser.Revenue** and records the values in the PDV, overwriting the retained values. Then, these values are written to the **Sasuser.Subset** data set.
8. The DO loop continues to iterate until *pickit* has a value of 151 (which is greater than the value of *totobs*, 142). At that point, the DO loops ends and the STOP statement executes, causing the DATA step to stop executing.

- When the DATA step finishes executing, the **Sasuser.Subset** data set contains a systematic sample of every tenth observation from **Sasuser.Revenue**.

## Creating a Random Sample with Replacement

Another type of representative sample that you might want to create is a **random sample**. A random sample contains observations that are chosen from the original data set on a random basis.

When you create a random sample **with replacement**, it is possible for one observation to appear in the sample multiple times. You can think of the original data set as a pool of possible observations that may be chosen for inclusion in the sample. For each observation in the sample data set, SAS chooses an observation randomly from the original pool, copies it to the sample data set, then replaces it in the pool.

## Using the RANUNI Function

In order to create a random sample, you need to generate a random number. SAS provides several random number functions to generate random numbers from various distributions. One example of a random number function is the RANUNI function.

### General form, RANUNI function:

**RANUNI (seed)**

where *seed* is a nonnegative integer with a value less than  $2^{31}-1$  (2,147,483,647).

The RANUNI function generates streams of random numbers from an initial starting point, called the **seed**. If you use a positive seed, you can always replicate the stream of random numbers by using the same DATA step. If you use zero as the seed, the computer clock initializes the stream, and the stream of random numbers is not replicable.

The numbers that the RANUNI function returns are all between 0 and 1 (noninclusive). Examples of the type of number RANUNI returns include .01253689 and .95196500. If you use RANUNI in a DATA step that generates only one random number, RANUNI returns only the first number from the stream. If you use RANUNI in a DATA step that generates multiple numbers, such as in a DO loop, RANUNI will return a different random number each time the loop iterates.

Let's take a look at some examples.

### Example

The following DATA step creates one observation with one variable named varone and assigns a random number to it. You can submit this DATA step multiple times, or in multiple SAS sessions, and varone will have the same value.

```
data random1;
  varone=ranuni(10);
run;
```

The following DATA step creates ten observations with one variable named varone and assigns a random number as a value for varone. You can submit this DATA step multiple times, or in multiple SAS sessions, and varone will have the same ten values.

```
data random2;
do i=1 to 10 by 1;
  varone=ranuni(10);
  output;
end;
run;
```

If you changed the seed value from *10* to a different value in either of the two DATA steps above, the values for varone would be different when you submitted the DATA step than it was when the seed value was *10*. However, varone will have the same ten values each time you submit the DATA step with a constant seed value.



For clarity and consistency in the sample programs, all remaining examples of the RANUNI function in this lesson will use a seed of *0*.

### Using a Multiplier with the RANUNI Function

By default, RANUNI generates numbers that are between 0 and 1. To increase the interval from which the random number is chosen, you use a **multiplier** on the RANUNI function. For example, if you want to generate a random number between 0 and 50, you use the following code:

```
ranuni(0)*50
```

**Q.3.**

Write an expression that will generate a **random number** between 0 and 10. Use 0 as the seed to generate your number.

---

You have seen that the RANUNI function generates a random number between 0 and 1. However, in order to create a random sample, you need to generate a random integer that will match one of the observation numbers in the original data set.

### Using the CEIL Function

You can use the CEIL function in conjunction with the RANUNI function to generate a random integer.

#### General form, CEIL function:

**CEIL (argument)**

where *argument* is numeric.

The CEIL function returns the smallest integer that is greater than or equal to the argument. Therefore, if you apply the CEIL function to the result of the RANUNI function, you can generate a random integer.

#### Example

The following example creates a random integer between 1 and 50:

```
ceil(ranuni(0)*50)
```

**Q.4.**

Write an expression that will generate a **random integer** between 1 and 10. Use 0 as the seed to generate your integer.

---

Now that you have seen how to use the CEIL function in conjunction with the RANUNI function, let's take a look at how to use these functions in a DATA step to create a random sample. You use the CEIL and RANUNI functions together to generate a random integer that is assigned as a value for the variable to which the POINT= option points.

### Example

In the following example, the CEIL and RANUNI functions are used together in the assignment statement for pickit, which is the variable that is pointed to by the POINT= option. The NOBS= option assigns the total number of observations in the **Sasuser.Revenue** data set as a value for the totobs variable. The variable totobs is then used as the multiplier in the CEIL function, so that for each iteration of the DO loop, every observation in **Sasuser.Revenue** has an equal chance of being picked for inclusion in the sample.

```
data work.rsubset (drop=i sampsize);
  sampsize=10;
  do i=1 to sampsize;
    pickit=ceil(ranuni(0)*totobs);
    set sasuser.revenue point=pickit nobs=totobs;
    output;
  end;
  stop;
run;

proc print data=work.rsubset label;
  title 'A Random Sample with Replacement';
run;
```

Since the program uses a seed of 0 for the RANUNI function, the **Work.Rsubset** data set will be different each time you submit this code. Here is an example of the possible output.

A Random Sample with Replacement

Obs	Origin	Dest	FlightID	Date	Rev1st	RevBusiness	RevEcon
1	DXB	FRA	IA07800	22DEC1999	18630	40608	53148
2	HND	HKG	IA11001	21DEC1999	19980	20160	34440
3	HNL	SFO	IA03001	13DEC1999	10868	18171	48544
4	WLG	CBR	IA10600	08DEC1999	15496	17908	29106
5	FRA	DXB	IA07700	09DEC1999	19872	47376	58092
6	SYD	HKG	IA10200	30DEC1999	35967	34830	123284
7	HNL	SFO	IA03000	06DEC1999	11856	16152	47888
8	SFO	HND	IA11200	12DEC1999	40337	49198	113505
9	SYD	HKG	IA10201	25DEC1999	34074	37410	107559
10	JRS	DEL	IA08901	02DEC1999	14434	16872	51300

### Creating a Random Sample without Replacement

You can also create a random sample **without replacement**. A sample without replacement **cannot** contain duplicate observations because after an observation is chosen from the original data set and output to the sample, it is programmatically excluded from being chosen again.

### Example

You can use a DO WHILE loop to avoid replacement as you create your random sample. In the following example,

- **Sasuser.Revenue** is the original data set.
- **sampszie** is the number of observations to read into the sample.
- **Work.Rsubset** is the data set that contains the random sample that you are creating.
- **obsleft** is the number of observations in the original data set that have not yet been considered for selection.
- **totobs** is the total number of observations in the original data set.
- **pickit** is the number of the observation to be read into the sample data set (if the RANUNI expression is true), and its starting value is 0.

With each iteration of the DO loop, pickit is incremented by 1. If the RANUNI expression is true, the observation that is indicated by the value of pickit is selected for the sample, and sampszie is decreased by 1. If the RANUNI expression is not true, the observation that is indicated by the value of pickit is not added to the sample. On each iteration of the loop, obsleft is decreased by 1 regardless of whether the observation is selected for the sample. The process ends when the value of sampszie is 0 and no additional observations are needed.

```
data work.rsubset(drop=obsleft sampszie);
  sampszie=10;
  obsleft=totobs;
do while(sampszie>0);
  pickit+1;
  if ranuni(0)<sampszie/obsleft then do;
    set sasuser.revenue point=pickit
    nobs=totobs;
    output;
    sampszie=sampszie-1;
  end;
  obsleft=obsleft-1;
end;
stop;
run;

proc print data=work.rsubset heading=h label;
  title 'A Random Sample without Replacement';
run;
```

Note that each observation is considered for selection once and only once.

Because the program above uses a seed of 0 for the RANUNI function, **Work.Rsubset** will contain different observations each time you run this code. Here is an example of the possible output.

**A Random Sample without Replacement**

Obs	Origin	Dest	FlightID	Date	Rev1st	RevBusiness	RevEcon
1	CBR	WLG	IA10500	28DEC1999	16092	21164	27324
2	CBR	WLG	IA10501	11DEC1999	13112	16280	23760
3	CCU	HKG	IA09901	25DEC1999	16272	19404	27225
4	CCU	SIN	IA09300	22DEC1999	18575	25350	29393
5	JED	LHR	IA08200	08DEC1999	23161	44820	51435
6	LHR	JNB	IA08301	09DEC1999	41940	76224	111456

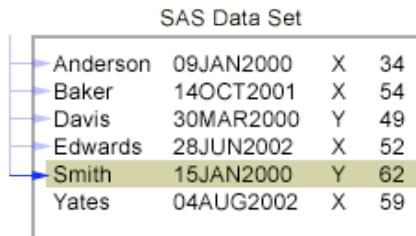
Obs	Origin	Dest	FlightID	Date	Rev1st	RevBusiness	RevEcon
7	LHR	RDU	IA00200	07DEC1999	20800	30520	79650
8	RDU	FRA	IA00301	03DEC1999	24654	31226	72540
9	RDU	LHR	IA00101	29DEC1999	20800	25070	75933
10	SFO	HNL	IA02900	26DEC1999	11856	16152	48216

## Using Indexes

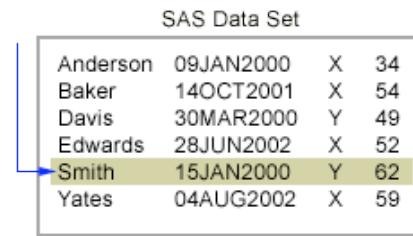
An **index** can help you quickly locate one or more particular observations that you want to read. An index is an optional file that you can create for a SAS data set in order to specify the location of observations based on values of one or more **key variables**. Indexes can provide direct access to observations in SAS data sets to

- yield faster access to small subsets of observations for WHERE processing
- return observations in sorted order for BY processing
- perform table lookup operations
- join observations
- modify observations.

Without an index, SAS accesses observations sequentially, in the order in which they are stored in a data set. For example, if you want to access the observation in the sample SAS data set shown below that has a value of *Smith* for the variable Name, SAS begins with the first observation and reads through each one until it reaches the observation that satisfies the condition.



An index stores values in ascending value order for a specific variable or variables and includes information about the location of those values within observations in the data file. That is, an index is composed of **value/identifier pairs** that enable you to locate an observation by value. For example, if you create an index on the sample SAS data set shown below based on the variable Name, SAS uses the index to find the observation that has a value of *Smith* for Name directly without having to read all the prior observations.



## Types of Indexes

You can create two types of indexes:

- a simple index
- a composite index.

A **simple index** consists of the values of one key variable, which can be character or numeric. When you create a simple index, SAS assigns the name of the key variable as the name of the index.

A **composite index** consists of the values of multiple key variables, which can be character, numeric, or a combination. The values of these key variables are concatenated to form a single value. For example, if an index is built on the key variables Lastname and Firstname, a value for the index is composed of the value for Lastname followed by the value for Firstname. When you create a composite index, you must specify a **unique** index name that is not the name of any existing variable or index in the data set.

Often, only the first variable of a composite index is used. For example, you could use the composite index specified in the example above (Lastname plus Firstname) for a WHERE expression that uses only Lastname. For example, the expression where Lastname='Smith' uses the the composite index because Lastname is the first variable in the index. That is, the value for Lastname is the first part of the value listed in the index.

## Creating Indexes in the DATA Step

To create an index at the same time that you create a data set, use the **INDEX= data set option** in the DATA statement.

### General form, DATA statement with the INDEX= option:

```
DATA SAS-data-file-name (INDEX=  
(index-specification-1</UNIQUE><...index-specification-n</UNIQUE>>));  
where
```

- *SAS-data-file-name* is a valid SAS data set name
- *index-specification* for a simple index is the name of the key variable
- *index-specification* for a composite index is (index-name=(variable-1...variable-n))
- the **UNIQUE** option specifies that values for the key variable must be unique for each observation.



SAS stores the name of a composite index exactly as you specify it in the INDEX= option. Therefore, if you want the name of your index to begin with a capital letter, you must specify the name with an initial capital letter in the INDEX= option.

You can create **multiple indexes** on a single SAS data set. However, keep in mind that creating and storing indexes does use system resources. Therefore, you should create indexes only on variables that are commonly used in a WHERE condition or on variables that are used to combine SAS data sets.



You can create an index on a SAS data file but not on a SAS data view.

The UNIQUE option guarantees that values for the key variable or the combination of a composite group of variables remain unique for every observation in the data set. In an existing data set, if the variable(s) on which you attempt to create a unique index has duplicate values, the index is not created. Similarly, if an update tries to add a record with a duplicate value for the index variable to that data set, the update is rejected. You will see examples of updating and maintaining indexes later in this lesson.

## Examples

The following example creates a simple index on the **Simple** data set. The index is named **Division**, and it contains values of the Division variable.

```
data simple (index=(division));
```

```
set sasuser.empdata;
run;
```

The following example creates two simple indexes on the **Simple2** data set. The first index is named **Division**, and it contains values of the Division variable. The second index is called **EmpID**, and it contains unique values of the EmpID variable.

```
data simple2 (index=(division empid/unique));
  set sasuser.empdata;
run;
```

The following example creates a composite index on the **Composite** data set. The index is named **Empdiv**, and it contains concatenated values of the Division variable and the EmpID variable.

```
data composite (index=(Empdiv=(division empid)));
  set sasuser.empdata;
run;
```

Complete the following statement to create a composite index named **Fromto** on the output data set that is based on the key variables Origin and Dest.

**Q.5.**

```
data flights _____
  set sasuser.revenue;
run;
```

When you create or use an index, you might want to verify that it has been created or used correctly. To display information in the SAS log concerning index creation or index usage, set the value of the **MSGLEVEL=** system option to I.

#### General form, **MSGLEVEL=** system option:

```
OPTIONS MSGLEVEL= N | I;
```

where

- **N** prints notes, warnings, and error messages only. This is the default.
- **I** prints additional notes or INFO messages pertaining to index usage, merge processing, and sort utilities along with standard notes, warnings, and error messages.

#### Example

The following code sets the **MSGLEVEL=** system option to I and creates the **Sasuser.Sale2000** data set with two indexes:

```
options msglevel=i;
data sasuser.sale2000(index=(origin
  flightdate=(flightid date)/unique));
infile sale2000 dsd;
input FlightID $ RouteID $ Origin $
  Dest $ Cap1st CapBusiness
  CapEcon CapTotal CapCargo
  Date Psgr1st PsgrBusiness
  PsgrEcon Rev1st RevBusiness
  RevEcon SaleMon $ CargoWgt RevCargo;
format date date9. ;
run;
```

Here are the messages that are written to the SAS log when the program above is submitted.

#### SAS Log

```
NOTE: The infile SALE2000 is:  
      File Name=C:\My SAS Files\9.0\sale2000.dat,  
      RECFM=V,LRECL=256  
NOTE: 153 records were read from the infile SALE2000.  
      The minimum record length was 82.  
      The maximum record length was 100.  
NOTE: The data set SASUSER.SALE2000 has 153 observations  
      and 19 variables.  
NOTE: Composite index flightdate has been defined.  
NOTE: Simple index origin has been defined.  
NOTE: DATA statement used (Total process time):  
      real time       1.08 seconds  
      cpu time       0.04 seconds
```

## Determining Whether SAS Is Using an Index

It is not always possible or more efficient for SAS to use an existing index to access specific observations directly. An index is **not used**

- with a subsetting IF statement in a DATA step
- with particular WHERE expressions
- if SAS determines it is more efficient to read the data sequentially.

### Example

You can use the MSGLEVEL= option to determine whether SAS is using an index. The following SAS log messages show examples of the INFO messages that indicate whether an index was used.

#### SAS Log

```
6 options msglevel=i;  
7  
8 proc print data=sasuser.revenue;  
9   where flightid ne 'IA11200';  
INFO: Index FlightID not used. Increasing bufno to 3 may help.
```

#### SAS Log

```
11 options msglevel=i;  
12  
13 data somflights;  
14   set sasuser.revenue;  
15   where flightid > 'IA11200';  
INFO: Index FlightID selected for WHERE clause optimization.
```

## Managing Indexes with PROC DATASETS

You have seen how to create an index at the same time that you create a data set. You can also create an index on an existing data set, or delete an index from a data set. One way to accomplish either

of these tasks is to rebuild the data set. However, rebuilding the data set is not the most efficient method for managing indexes.

You can use the DATASETS procedure to manage indexes on an existing data set. This uses fewer resources than it would to rebuild the data set. You use the **MODIFY statement** with the **INDEX CREATE statement** to create indexes on a data set. You use the MODIFY statement with the **INDEX DELETE statement** to delete indexes from a data set. You can also use the INDEX CREATE statement and the INDEX DELETE statement in the same step.

#### General form, PROC DATASETS to create and delete an index:

```
PROC DATASETS LIBRARY=libref <NOLIST>;
   MODIFY SAS-data-set-name;
   INDEX DELETE index-name;
   INDEX CREATE index-specification;
QUIT;
```

where

- *libref* points to the SAS library that contains *SAS-data-set-name*
- the **NOLIST** option suppresses the printing of the directory of SAS files in the SAS log and as ODS output
- *index-name* is the name of an existing index to be deleted
- *index-specification* for a simple index is the name of the key variable
- *index-specification* for a composite index is *index-name=(variable-1...variable-n)*.

The INDEX CREATE statement in PROC DATASETS **cannot** be used if the index to be created already exists. In this case, you must delete the existing index of the same name, then create the new index.



PROC DATASETS executes statements in order. Therefore, if you are deleting and creating indexes in the same step, you should delete indexes first so that the newly created indexes can reuse space of the deleted indexes.

#### Example

The following example creates an index named **Origin** on the **Sasuser.Sale2000** data set. **Origin** is a simple index that is based on the key variable **Origin**.

```
proc datasets library=sasuser nolist;
  modify sale2000;
  index create origin;
quit;
```

The following example first deletes the **Origin** index from the **Sasuser.Sale2000** data set, then creates two new indexes on the **Sasuser.Sale2000** data set. **FlightID** is a simple index that is based on the values of the key variable **FlightID**. **Fromto** is a composite index that is based on the concatenated values of the key variables **Origin** and **Dest**.

```
proc datasets library=sasuser nolist;
  modify sale2000;
  index delete origin;
  index create flightid;
  index create Fromto=(origin dest);
quit;
```

Complete the following step to create a new simple index on **Sasuser.Revenue** that is based on the key variable Date.

**Q.6.**

```
proc datasets library=sasuser nolist;
  modify revenue;
  index delete flightid;
  -----
  quit;
```

## Managing Indexes with PROC SQL

You can also create indexes on or delete indexes from an existing data set within a PROC SQL step. The **CREATE INDEX statement** enables you to create an index on a data set. The **DROP INDEX statement** enables you to delete an index from a data set.

**General form, PROC SQL to create and delete an index:**

```
PROC SQL;
  CREATE <UNIQUE> INDEX index-name
    ON table-name(column-name-1<...>,column-name-n);
  DROP INDEX index-name FROM table-name ;
  QUIT;
```

where

- *index-name* is the same as *column-name-1* if the index is based on the values of one column only
- *index-name* is not the same as any *column-name* if the index is based on multiple columns
- *table-name* is the name of the data set to which *index-name* is associated.

### Example

The following example creates a simple index named **Origin** on the **Sasuser.Sale2000** data set. The index is based on the values of the Origin column.

```
proc sql;
  create index origin on sasuser.sale2000(origin);
  quit;
```

The following example deletes the **Origin** index from the **Sasuser.Sale2000** data set and creates a new index named **Tofrom** that is based on the concatenation of the values from the columns Origin and Dest:

```
proc sql;
  create index Tofrom
    on sasuser.sale2000(origin, dest);
  drop index origin from sasuser.sale2000;
  quit;
```

Complete the following step to create a new simple index on **Sasuser.Revenue** that is based on the key variable Date.

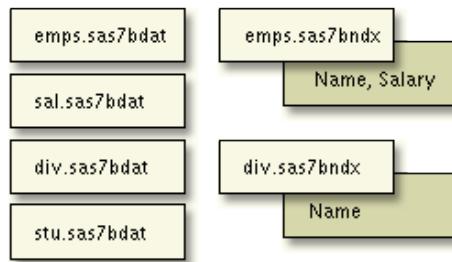
**Q.7.**

```
proc sql;
  -----
  on sasuser.revenue(date);
  quit;
```

## Documenting and Maintaining Indexes

Indexes are stored in the same SAS data library as the data set that they index, but in a separate SAS file from the data set. Index files have the **same name** as the associated data file, and have a member type of INDEX. There is only one index file per data set; all indexes for a data set are stored together in a single file.

The following image shows the relationship of SAS data set files and SAS index files in a Windows operating environment. Notice that although they have different file extensions, the index files have the same name as the data set with which they are associated. Also, notice that each index file can contain one or more indexes, and that different index files can contain indexes with identical names.



- Although index files are stored in the same location as the data sets to which they are associated,
- index files do not appear in the SAS Explorer window
- index files do not appear as separate files in z/OS (OS/390) operating environment file lists.

Sometimes, you might want to view a list of the indexes that exist for a data set. You might also want to see information about the indexes such as whether they are unique, and what key variables they use. Let's take a look at some ways to document indexes.

Information about indexes is stored in the descriptor portion of the data set. You can use either the **CONTENTS procedure** or the **CONTENTS statement in PROC DATASETS** to list information from the descriptor portion of a data set.

Output from the CONTENTS procedure or from the CONTENTS statement in PROC DATASETS contains the following information about the data set:

- general and summary information
- engine/host dependent information
- alphabetic list of variables and attributes
- alphabetic list of integrity constraints
- alphabetic list of indexes and attributes.

### General form, PROC CONTENTS:

```
PROC CONTENTS DATA=<libref.>SAS-data-set-name;  
      RUN;
```

where *SAS-data-set-name* specifies the data set for which the information will be listed.

### General form, PROC DATASETS with the CONTENTS statement:

```
PROC DATASETS <LIBRARY=libref> <NOLIST>;
```

```
CONTENTS DATA=<libref.>SAS-data-set-name;
QUIT;
```

where

- *SAS-data-set-name* specifies the data set for which the information will be listed
- the **NOLIST** option suppresses the printing of the directory of SAS files in the SAS log and as ODS output



If you use the **LIBRARY=** option, you do not need to specify a *libref* in the **DATA=** option. Likewise, if you specify a *libref* in the **DATA=** option, you do not need to use the **LIBRARY=** option.

## Example

The following example prints information about the **Sasuser.Sale2000** data set. Notice that the library is specified in the **LIBRARY=** option of the PROC DATASETS statement.

```
proc datasets library=sasuser nolist;
  contents data=sale2000;
  quit;
```

The following example also prints information about the **Sasuser.Sale2000** data set. Notice that the library is specified in the **CONTENTS** statement.

```
proc datasets nolist;
  contents data=sasuser.sale2000;
  quit;
```

The following example also prints information about the **Sasuser.Sale2000** data set:

```
proc contents data=sasuser.sale2000;
run;
```

The PROC DATASETS and PROC CONTENTS output from these programs is identical. The last piece of information printed in each set of output is a list of the indexes that have been created for **Sasuser.Sale2000**, as shown below.

Alphabetic List of Indexes and Attributes				
#	Index	Unique Option	# of Unique Values	Variables
1	Origin		52	
2	FlightDate	YES	26920	FlightID Date

You can also use either of these methods to list information about an **entire SAS library** rather than an individual data set. To list the contents of all files in a SAS data library with either PROC CONTENTS or with the **CONTENTS** statement in PROC DATASETS, you specify the keyword **\_ALL\_** in the **DATA=** option.

## Example

The following example prints information about all of the files in the **Work** data library:

```
proc contents data=work._all_;
run;
```

The following example also prints information about all of the files in the **Work** data library:

```
proc datasets library=work nolist;
  contents data=_all_;
quit;
```

Complete the following code to print a list of the contents of the **Sasuser.Revenue** data set.

**Q.8.** proc datasets nolist;

quit;

Remember that indexes are stored in a separate SAS file. When you perform maintenance tasks on a data set, there might be resulting effects on the index file. If you alter the variables or values within a data set, there might be a resulting effect on the value/identifier pairs within a particular index.

The following table describes the effects on an index or an index file that result from several common maintenance tasks.

Task	Effect
Add observation(s) to data set	Value/identifier pairs are added to index(es).
Delete observation(s) from data set	Value/identifier pairs are deleted from index(es).
Update observation(s) in data set	Value/identifier pairs are updated in index(es).
Delete data set	The index file is deleted.
Rebuild data set with DATA step	The index file is deleted.
Sort the data in place with the FORCE option in PROC SORT	The index file is deleted.

Let's take a look at some of the other common tasks that you might perform on your data sets, as well as the actions that SAS performs on the index files as a result.

## Copying Data Sets

You might want to copy an indexed data set to a new location. You can copy a data set with the **COPY statement** in a PROC DATASETS step. When you use the COPY statement to copy a data set that has an index associated with it, a new index file is automatically created for the new data file.

### General form, PROC DATASETS with the COPY statement:

```
PROC DATASETS LIBRARY=old-libref <NOLIST>;
  COPY OUT=new-libref;
  SELECT SAS-data-set-name;
  QUIT;
```

where

- *old-libref* names the library from which the data set will be copied
- *new-libref* names the library to which the data set will be copied
- *SAS-data-set-name* names the data set that will be copied.

You can also use the **COPY procedure** to copy data sets to a new location. Generally, PROC COPY functions the same as the COPY statement in the DATASETS procedure. When you use PROC COPY to copy a data set that has an index associated with it, a new index file is automatically created for the new data file. If you use the MOVE option in the COPY procedure, the index file is deleted from the original location and rebuilt in the new location.

### General form, PROC COPY step:

```
PROC COPY OUT=new-libref IN=old-libref  
  <MOVE>;  
  SELECT SAS-data-set-name(s);  
 RUN;  
 QUIT;
```

where

- *old-libref* names the library from which the data set will be copied
- *new-libref* names the library to which the data set will be copied
- *SAS-data-set-name* names the data set or data sets that will be copied.

### Examples

The following programs produce the same result. Both programs copy the **Sale2000** data set from the **Sasuser** library and place it in the **Work** library. Likewise, both of these programs cause a new index file to be created for **Work.Sale2000** that contains all indexes that exist in **Sasuser.Sale2000**.

```
proc datasets library=sasuser nolist;  
  copy out=work;  
  select sale2000;  
 quit;  
  
proc copy out=work in=sasuser;  
  select sale2000;  
 quit;
```



If you copy and paste a data set in either **SAS Explorer** or in **SAS Enterprise Guide**, a new index file is automatically created for the new data file.

Complete the following statement to copy the **Sasuser.Revenue** data set, along with any associated indexes, into the **Work** library.

**Q.9.** proc datasets library=sasuser nolist;

```
  _____  
  select revenue;  
  quit;
```

### Renaming Data Sets

Another common task is to rename an indexed data set. To preserve the index, you can use the **CHANGE statement** in PROC DATASETS to rename a data set. The index file will be automatically renamed as well.

### General form, PROC DATASETS with the CHANGE statement:

```
PROC DATASETS LIBRARY=libref <NOLIST>;  
  CHANGE old-data-set-name = new-data-set-name;  
  QUIT;
```

where

- *libref* names the SAS library where the data set is stored
- *old-data-set-name* is the current name of the data set
- *new-data-set-name* is the new name of the data set.

### Example

The following example copies the **Revenue** data set from **Sasuser** into **Work**, then renames the **Work.Revenue** data set to **Work.Income**. The index file that is associated with **Work.Revenue** is also renamed to **Work.Income**.

```
proc copy out=work in=sasuser;
  select revenue;
run;

proc datasets library=work nolist;
  change revenue=income;
quit;
```

## Renaming Variables

You have seen how to use PROC DATASETS to rename an indexed data set. Similarly, you might want to rename one or more variables within an indexed data set. In order to preserve any indexes that are associated with the data set, you can use the **RENAME statement** in the DATASETS procedure to rename variables.

### General form, PROC DATASETS with the RENAME statement:

```
PROC DATASETS LIBRARY=libref <NOLIST>;
  MODIFY SAS-data-set-name;
  RENAME old-var-name-1 = new-var-name-1
    <...old-var-name-n = new-var-name-n>;
QUIT;
```

where

- *libref* names the SAS library where the data set is stored
- *SAS-data-set-name* is the name of the data set that contains the variables to be renamed
- *old-var-name* is the original variable name
- *new-var-name* is the new name to be assigned to the variable.

When you use the RENAME statement to change the name of a variable for which there is a simple index, the statement also renames the index. If the variable that you are renaming is used in a composite index, the composite index automatically references the new variable name. However, if you attempt to rename a variable to a name that has already been used for a composite index, you will receive an error message.

### Example

The following example renames the variable **FlightID** as **FlightNum** in the **Work.Income** data set. If a simple index exists that is named **FlightID**, the index will be renamed **FlightNum**.

```
proc datasets library=work nolist;
  modify income;
  rename flightid=FlightNum;
quit;
```

Complete the following statement to rename the **Sasuser.Expenses** data set **Sasuser.Cost**.

**Q.10.** proc datasets library=sasuser nolist;
 \_\_\_\_\_
 quit;

## **Chapter Summary**

### **Creating a Systematic Sample from a Known Number of Observations**

Sometimes you might want to create a representative sample of a large data set. One type of representative sample, called a systematic sample, contains observations that are chosen from the original data set at regular intervals. You can use the POINT= option in the SET statement to make SAS read a specific observation into the sample. Since SAS uses direct-access read mode with the POINT= option, you must use a STOP statement to prevent the DATA step from looping continuously.

### **Creating a Systematic Sample from an Unknown Number of Observations**

You might want to create a systematic sample from a data set that contains an unknown number of observations. In order to be sure that your sample observations are chosen from regular intervals across the entire original data set, you need to know how many observations are in the data set. You can use the NOBS= option in the SET statement to determine how many observations are in the input data set. You can use the NOBS= option in conjunction with the POINT= option to direct SAS to read specific observations that will form a systematic sample.

### **Creating a Random Sample with Replacement**

Another type of representative sample that you might want to create is a random sample, in which observations are chosen randomly from the original data set. You can use the RANUNI function in conjunction with the CEIL function to generate a random integer. With a random integer, you can direct SAS to read a specific (but random) observation into the sample. When you create a random sample with replacement, each observation in the original data set has an equal chance of being chosen for inclusion in the sample each time SAS chooses an observation. That is, in a random sample with replacement, one observation may be chosen from the original data set and included in the sample multiple times.

### **Creating a Random Sample without Replacement**

You can also create a random sample without replacement. This means that once an observation has been included in the sample it is no longer eligible to be chosen again. You can use a DO WHILE loop to prevent replacement in your random samples.

## **Using Indexes**

An index is a SAS file that is associated with a data set and that contains information about the location and the values of key variables in the data set. Indexes enable SAS to directly access specific observations rather than having to read all observations sequentially. An index can be simple or composite.

### **Creating Indexes in the DATA Step**

You can create an index at the same time that you create a data set by using the INDEX= option in the DATA statement. Both simple and composite indexes can be unique, if there are no duplicate values for any key variable in the data set. You can create multiple indexes on one data set. You can use the MSGLEVEL= system option to write informational messages to the SAS log that pertain to indexes. Indexes can improve the efficiency of SAS, but there are certain instances where indexes will not improve efficiency and therefore will not be used.

### **Managing Indexes with PROC DATASETS and PROC SQL**

You can use the INDEX CREATE statement or the INDEX DELETE statement in PROC DATASETS to create an index on or delete an index from an existing data set. Using PROC DATASETS to manage indexes uses less system resources than it would to rebuild the data set and update indexes in the DATA step. If you want to delete an index and create an index in the same PROC DATASETS step, you should delete the old index before you create the new index so that SAS can reuse space from the deleted index. You can also use PROC SQL to create an index on or delete an index from an existing data set.

## Documenting and Maintaining Indexes

All indexes that are created for a particular data set are stored in one file in the same SAS data library as the data set. You can use PROC CONTENTS to print a list of all indexes that exist for a data set, along with other information about the data set. The CONTENTS statement of the PROC DATASETS step can generate the same list of indexes and other information about a data set.

Many of the maintenance tasks that you perform on your data sets will affect the index file that is associated with the data set. When you copy a data set with the COPY statement in PROC DATASETS, the index file is reconstructed for you. When you rename a data set or rename a variable with PROC DATASETS, the index file is automatically updated to reflect this change.

### Syntax

```
DATA SAS-data-set-name;
  point-variable=CEIL(RANUNI(seed)*nobs-variable);
  SET SAS-data-set-name POINT=point-variable NOBS=nobs-variable;
  STOP:
RUN;

OPTIONS MSGLEVEL= N | I;

DATA SAS-data-file-name (INDEX=
  (index-specification-1</UNIQUE><...index-specification-n</UNIQUE>>));
  SET SAS-data-set-name ;
RUN;

PROC DATASETS LIBRARY=libref <NOLIST>;
  MODIFY SAS-data-set-name;
  INDEX DELETE index-name ;
  INDEX CREATE index-specification;
QUIT;

PROC SQL;
  CREATE <UNIQUE> INDEX index-name
    ON table-name(column-name-1<...,column-name-n>);
  DROP INDEX index-name FROM table-name ;
QUIT;

PROC CONTENTS DATA=<libref.>SAS-data-set-name;
RUN;

PROC DATASETS <LIBRARY=libref> <NOLIST>;
  CONTENTS DATA=<libref.>SAS-data-set-name;
QUIT;

PROC DATASETS LIBRARY=old-libref <NOLIST>;
  COPY OUT=new-libref;
  SELECT SAS-data-set-name;
QUIT;

PROC COPY OUT=new-libref IN=old-libref <MOVE>;
  SELECT SAS-data-set-name(s);
RUN;
QUIT;
```

```

PROC DATASETS LIBRARY=libref <NOLIST>;
    CHANGE old-data-set-name = new-data-set-name;
QUIT;

PROC DATASETS LIBRARY=libref <NOLIST>;
    MODIFY SAS-data-set-name;
    RENAME old-var-name-1 = new-var-name-1 ;
    <...old-var-name-n = new-var-name-n> ;
QUIT;
```

## Sample Programs

### Creating a Systematic Sample from a Known Number of Observations

```

data sasuser.subset;
do pickit=1 to 142 by 15;
  set sasuser.revenue point=pickit;
  output;
end;
stop;
run;
```

### Creating a Systematic Sample from an Unknown Number of Observations

```

data sasuser.subset;
do pickit=1 to totobs by 10;
  set sasuser.revenue point=pickit
  nobs=totobs;
  output;
end;
stop;
run;
```

### Creating a Random Sample with Replacement

```

data work.rsubset (drop=i sampsize);
  sampsize=10;
  do i=1 to sampsize;
    pickit=ceil(ranuni(0)*totobs);
    set sasuser.revenue point=pickit
    nobs=totobs;
    output;
  end;
  stop;
run;

proc print data=work.rsubset label;
  title 'A Random Sample with Replacement';
run;
```

### Creating a Random Sample without Replacement

```

data work.rsubset(drop=obsleft sampsize);
  sampsize=10;
  obsleft=totobs;
  do while(sampsize>0);
    pickit+1;
    if ranuni(0)<sampsize/obsleft then do;
      set sasuser.revenue point=pickit
      nobs=totobs;
      output;
    end;
  end;
run;
```

```

    sampsize=sampsize-1;
  end;
obsleft=obsleft-1;
end;
stop;
run;

proc print data=work.rsubset heading=h label;
title 'A Random Sample without Replacement';
run;

```

### **Creating an Index in the DATA Step**

```

options msglevel=i;
data sasuser.sale2000(index=(origin FlightDate=
                           (flightid date)/unique));
infile 'sale2000.dat';
input FlightID $7. RouteID $7. Origin $3.
      Dest $3. Cap1st 8. CapBusiness 8.
      CapEcon 8. CapTotal 8. CapCargo 8.
      Date date9. Psgrlst 8./
      PsgrBusiness 8. PsgrEcon 8.
      Revlst dollar15.2
      RevBusiness dollar15.2
      RevEcon dollar15.2 SaleMon $7.
      CargoWgt 8./ RevCargo dollar15.2;
run;

```

### **Managing Indexes with PROC DATASETS**

```

proc datasets library=sasuser nolist;
  modify sale2000;
  index delete origin;
  index create flightid;
  index create Tofrom=(origin dest);
quit;

```

### **Managing Indexes with PROC SQL**

```

proc sql;
  create index Tofrom on
    sasuser.sale2000(origin, dest);
  drop index origin from sasuser.sale2000;
quit;

```

### **Points to Remember**

If you use direct-access read mode to create a representative sample of a data set, you need to use a STOP statement to prevent the DATA step from looping continuously.

An index can enable SAS to more efficiently access specific observations of a data set, but indexes use system resources and should only be created on variables that are commonly used in a WHERE condition or on variables that are used to combine SAS data sets.

An index is associated with a data set but is stored as a separate file. You can use PROC DATASETS, the CONTENTS statement, or the SAS Explorer to view a list of the indexes that exist for a data set. It is especially useful for you to view this information after you have performed maintenance tasks on your data set in order to be sure that the index file has been maintained.

## Chapter Quiz

Select the best answer for each question.

1. The variable that is created by the POINT= option is assigned a value
  - a. automatically during compilation of the DATA step.
  - b. automatically during execution of the DATA step.
  - c. during compilation of the DATA step, by program statements.
  - d. during execution of the DATA step, by program statements.
2. Which of the following programs correctly creates a systematic sample from a data set with an unknown number of observations and outputs these sample observations to a data set named **Sample**?
  - a. 

```
data sample;
set sasuser.sale2000 point=thisone nobs=totnum;
output;
stop;
run;
```
  - b. 

```
data sample;
do thisone=100 to totnum by 100;
set sasuser.sale2000 point=thisone nobs=totnum;
output;
end;
stop;
run;
```
  - c. 

```
data sample;
do thisone=100 to 1000 by 100;
set sasuser.sale2000 point=thisone;
output;
end;
stop;
run;
```
  - d. 

```
data sample;
do thisone=100 to totnum by 100;
set sasuser.sale2000 point=thisone nobs=totnum;
end;
run;
```
3. Which of the following expressions will generate a random integer between 1 and 50?
  - a. `ceil(ranuni(50))`
  - b. `ranuni(50)`
  - c. `ceil(ranuni(0)*50)`
  - d. `ceil(ranuni(0))*50`

4. An index
- is an optional file that is associated with a data set.
  - provides direct access to specific observations of a data set, based on the value of one or more key variables.
  - can be classified as simple or composite, either of which can consist of unique values.
  - all of the above
5. Which of the following correctly creates a data set named **Flights** from the **Sasuser.Revenue** data set, creates a composite index named **Fromto** that is based on the values of Origin and Dest, and prints informational messages about the index to the SAS log?
- ```
options msglevel=i;
data flights index=(Fromto=origin dest);
  set sasuser.revenue;
run;
```
  - ```
options msglevel=n;
data flights (index=(Fromto=origin dest));
  set sasuser.revenue;
run;
```
  - ```
options msglevel=i;
data flights (index=(Fromto=(origin dest)));
  set sasuser.revenue;
run;
```
  - ```
options msglevel=n;
data flights (index=Fromto);
  set sasuser.revenue;
run;
```
6. Which of the following is true?
- When you add observations to a data set, the index(es) are automatically updated with additional value/identifier pairs.
  - When you rename a variable that is used as the key variable in a simple index, you must re-create the index.
  - When you delete a data set, the index file remains until you delete it as well.
  - When you copy a data set with the COPY statement, you must also copy the index file in another step.
7. To create an index on an existing data set, you use
- PROC DATASETS.
  - PROC SQL.
  - the DATA step with the INDEX= option, to rebuild the data set.
  - any of the above
8. Which of the following correctly creates a simple index named **Origin** on the **Revenue** data set?
- ```
proc sql;
  create index origin on revenue(origin);
quit;
```

- b. proc sql;  
    modify revenue;  
    index=origin;  
    quit;
  - c. proc sql data=revenue;  
    create index origin;  
    quit;
  - d. proc sql;  
    index=origin on revenue;  
    quit;
- 9. To view a list of the indexes that are associated with a data set, you use
  - a. PROC COPY or the COPY statement in PROC DATASETS.
  - b. PROC CONTENTS or the CONTENTS statement in PROC DATASETS.
  - c. the MSGLEVEL= system option and a PROC PRINT step.
  - d. any of the above
- 10. Suppose that the **Sasuser.Revenue** data set has a simple index named **FlightID**. For which of the following programs will the index be used?
  - a. proc print data=sasuser.revenue;  
    where flightid ne 'IA11200';  
    run;
  - b. data someflights;  
    set sasuser.revenue;  
    where flightid > 'IA11200';  
    run;
  - c. data someflights;  
    set sasuser.revenue;  
    if flightid > 'IA11200';  
    run;
  - d. proc print data=sasuser.revenue;  
    where origin='RDU' or flightid='IA03400';  
    run;

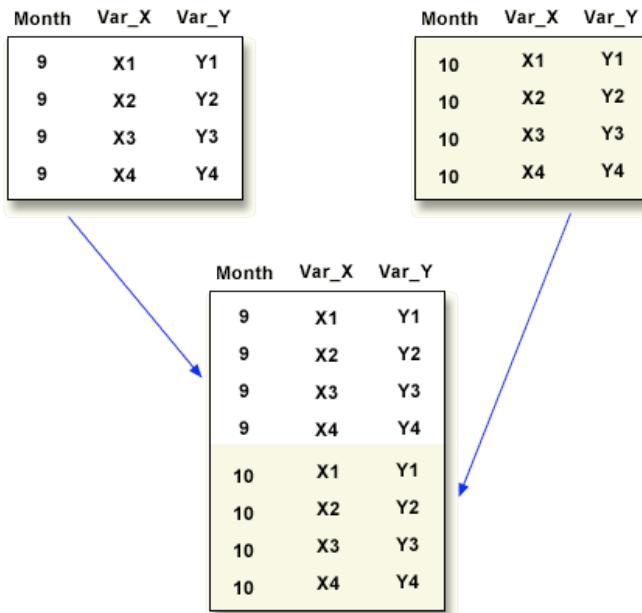
# Chapter 2

## Combining Data Vertically

### Introduction

Combining data vertically refers to the process of concatenating or interleaving data. In some cases the data may be in SAS data sets. In other cases the data may be stored in raw data files.

In this lesson you learn how to create a SAS data set by concatenating multiple raw data files using the FILENAME and INFILE statements. You also learn how to concatenate SAS data sets using PROC APPEND.



### Setting Up Filerefs for Practices in This Lesson

Submit this SAS program to set up filerefs for practices. If you have already done this in the current SAS session, you can skip this task. (This program is the same for all lessons.)

### Objectives

In this lesson, you learn to

- create a SAS data set from multiple raw data files using a FILENAME statement
- create a SAS data set from multiple raw data files using an INFILE statement with the FILEVAR= option
- append SAS data sets using the APPEND procedure.

## Using a FILENAME Statement

You already know that you can use a FILENAME statement to associate a fileref with a single raw data file. You can also use a FILENAME statement to concatenate raw data files by assigning a single fileref to the raw data files that you want to combine.

### General form, FILENAME statement:

```
FILENAME fileref ('external-file1' 'external-file2' ...'external-filen');
```

where

- *fileref* is any SAS name that is eight characters or fewer.
- '*external-file*' is the physical name of an external file. The physical name is the name that is recognized by the operating environment.



All of the file specifications must be enclosed in one set of parentheses.

When the fileref is specified in an INFILE statement, each raw data file that has been referenced can be sequentially read into a data set using an INPUT statement.



If you are not familiar with the content and structure of your raw data files, you can use PROC FSLIST to view them.

### Example

In the following program, the FILENAME statement creates the fileref **Qtr1**, which references the raw data files **Month1.dat**, **Month2.dat**, and **Month3.dat**. The files are stored in the **C:\Sasuser** directory in the Windows operating environment. In the DATA step, the INFILE statement identifies the fileref, and the INPUT statement describes the data, just as if **Qtr1** referenced a single raw data file.

```
filename qtr1 ('c:\sasuser\month1.dat''c:\sasuser\month2.dat'  
'c:\sasuser\month3.dat');

data work.firstqtr;  
  infile qtr1;  
  input Flight $ Origin $ Dest $  
    Date : date9. RevCargo : comma15.2;  
  run;
```



For convenience, the sample raw data files for this lesson are stored in the same physical location as your **Sasuser** directory. This location might differ from the location in the sample code above. If you want to submit the sample code, you need to determine the **physical location** for your **Sasuser** directory and point to that location instead.



For convenience, the sample raw data files for this lesson are stored in the same physical location as your **Sasuser** directory. This location might differ from the location in the sample code above. If you want to submit the sample code, you need to determine the **physical location** for your **Sasuser** directory and point to that location instead.

**Raw Data File Month1.dat (first five records)**

| ---     | 10  | --- | 20        | ---          | 30 | --- | 40 |
|---------|-----|-----|-----------|--------------|----|-----|----|
| IA10200 | SYD | HKG | 01JAN2000 | \$191,187.00 |    |     |    |
| IA10201 | SYD | HKG | 01JAN2000 | \$169,653.00 |    |     |    |
| IA10300 | SYD | CBR | 01JAN2000 | \$850.00     |    |     |    |
| IA10301 | SYD | CBR | 01JAN2000 | \$970.00     |    |     |    |
| IA10302 | SYD | CBR | 01JAN2000 | \$1,030.00   |    |     |    |

**Raw Data File Month2.dat (first five records)**

| ---     | 10  | --- | 20        | ---          | 30 | --- | 40 |
|---------|-----|-----|-----------|--------------|----|-----|----|
| IA10200 | SYD | HKG | 01FEB2000 | \$177,801.00 |    |     |    |
| IA10201 | SYD | HKG | 01FEB2000 | \$174,891.00 |    |     |    |
| IA10300 | SYD | CBR | 01FEB2000 | \$1,070.00   |    |     |    |
| IA10301 | SYD | CBR | 01FEB2000 | \$1,310.00   |    |     |    |
| IA10302 | SYD | CBR | 01FEB2000 | \$850.00     |    |     |    |

**Raw Data File Month3.dat (first five records)**

| ---     | 10  | --- | 20        | ---          | 30 | --- | 40 |
|---------|-----|-----|-----------|--------------|----|-----|----|
| IA10200 | SYD | HKG | 01MAR2000 | \$181,293.00 |    |     |    |
| IA10201 | SYD | HKG | 01MAR2000 | \$173,727.00 |    |     |    |
| IA10300 | SYD | CBR | 01MAR2000 | \$1,150.00   |    |     |    |
| IA10301 | SYD | CBR | 01MAR2000 | \$910.00     |    |     |    |
| IA10302 | SYD | CBR | 01MAR2000 | \$1,170.00   |    |     |    |

The SAS log indicates that the raw data files referenced by **Qtr1** are sequentially read into the SAS data set **Work.FirstQtr**.

**SAS Log**

```
9 filename qtr1 ('c:\sasuser\month1.dat'"c:\sasuser\month2.dat'
10      'c:\sasuser\month3.dat');

11 data work.firstqtr;
12   infile qtr1;
13   input Flight $ Origin $ Dest $;
14   Date : date9. RevCargo : commal5.2;
15 run;
```

NOTE: The infile QTR1 is:

File Name=c:\sasuser\month1.dat,  
File List=(‘c:\sasuser\month1.dat’ ‘c:\sasuser\month2.dat’  
‘c:\sasuser\month3.dat’),  
RECFM=V,LRECL=256

NOTE: The infile QTR1 is:

File Name=c:\sasuser\month2.dat,  
File List=(‘c:\sasuser\month1.dat’ ‘c:\sasuser\month2.dat’  
‘c:\sasuser\month3.dat’),  
RECFM=V,LRECL=256

NOTE: The infile QTR1 is:

File Name=c:\sasuser\month3.dat,  
File List=(‘c:\sasuser\month1.dat’ ‘c:\sasuser\month2.dat’  
‘c:\sasuser\month3.dat’),  
RECFM=V,LRECL=256

NOTE: 50 records were read from the infile QTR1.  
The minimum record length was 33.  
The maximum record length was 37.

NOTE: 50 records were read from the infile QTR1.  
The minimum record length was 33.  
The maximum record length was 37.

NOTE: 50 records were read from the infile QTR1.  
The minimum record length was 33.  
The maximum record length was 37.

NOTE: The data set WORK.FIRSTQTR has 150 observations  
and 5 variables.

NOTE: DATA statement used (Total process time):  
real time 4.02 seconds  
cpu time 0.93 seconds

The following PROC PRINT output shows a portion of the observations in the **Work.FirstQtr** data set.

```
proc print data=work.firstqtr (firstobs=45 obs=55);
  format date date9. revcargo dollar11.2;
run;
```

| Obs | Flight  | Origin | Dest | Date      | RevCargo     |
|-----|---------|--------|------|-----------|--------------|
| 45  | IA03505 | RDU    | BNA  | 01JAN2000 | \$2,697.00   |
| 46  | IA03904 | RDU    | MCI  | 01JAN2000 | \$4,161.00   |
| 47  | IA04503 | LHR    | GLA  | 01JAN2000 | \$3,498.00   |
| 48  | IA04703 | LHR    | FRA  | 01JAN2000 | \$3,225.00   |
| 49  | IA05002 | BRU    | LHR  | 01JAN2000 | \$1,989.00   |
| 50  | IA05003 | BRU    | LHR  | 01JAN2000 | \$2,379.00   |
| 51  | IA10200 | SYD    | HKG  | 01FEB2000 | \$177,801.00 |
| 52  | IA10201 | SYD    | HKG  | 01FEB2000 | \$174,891.00 |
| 53  | IA10300 | SYD    | CBR  | 01FEB2000 | \$1,070.00   |
| 54  | IA10301 | SYD    | CBR  | 01FEB2000 | \$1,310.00   |
| 55  | IA10302 | SYD    | CBR  | 01FEB2000 | \$850.00     |

**Q1.**

Write a statement to assign the fileref **Revenue** to the raw data files **Qtr3.dat** and **Qtr4.dat**. The files are stored in the **C:\Data** directory in the Windows operating environment.

## Using an INFILE Statement

You can make the process of concatenating raw data files more flexible by using an INFILE statement with the FILEVAR= option. The FILEVAR= option enables you to dynamically change the currently opened input file to a new input file.

### General form, INFILE statement with the FILEVAR= option:

**INFILE** *file-specification* **FILEVAR=** *variable*;

where

- **FILEVAR=** *variable* names a variable whose change in value causes the INFILE statement to close the current input file and open a new input file.
- *variable* contains a character string that is a physical filename.

When you use an INFILE statement with the FILEVAR= option, the file specification is a placeholder, not an actual filename or a fileref that had been assigned previously to a file. SAS uses this placeholder for reporting processing information to the SAS log. The file specification must conform to the same rules as a fileref.

When the INFILE statement executes, it reads from the file that the FILEVAR= variable specifies. Like automatic variables, this variable is not written to the data set.

### Example

Suppose you want to create a SAS data set that contains three months of data stored in three raw data files. The three months are the current month and the previous two months.



In the following INFILE statement, temp is an arbitrarily named placeholder, not an actual filename or a fileref that had been assigned to a file previously. The FILEVAR= variable, nextfile, contains the name of the raw data file to be read, for example, **Month9.dat**, **Month10.dat**, or **Month11.dat**. A RUN statement is not included because the program is not complete.

```
data work.quarter  
  infile temp filevar=nextfile;  
  input Flight $ Origin $ Dest $  
    Date : date9. RevCargo : commal5.2;
```

**Raw Data File Month9.dat (first five records)**

| ---     | 10  | --- | 20        | ---          | 30 | --- | 40 |
|---------|-----|-----|-----------|--------------|----|-----|----|
| IA10200 | SYD | HKG | 01SEP2000 | \$189,441.00 |    |     |    |
| IA10201 | SYD | HKG | 01SEP2000 | \$175,473.00 |    |     |    |
| IA10300 | SYD | CBR | 01SEP2000 | \$1,370.00   |    |     |    |
| IA10301 | SYD | CBR | 01SEP2000 | \$710.00     |    |     |    |
| IA10302 | SYD | CBR | 01SEP2000 | \$1,210.00   |    |     |    |

**Raw Data File Month10.dat (first five records)**

| ---     | 10  | --- | 20        | ---          | 30 | --- | 40 |
|---------|-----|-----|-----------|--------------|----|-----|----|
| IA10200 | SYD | HKG | 01OCT2000 | \$182,457.00 |    |     |    |
| IA10201 | SYD | HKG | 01OCT2000 | \$160,923.00 |    |     |    |
| IA10300 | SYD | CBR | 01OCT2000 | \$1,030.00   |    |     |    |
| IA10301 | SYD | CBR | 01OCT2000 | \$870.00     |    |     |    |
| IA10302 | SYD | CBR | 01OCT2000 | \$770.00     |    |     |    |

### Raw Data File Month11.dat (first five records)

| ---     | 10  | --- | 20        | ---          | 30 | --- | 40 |
|---------|-----|-----|-----------|--------------|----|-----|----|
| IA10200 | SYD | HKG | 01NOV2000 | \$176,637.00 |    |     |    |
| IA10201 | SYD | HKG | 01NOV2000 | \$164,997.00 |    |     |    |
| IA10300 | SYD | CBR | 01NOV2000 | \$1,230.00   |    |     |    |
| IA10301 | SYD | CBR | 01NOV2000 | \$1,230.00   |    |     |    |
| IA10302 | SYD | CBR | 01NOV2000 | \$790.00     |    |     |    |



You can also use multiple INFILE statements or operating system techniques to combine raw data files. However, this lesson discusses only the FILENAME statement and the INFILE statement with the FILEVAR= option.

## Assigning the Names of the Files to Be Read

The next step is to assign the names of the three files to be read to the variable `nextfile`:

```
data work.quarter;
  infile temp filevar=nextfile;
  input Flight $ Origin $ Dest $
    Date : date9. RevCargo : comma15.2;
```

In this case, let's use the raw data files **Month9.dat**, **Month10.dat**, and **Month11.dat**. Notice that the titles of the raw data files are very similar. They each start with "**Month**" and are followed by numeric characters and the file extension **.dat**:

```
Month9.dat
Month10.dat
Month11.dat
```

You can use an iterative DO loop and the PUT function to automatically change the values assigned to `nextfile`.

### Example

In the following code, the DO statement creates the index variable `i` and assigns it the values of 9, 10, and 11. The assignment statement then assigns the name of the raw data file to `nextfile` using the current value of `i` and the PUT function.

**Month9.dat**, **Month10.dat**, and **Month11.dat** are stored in the **C:\Sasuser** directory in the Windows operating environment. On the right side of the assignment statement, the text string `c:\sasuser\month` is concatenated with the current value of `i` using the double exclamation point (!!) concatenation operator. `c:\sasuser\monthi` is then concatenated with the text string `.dat`.

```
data work.quarter;
  do i = 9, 10, 11;
    nextfile="c:\sasuser\month"
      !!put(i,2.)!!".dat";
  infile temp filevar=nextfile;
  input Flight $ Origin $ Dest $
    Date : date9. RevCargo : comma15.2;
  end;
```

The following table shows the value of `nextfile` as the value of `i` changes.

| When i= | nextfile=              |
|---------|------------------------|
| 9       | c:\sasuser\month 9.dat |
| 10      | c:\sasuser\month10.dat |
| 11      | c:\sasuser\month11.dat |



Depending on the characters available on your keyboard, the symbol you use as the concatenation operator can be a double vertical bar (||), a double broken vertical bar (!!), or a double exclamation point (!!).

## Using the COMPRESS Function

Note the space between *month* and 9 in *c:\sasuser\month 9.dat*. You can eliminate the space by using the COMPRESS function.

| When i= | nextfile=              |
|---------|------------------------|
| 9       | c:\sasuser\month 9.dat |
| 10      | c:\sasuser\month10.dat |
| 11      | c:\sasuser\month11.dat |

### General form, COMPRESS function:

**COMPRESS**(*source*, <*characters-to-remove*>);

where

- *source* specifies a source string that contains the characters to remove
- *characters-to-remove* specifies the character or characters that SAS removes from the source string.



If the *characters-to-remove* is omitted, the COMPRESS function removes blank spaces from the source.

## Example

In the following code, the COMPRESS function removes blank spaces from the value of nextfile:

```
data work.quarter;
do i = 9, 10, 11;
  nextfile="c:\sasuser\month"!!put(i,2)!!".dat";
  nextfile=compress(nextfile,' ');
  infile temp filevar=nextfile;
  input Flight $ Origin $ Dest $
    Date : date9. RevCargo : comma15.2;
  end;
```

The COMPRESS function can be combined with the assignment statement for greater efficiency:

```
data work.quarter;
do i = 9, 10, 11;
  nextfile="c:\sasuser\month"!!compress(put(i,2)!!".dat",' ');
  infile temp filevar=nextfile;
  input Flight $ Origin $ Dest $
    Date : date9. RevCargo : comma15.2;
  end;
```

With the addition of the COMPRESS function, when the value of *i* equals 9, nextfile is assigned the correct value, *c:\sasuser\month9.dat*.

| <b>When i=</b> | <b>nextfile=</b>       |
|----------------|------------------------|
| 9              | c:\sasuser\month9.dat  |
| 10             | c:\sasuser\month10.dat |
| 11             | c:\sasuser\month11.dat |

Let's add a few more statements to the program. An OUTPUT statement within the DO loop outputs each record to the SAS data set **Work.Qarter** and a STOP statement prevents an infinite loop of the DATA step.

```
data work.quarter;
do i = 9, 10, 11;
  nextfile="c:\sasuser\month"
    !!compress(put(i,2.)!!".dat",' ');
infile temp filevar=nextfile;
input Flight $ Origin $ Dest $ Date : date9.
  RevCargo : commal5.2;
  output;
end;
stop;
```

The program is almost complete. However, there are several other statements that need to be added in order to read all of the input data.

### Using the END= Option

When you read past the last record in a raw data file, the DATA step normally stops processing. In this case, you need to read the last record in the first two raw data files. However, you do not want to read past the last record in either of those files because the DATA step will stop processing. You can use the END= option with the INFILE statement to determine when you are reading the last record in the last raw data file.

#### General form, INFILE statement with the END= option:

**INFILE** *file-specification* **END=variable**;

where

- *variable* names a variable that SAS sets to
- 0 when the current input data record **is not** the last record in the input file
- 1 when the current input record **is** the last record in the input file.



Like automatic variables, the END= variable is not written to the SAS data set.

The END= option enables you to name a variable whose value is controlled by SAS. The value of the variable is 0 when you are **not** reading the last record in an input file and 1 when you **are** reading the last record in an input file. You can test the value of the END= variable to determine if the DATA step should continue processing.

### Example

The END= variable lastobs is created in the INFILE statement. The DO UNTIL statement conditionally executes until the value of lastobs equals 1. A RUN statement completes the program.

```
data work.quarter;
do i = 9, 10, 11;
  nextfile="c:\sasuser\month"
```

```

!!compress(put(i,2.)!!".dat",'');
do until (lastobs);
  infile temp filevar=nextfile end=lastobs;
  input Flight $ Origin $ Dest $ Date : date9.
    RevCargo : comma15.2;
  output;
end;
end;
stop;
run;

```



For convenience, the sample raw data files for this lesson are stored in the same physical location as your **Sasuser** directory. This location might differ from the location in the sample code above. If you want to submit the sample code, you need to determine the **physical location** for your **Sasuser** directory and point to that location instead.



For convenience, the sample raw data files for this lesson are stored in the same physical location as your **Sasuser** directory. This location might differ from the location in the sample code above. If you want to submit the sample code, you need to determine the **physical location** for your **Sasuser** directory and point to that location instead.

PROC PRINT output shows a portion of the observations in the SAS data set **Work.Quarter**. A LABEL statement is used to assign the descriptive label **Month** to the variable i. Notice that the variables nextfile and lastobs are not written to the data set.

```

proc print data=work.quarter (firstobs=45 obs=55) label;
  label i='Month';
  format date date9. revcargo dollar11.2;
run;

```

| Obs | Month | Flight  | Origin | Dest | Date      | RevCargo     |
|-----|-------|---------|--------|------|-----------|--------------|
| 45  | 9     | IA04300 | LHR    | CDG  | 01SEP2000 | \$1,750.00   |
| 46  | 9     | IA05002 | BRU    | LHR  | 01SEP2000 | \$1,859.00   |
| 47  | 9     | IA05005 | BRU    | LHR  | 01SEP2000 | \$2,197.00   |
| 48  | 9     | IA05101 | LHR    | GVA  | 01SEP2000 | \$3,741.00   |
| 49  | 9     | IA05202 | GVA    | LHR  | 01SEP2000 | \$4,089.00   |
| 50  | 9     | IA05204 | GVA    | LHR  | 01SEP2000 | \$3,741.00   |
| 51  | 10    | IA10200 | SYD    | HKG  | 01OCT2000 | \$182,457.00 |
| 52  | 10    | IA10201 | SYD    | HKG  | 01OCT2000 | \$160,923.00 |
| 53  | 10    | IA10300 | SYD    | CBR  | 01OCT2000 | \$1,030.00   |
| 54  | 10    | IA10301 | SYD    | CBR  | 01OCT2000 | \$870.00     |
| 55  | 10    | IA10302 | SYD    | CBR  | 01OCT2000 | \$770.00     |

What happens when the value of last equals 1?

```

data work.revenue;
  do i = 1, 2, 3;
    ReadFile="c:\work\month"!!put(i,1.)!!".dat";
    do until (last);
      infile new filevar=readfile end=last;
      input Date : date9. Flight $
        RevCargo : comma15.2
        RevPassengers : comma15.2;

```

**Q.2.**

```

        output;
      end;
    end;
    stop;
run;

```

- A The DO UNTIL loop is exited.  
 B The DO UNTIL loop continues processing.

Which of the following statements correctly assigns the names of the raw data files **Movies\_April.dat**, **Movies\_May.dat**, and **Movies\_June.dat** to the variable `nextfile`? The values of `i` are *April*, *May*, and *June*.

- Q.3.**
- A `nextfile="c:\flights\movies_"`  
`!!compress(put(i,$4.)!!".dat",'');`  
 B `nextfile="c:\flights\movies_"`  
`!!compress(put(i,$5.)!!".dat",'');`  
 C `nextfile="c:\flights\movies_"`  
`!!compress(put(i,5.)!!".dat",'');`

- Q.4.** Write an assignment statement that uses the COMPRESS function to remove the character string *Special* from the values of the variable `Menu`. Name the new variable `Var`.
- 

## Using Date Functions

You can make your program more flexible by eliminating the need to include explicit month numbers in your SAS statements. To create a program that will always read the **current** month and the previous two months, you can use date functions to obtain the month number of today's date to begin the quarter.

### Example

In the following program, the MONTH and TODAY functions are used to obtain the value of the variable `monthnum`. The TODAY function returns the current date from the system clock as a SAS date value. The month number is then extracted from the current date using the MONTH function.

The value of `midmon` is calculated by subtracting 1 from the value of `monthnum`. The value of `lastmon` is then calculated by subtracting 1 from the values of `midmon`. The following table shows the values `monthnum`, `midmon`, and `lastmon` if the current date is October 22, 2002.

| Variable              | Value |
|-----------------------|-------|
| <code>monthnum</code> | 10    |
| <code>midmon</code>   | 9     |
| <code>lastmon</code>  | 8     |

In the previous example, the DO statement created the index variable `i` and assigned it the values of 9, 10, and 11. Here, the DO statement assigns `i` the values of `monthnum`, `midmon`, and `lastmon`:

```

data work.quarter (drop=monthnum midmon lastmon);
monthnum=month(today());
midmon=monthnum-1;
lastmon=midmon-1;
do i = monthnum, midmon, lastmon;

```

```

nextfile="c:\sasuser\month"
  !!compress(put(i,2.)!!".dat",' ');
do until (lastobs);
  infile temp filevar=nextfile end=lastobs;
  input Flight $ Origin $ Dest $ Date : date9.
    RevCargo : commal5.2;
  output;
end;
end;
stop;
run;

```

The following PROC PRINT output shows a portion of the observations in **Work.Quarter**.

```

proc print data=work.quarter (firstobs=45 obs=55) label;
  label i='Month';
  format date date9. revcargo dollar11.2;
run;

```

| Obs | Month | Flight  | Origin | Dest | Date      | RevCargo     |
|-----|-------|---------|--------|------|-----------|--------------|
| 45  | 10    | IA02700 | RDU    | MIA  | 01OCT2000 | \$6,028.00   |
| 46  | 10    | IA02703 | RDU    | MIA  | 01OCT2000 | \$5,764.00   |
| 47  | 10    | IA03503 | RDU    | BNA  | 01OCT2000 | \$3,625.00   |
| 48  | 10    | IA03504 | RDU    | BNA  | 01OCT2000 | \$2,697.00   |
| 49  | 10    | IA03804 | MSY    | RDU  | 01OCT2000 | \$6,517.00   |
| 50  | 10    | IA03903 | RDU    | MCI  | 01OCT2000 | \$6,213.00   |
| 51  | 9     | IA10200 | SYD    | HKG  | 01SEP2000 | \$189,441.00 |
| 52  | 9     | IA10201 | SYD    | HKG  | 01SEP2000 | \$175,473.00 |
| 53  | 9     | IA10300 | SYD    | CBR  | 01SEP2000 | \$1,370.00   |
| 54  | 9     | IA10301 | SYD    | CBR  | 01SEP2000 | \$710.00     |
| 55  | 9     | IA10302 | SYD    | CBR  | 01SEP2000 | \$1,210.00   |

### Using the INTNX Function

In the previous example the current month was October. What happens if the current month is January or February?

Suppose the current date is February 16, 2003. Using the following program, the values for midmon (January) and lastmon (December) would be 1 and 0 respectively. Since there is no "0" month, the program would fail to read the third raw data file.

| Variable | Value |
|----------|-------|
| monthnum | 2     |
| midmon   | 1     |
| lastmon  | 0     |

```

data work.quarter (drop=monthnum midmon lastmon);
  monthnum=month(today());
  midmon=monthnum-1;

```

```

lastmon=midmon-1;
do i = monthnum, midmon, lastmon;
  nextfile="c:\sasuser\month"
    !!compress(put(i,2.)!!".dat",' ');
  do until (lastobs);
    infile temp filevar=nextfile end=lastobs;
    input Flight $ Origin $ Dest $ Date : date9.
      RevCargo : commal5.2;
    output;
  end;
end;
stop;
run;

```

You can use the INTNX function with the TODAY and MONTH functions to correctly determine the values of midmon and lastmon for any given date. Remember that the INTNX function increments a date, time, or datetime value by a given interval or intervals, and returns a date, time, or datetime value.

### Example

Suppose the current date is January 30, 2003. In the following program monthnum is assigned a value of 1 using the TODAY and MONTH functions. The INTNX function is used with the TODAY and MONTH functions to assign a value of 12 to midmon and a value of 11 to lastmon.

| Variable | Value |
|----------|-------|
| monthnum | 1     |
| midmon   | 12    |
| lastmon  | 11    |

```

data work.quarter (drop=monthnum midmon lastmon);
monthnum=month(today());
midmon=month(intnx('month',today(),-1));
lastmon=month(intnx('month',today(),-2));
do i = monthnum, midmon, lastmon;
  nextfile="c:\sasuser\month"!!compress(put(i,2.)!!".dat",' ');
  do until (lastobs);
    infile temp filevar=nextfile end=lastobs;
    input Flight $ Origin $ Dest $ Date : date9.
      RevCargo : commal5.2;
    output;
  end;
end;
stop;
run;

```

The following PROC PRINT output shows a portion of the observations in **Work.Quarter**.

```

proc print data=work.quarter (firstobs=45 obs=55) label;
  label i='Month';
  format date date9. revcargo dollar11.2;
run;

```

| Obs | Month | Flight  | Origin | Dest | Date      | RevCargo     |
|-----|-------|---------|--------|------|-----------|--------------|
| 45  | 1     | IA03505 | RDU    | BNA  | 01JAN2000 | \$2,697.00   |
| 46  | 1     | IA03904 | RDU    | MCI  | 01JAN2000 | \$4,161.00   |
| 47  | 1     | IA04503 | LHR    | GLA  | 01JAN2000 | \$3,498.00   |
| 48  | 1     | IA04703 | LHR    | FRA  | 01JAN2000 | \$3,225.00   |
| 49  | 1     | IA05002 | BRU    | LHR  | 01JAN2000 | \$1,989.00   |
| 50  | 1     | IA05003 | BRU    | LHR  | 01JAN2000 | \$2,379.00   |
| 51  | 12    | IA10200 | SYD    | HKG  | 01DEC2000 | \$188,277.00 |
| 52  | 12    | IA10201 | SYD    | HKG  | 01DEC2000 | \$178,965.00 |
| 53  | 12    | IA10300 | SYD    | CBR  | 01DEC2000 | \$910.00     |
| 54  | 12    | IA10301 | SYD    | CBR  | 01DEC2000 | \$950.00     |
| 55  | 12    | IA10302 | SYD    | CBR  | 01DEC2000 | \$1,310.00   |

**Q.5.**

Write an assignment statement to create a variable named ThisYear that contains the value of the current year based on the value of today's date.

**Q.6.**

If the current date is April 6, 2003, what is the value of FirstMon given the following statement?

```
FirstMon=month(intnx('month',today(),-3));
```

- A 1
- B 3
- C 7

## Appending SAS Data Sets

Now that you have seen several methods for concatenating raw data files, let's take a look at how you can use the APPEND procedure to concatenate two SAS data sets.

### General form, PROC APPEND:

```
PROC APPEND BASE=SAS-data-set DATA=SAS-data-set;
  RUN;
```

where

- **BASE=SAS-data-set** names the data set to which you want to add observations
- **DATA=SAS-data-set** names the SAS data set containing observations that you want to append to the end of the SAS data set specified in the **BASE=** argument.

PROC APPEND only reads the data in the DATA= SAS data set, not the BASE= SAS data set. PROC APPEND concatenates data sets even though there may be variables in the BASE= data set that do not exist in the DATA= data set.

When the BASE= data set contains more variables than the DATA= data set, missing values for the additional variables are assigned to the observations that are read in from the DATA= data set and a warning message is written to the SAS log.

## Example

The SAS data sets **Work.Cap2001** and **Work.Capacity** both contain the following variables: Cap1st, CapBusiness, CapEcon, Dest, FlightID, Origin, and RouteID. However, the BASE= data set (**Work.Cap2001**) contains an additional variable, Date, that is not included in the DATA= data set (**Work.Capacity**).



If you want to submit the sample code shown in this section, you need to copy the data sets shown from the **Sasuser** library to the **Work** library.

When the following program is submitted, the SAS log indicates that the variable Date was not found in the DATA= file.

```
proc append base=work.cap2001  
          data=work.capacity;  
run;
```

### SAS Log

```
NOTE: Appending WORK.CAPACITY to WORK.CAP2001.  
WARNING: Variable Date was not found on DATA file.  
NOTE: There were 50 observations read from the data set WORK.CAPACITY.  
NOTE: 50 observations added.  
NOTE: The data set WORK.CAP2001 has 100 observations and 8 variables.  
NOTE: PROCEDURE APPEND used (Total process time):
```

PROC PRINT output of the appended version of **Work.Cap2001** shows that missing values have been assigned to Date in the observations that were read in from the DATA= data set.

```
proc print data=work.cap2001 (firstobs=45 obs=55);  
run;
```

| Obs | FlightID | RouteID | Origin | Dest | Cap1st | CapBusiness | CapEcon | Date      |
|-----|----------|---------|--------|------|--------|-------------|---------|-----------|
| 45  | IA02503  | 0000025 | RDU    | IND  | 12     | 0           | 138     | 21JAN2000 |
| 46  | IA02504  | 0000025 | RDU    | IND  | 12     | 0           | 138     | 24JAN2000 |
| 47  | IA02600  | 0000026 | IND    | RDU  | 12     | 0           | 138     | 02JAN2000 |
| 48  | IA02605  | 0000026 | IND    | RDU  | 12     | 0           | 138     | 05JAN2000 |
| 49  | IA02603  | 0000026 | IND    | RDU  | 12     | 0           | 138     | 16JAN2000 |
| 50  | IA02703  | 0000027 | RDU    | MIA  | 12     | 0           | 138     | 17JAN2000 |
| 51  | IA00100  | 0000001 | RDU    | LHR  | 14     | 30          | 163     | .         |
| 52  | IA00201  | 0000002 | LHR    | RDU  | 14     | 30          | 163     | .         |
| 53  | IA00300  | 0000003 | RDU    | FRA  | 14     | 30          | 163     | .         |
| 54  | IA00400  | 0000004 | FRA    | RDU  | 14     | 30          | 163     | .         |
| 55  | IA00500  | 0000005 | RDU    | JFK  | 16     | .           | 251     | .         |



You can also use the DATA step SET statement to combine SAS data vertically. If multiple data set names appear in the SET statement, the resulting output data set is a concatenation of all the data sets listed. Unlike the APPEND procedure, the SET statement in the DATA step reads all observations in both input data sets in order to concatenate them. Therefore, the APPEND

procedure is more efficient than the SET statement in the DATA step for concatenating data sets because it reads only the data in the DATA= data set.

In the following program, SAS reads all of the observations from **Work.Cap2001**, then all of the observations from **Work.Capacity**.

```
data work.new;
  set work.cap2001 work.capacity;
run;
```



You can also use the SQL procedure to combine SAS data vertically. For information on using the SQL procedure to combine data vertically, see the lesson **Combining Tables Vertically Using PROC SQL**.

### Using the FORCE Option

In the previous example, the DATA= data set (**Work.Capacity**) contained **fewer** variables than the BASE= data set (**Work.Cap2001**). However, you may need to append data sets when the DATA= data set contains **more** variables than the BASE= data set.

You must use the FORCE option with the APPEND procedure to concatenate data sets when the DATA= data set contains variables that are not in the BASE= data set.

#### General form, PROC APPEND with the FORCE option:

```
PROC APPEND BASE=SAS-data-set DATA=SAS-data-set <FORCE>;
```



The FORCE option can cause loss of data due to truncation or dropping of variables.

When you use the FORCE option, the structure of the BASE= data set is used for the appended data set.

#### Example

Remember that the SAS data sets **Work.Cap2001** and **Work.Capacity** both contain the following variables: Cap1st, CapBusiness, CapEcon, Dest, FlightID, Origin, and RoutID. In this case, the **DATA=** data set (**Work.Cap2001**) contains an additional variable, Date, that is not included in the **BASE=** data set (**Work.Capacity**).

When the following program is submitted, the SAS log indicates that the data sets were not appended because the variable Date was not found in the BASE= file.

```
proc append base=work.capacity
  data=work.cap2001;
run;
```

#### SAS Log

```
NOTE: Appending WORK.CAP2001 to WORK.CAPACITY.
WARNING: Variable Date was not found on 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 WORK.CAPACITY has 50 observations and 7 variables.
NOTE: Statements not processed because of errors noted above.
```

NOTE: PROCEDURE APPEND used (Total process time):

real time 0.02 seconds  
cpu time 0.03 seconds

NOTE: The SAS System stopped processing this step because of errors.

When the FORCE option is used with PROC APPEND, the SAS log indicates that observations have been read from the DATA= data set, but that dropping or truncating will occur.

```
proc append base=work.capacity  
          data=work.cap2001 force;  
run;
```

### SAS Log

NOTE: Appending WORK.CAP2001 to WORK.CAPACITY.

**WARNING: Variable Date was not found on BASE file.**

**NOTE: FORCE is specified, so dropping/truncating will occur.**

NOTE: There were 50 observations read from the data set WORK.CAP2001.

NOTE: 50 observations added.

NOTE: The data set WORK.CAPACITY has 100 observations and 7 variables.

NOTE: PROCEDURE APPEND used (Total process time):

real time 0.03 seconds  
cpu time 0.03 seconds

PROC PRINT output shows that the variable Date has been dropped from the appended version of **Work.Capacity**.

```
proc print data=work.capacity (firstobs=46 obs=55);  
run;
```

| Obs | FlightID | RouteID | Origin | Dest | Cap1st | CapBusiness | CapEcon |
|-----|----------|---------|--------|------|--------|-------------|---------|
| 46  | IA04600  | 0000046 | GLA    | LHR  | 14     | .           | 125     |
| 47  | IA04700  | 0000047 | LHR    | FRA  | 14     | .           | 125     |
| 48  | IA04800  | 0000048 | FRA    | LHR  | 14     | .           | 125     |
| 49  | IA04900  | 0000049 | LHR    | BRU  | 14     | .           | 125     |
| 50  | IA05000  | 0000050 | BRU    | LHR  | 14     | .           | 125     |
| 51  | IA00100  | 0000001 | RDU    | LHR  | 14     | 30          | 163     |
| 52  | IA00101  | 0000001 | RDU    | LHR  | 14     | 30          | 163     |
| 53  | IA00201  | 0000002 | LHR    | RDU  | 14     | 30          | 163     |
| 54  | IA00301  | 0000003 | RDU    | FRA  | 14     | 30          | 163     |
| 55  | IA00603  | 0000006 | JFK    | RDU  | 16     | 34          | 251     |

### Appending Variables with Different Lengths

If the DATA= data set contains variables that are longer than the variables in the BASE= data set, the FORCE option must be used with PROC APPEND. Using the FORCE option enables you to append the data sets. However, the DATA= variable values will be truncated.

## Example

In the SAS data set **Work.Acities**, the variable City has a length of 22. In the SAS data set **Work.WestAust**, City has a length of 50. You can use the CONTENTS procedure to view the attributes of the variables in each data set.

```
proc contents  
    data=work.acities;  
run;
```

Partial Output

| Alphabetic List of Variables and Attributes |          |      |     |                                  |
|---------------------------------------------|----------|------|-----|----------------------------------|
| #                                           | Variable | Type | Len | Label                            |
| 1                                           | City     | Char | 22  | City Where Airport is Located    |
| 2                                           | Code     | Char | 3   | Start Point                      |
| 4                                           | Country  | Char | 40  | Country Where Airport is Located |
| 3                                           | Name     | Char | 50  | Airport Name                     |

```
proc contents  
    data=work.westaust;  
run;
```

Partial Output

| Alphabetic List of Variables and Attributes |          |      |     |                                  |
|---------------------------------------------|----------|------|-----|----------------------------------|
| #                                           | Variable | Type | Len | Label                            |
| 2                                           | City     | Char | 50  | City Where Airport is Located    |
| 1                                           | Code     | Char | 3   | Airport Code                     |
| 3                                           | Country  | Char | 40  | Country Where Airport is Located |
| 4                                           | Name     | Char | 50  | Airport Name                     |

When the following program is submitted, the SAS log indicates that the data sets were not appended due to different lengths for City in the BASE= and DATA= data sets.

```
proc append base=work.acities  
    data=work.westaust;  
run;
```

SAS Log

NOTE: Appending WORK.WESTAUST to WORK.ACITIES.  
**WARNING: Variable City has different lengths on BASE and**

**DATA files (BASE 22 DATA 50).**

**ERROR: No appending done because of anomalies listed above.**

**Use FORCE option to append these files.**

NOTE: 0 observations added.

NOTE: The data set WORK.ACITIES has 50 observations and 4 variables.

NOTE: Statements not processed because of errors noted above.

NOTE: PROCEDURE APPEND used (Total process time):

```
real time      1.44 seconds  
cpu time      0.06 seconds
```

NOTE: The SAS System stopped processing this step because of errors.

When the FORCE option is used, the SAS log indicates that the data sets are appended, but that dropping or truncating will occur.

```
proc append base=work.acities
            data=work.airports force;
run;
```

### SAS Log

```
NOTE: Appending WORK.WESTAUST to WORK.ACITIES.
WARNING: Variable City has different lengths on BASE and DATA files

(BASE 22 DATA 50).
NOTE: FORCE is specified, so dropping/truncating will occur.
NOTE: There were 50 observations read from the data set WORK.WESTAUST.
NOTE: 50 observations added.
NOTE: The data set WORK.ACITIES has 100 observations and 4 variables.
NOTE: PROCEDURE APPEND used (Total process time):
      real time      1.44 seconds
      cpu time      0.06 seconds
```

PROC CONTENTS output for the appended version of **Work.Acities** shows that the variable City has retained a length of 22 from the BASE= data set. Also notice that the variable Code has retained the label **Start Point** from the BASE= data set.

```
proc contents
      data=work.acities;
run;
```

| Alphabetic List of Variables and Attributes |          |      |     |                                  |
|---------------------------------------------|----------|------|-----|----------------------------------|
| #                                           | Variable | Type | Len | Label                            |
| 1                                           | City     | Char | 22  | City Where Airport is Located    |
| 2                                           | Code     | Char | 3   | Start Point                      |
| 4                                           | Country  | Char | 40  | Country Where Airport is Located |
| 3                                           | Name     | Char | 50  | Airport Name                     |

PROC PRINT output shows that some of the values of City are truncated in the appended version of **Work.Acities**.

```
proc print data=work.acities (firstobs=45 obs=55);
run;
```

| Obs | City                   | Code | Name                                 | Country   |
|-----|------------------------|------|--------------------------------------|-----------|
| 45  | Portland, ME           | PWM  | Portland International Jetport       | USA       |
| 46  | Raleigh-Durham, NC     | RDU  | Raleigh-Durham International Airport | USA       |
| 47  | Seattle, WA            | SEA  | Seattle-Tacoma International Airport | USA       |
| 48  | San Francisco, CA      | SFO  | San Francisco International Airport  | USA       |
| 49  | Singapore              | SIN  | Changi International Airport         | Singapore |
| 50  | Sydney, New South Wale | SYD  | Kingsford Smith                      | Australia |
| 51  | Argyle Downs, Western  | AGY  |                                      | Australia |
| 52  | Albany, Western Austra | ALH  |                                      | Australia |
| 53  | Big Bell, Western Aust | BBE  |                                      | Australia |

| Obs | City                   | Code | Name | Country   |
|-----|------------------------|------|------|-----------|
| 54  | Bedford Downs, Western | BDW  |      | Australia |
| 55  | Beagle Bay, Western Au | BEE  |      | Australia |

## Appending Variables with Different Types

If the DATA= data set contains a variable that does not have the same type as the corresponding variable in the BASE= data set, the FORCE option must be used with PROC APPEND. Using the FORCE option enables you to append the data sets. However, missing values are assigned to the DATA= variable values for the variable whose type did not match.

### Example

In the SAS data set **Work.Allemps**, the variable Phone is a character variable. In the SAS data set **Work.Newemps**, Phone is a numeric variable. You can use PROC CONTENTS to view the attributes of the variables in each data set.

```
proc contents data=work.allemps;
run;
```

**Partial Output**

| Alphabetic List of Variables and Attributes |          |      |     |
|---------------------------------------------|----------|------|-----|
| #                                           | Variable | Type | Len |
| 5                                           | Division | Char | 30  |
| 1                                           | EmplID   | Char | 6   |
| 2                                           | LastName | Char | 15  |
| 4                                           | Location | Char | 13  |
| 3                                           | Phone    | Char | 4   |

```
proc contents data=work.newemps;
run;
```

**Partial Output**

| Alphabetic List of Variables and Attributes |          |      |     |
|---------------------------------------------|----------|------|-----|
| #                                           | Variable | Type | Len |
| 4                                           | Division | Char | 30  |
| 1                                           | EmplID   | Char | 6   |
| 2                                           | LastName | Char | 15  |
| 3                                           | Location | Char | 13  |
| 5                                           | Phone    | Num  | 8   |

When the following program is submitted, the SAS log indicates that there is a type mismatch for the variable Phone and that data sets were not appended.

```
proc append base=work.allemps
            data=work.newemps;
run;
```

### SAS Log

|                                                                |
|----------------------------------------------------------------|
| NOTE: Appending WORK.NEWEMPS to WORK.ALLEMPs.                  |
| WARNING: Variable Phone not appended because of type mismatch. |
| ERROR: No appending done because of anomalies listed above.    |
| Use FORCE option to append these files.                        |

```

NOTE: 0 observations added.
NOTE: The data set WORK.ALLEMP has 550 observations and 5 variables.
NOTE: Statements not processed because of errors noted above.
NOTE: PROCEDURE APPEND used (Total process time):
      real time      0.08 seconds
      cpu time      0.01 seconds
NOTE: The SAS System stopped processing this step because of errors.

```

When the FORCE option is used, the SAS log indicates that the data sets are appended, but that the variable Phone is not appended due to the type mismatch.

```

proc append base=work.alldeps
            data=work.newdeps force;
run;

```

#### SAS Log

```

NOTE: Appending WORK.NEWDEPS to WORK.ALLEMP.
WARNING: Variable Phone not appended because of type mismatch.
NOTE: FORCE is specified, so dropping/truncating will occur.
NOTE: There were 19 observations read from the data set WORK.NEWDEPS.
NOTE: 19 observations added.
NOTE: The data set WORK.ALLEMP has 569 observations and 5 variables.
NOTE: PROCEDURE APPEND used (Total process time):
      real time      0.05 seconds
      cpu time      0.02 seconds

```

PROC CONTENTS output for the appended version of **Work.Alldeps** shows that the variable Phone has retained the type of character from the BASE= data set.

```

proc contents
            data=work.alldeps;
run;

```

#### Partial Output

| Alphabetic List of Variables and Attributes |          |      |     |
|---------------------------------------------|----------|------|-----|
| #                                           | Variable | Type | Len |
| 5                                           | Division | Char | 30  |
| 1                                           | EmpID    | Char | 6   |
| 2                                           | LastName | Char | 15  |
| 4                                           | Location | Char | 13  |
| 3                                           | Phone    | Char | 4   |

PROC PRINT output of the appended version of **Work.Alldeps** shows that the values for Phone are missing in the records that were read in from the DATA= data set.

```

proc print data=work.alldeps (firobs=45 obs=55);
run;

```

| Obs | EmpID  | LastName   | Phone | Location   | Division           |
|-----|--------|------------|-------|------------|--------------------|
| 45  | E00213 | DICKEY     | 1519  | CARY       | AIRPORT OPERATIONS |
| 46  | E00226 | BAUCOM     | 1124  | CARY       | AIRPORT OPERATIONS |
| 47  | E00231 | SPENCER    | 2868  | CARY       | AIRPORT OPERATIONS |
| 48  | E00236 | BAILEY     | 1088  | CARY       | AIRPORT OPERATIONS |
| 49  | E00243 | FILIPOWSKI | 1635  | CARY       | AIRPORT OPERATIONS |
| 50  | E00249 | YUAN       | 3241  | CARY       | AIRPORT OPERATIONS |
| 51  | E00490 | CANCELLO   |       | ROME       | FINANCE & IT       |
| 52  | E00496 | PRESTON    |       | LONDON     | FINANCE & IT       |
| 53  | E00499 | ZILSTORFF  |       | COPENHAGEN | AIRPORT OPERATIONS |
| 54  | E00500 | LEY        |       | FRANKFURT  | FINANCE & IT       |
| 55  | E00503 | BRAMMER    |       | COPENHAGEN | SALES & MARKETING  |

**Q.7.**

Write a PROC APPEND statement to append the SAS data set **Flights.December** to the SAS data set **Flights.Y2002**. The variables and variable attributes of the data sets are identical.

run;

**Q.8.**

Is it necessary to use the FORCE option if you are using PROC APPEND to append the SAS data set **Training.Denver** to the SAS data set **Training.Raleigh**?

| Data Set Description for Training.Raleigh |      |        |
|-------------------------------------------|------|--------|
| Variable                                  | Type | Length |
| Division                                  | num  | 8      |
| EmployeeID                                | num  | 8      |
| Name                                      | num  | 8      |
| Extension                                 | num  | 8      |

| Data Set Description for Training.Denver |      |        |
|------------------------------------------|------|--------|
| Variable                                 | Type | Length |
| Division                                 | num  | 8      |
| EmployeeID                               | char | 8      |
| Name                                     | num  | 8      |
| Extension                                | num  | 8      |

A No

B Yes

## Additional Features

In addition to the methods for appending raw data files that were discussed earlier in this lesson, you can also append raw data files using a SAS data set or an external file that contains the names of the raw data files to be appended.

### Storing Raw Data Filenames in a SAS Data Set

In the following program, five raw data files, **Route1.dat**, **Route2.dat**, **Route3.dat**, **Route4.dat**, and **Route5.dat**, are concatenated to create the SAS data set **Work.NewRoute**. The names of the raw data files are stored in the SAS data set **Sasuser.Rawdata**, which is referenced using a SET statement. The name of the FILEVAR= variable, readit, is the name of the variable in **Sasuser.Rawdata** whose value is the name of the file to be read.

**SAS Data Set Sasuser.Rawdata**

| Obs | readit     |
|-----|------------|
| 1   | route1.dat |
| 1   | route2.dat |
| 3   | route3.dat |
| 4   | route4.dat |
| 5   | route5.dat |

```
data work.newroute;
  set sasuser.rawdata;
  infile in filevar = readit end = lastfile;
  do while(lastfile = 0);
    input @1 RouteID $7. @8 Origin $3. @11 Dest $3.
      @14 Distance 5. @19 Farelst 4.
      @23 FareBusiness 4. @27 FareEcon 4.
      @31 FareCargo 5.;

    output;
  end;
run;
```

### Storing Raw Data Filenames in an External File

In the following program, **Route1.dat**, **Route2.dat**, **Route3.dat**, **Route4.dat**, and **Route5.dat** are also concatenated to create the SAS data set **Work.NewRoute**. In this example, the names of the raw data files are stored in the external file **Rawdatafiles.dat**, which is referenced in the first INFILE statement. The name of the FILEVAR= variable, readit, is the name of the variable read from **Rawdatafiles.dat**. The value of readit is the name of the raw data file to be read.

**Raw Data File Rawdatafiles.dat**

| 1---+----10---+----20 |
|-----------------------|
| route1.dat            |
| route2.dat            |
| route3.dat            |
| route4.dat            |
| route5.dat            |

```
data work.newroute;
  infile 'rawdatafiles.dat';
  input readit $10.;

  infile in filevar=readit end=lastfile;
  do while(lastfile = 0);
    input @1 RouteID $7. @8 Origin $3. @11 Dest $3.
      @14 Distance 5. @19 Farelst 4.
      @23 FareBusiness 4. @27 FareEcon 4.
      @31 FareCargo 5.;

    output;
  end;
run;
```

## Chapter Summary

### Using a FILENAME Statement

You can use a FILENAME statement to concatenate raw data files by assigning a single fileref to the raw data files that you want to combine. When the fileref is specified in an INFILE statement, each raw data file that has been referenced can be sequentially read into a data set using an INPUT statement.

### Using an INFILE Statement

You can make the process of concatenating raw data files more flexible by using an INFILE statement with the FILEVAR= option. The FILEVAR= option enables you to dynamically change the currently opened input file to a new input file. When the INFILE statement executes, it reads from the file that the FILEVAR= variable specifies.

In some cases, you may need to use the COMPRESS function to eliminate spaces in the filenames you generate.

When you read the last record in a raw data file, the DATA step normally stops processing. When you are concatenating raw data files, you do not want to read past the last record until you reach the end of the last input file. You can determine if you are reading the last record in the last raw data file by using the END= option with the INFILE statement. You can then test the value of the END= variable to determine if the DATA step should continue processing.

If you are working with date-related data, you may be able to make your program more flexible by eliminating the need to include explicit month numbers in your SAS statements. To create a program that will always read the current month and the previous two months, you can use the MONTH and TODAY functions to obtain the month number of today's date to begin the quarter. In some cases, you may need to use the INTNX function with the TODAY and MONTH functions to correctly determine the month numbers.

### Appending SAS Data Sets

You can use PROC APPEND to concatenate two SAS data sets. PROC APPEND reads only the data in the DATA= SAS data set, not in the BASE= SAS data set. PROC APPEND concatenates data sets even though there may be variables in the BASE= data set that do not exist in the DATA= data set.

The FORCE option must be used if the DATA= data set contains variables that

- are not in the BASE= data set
- are longer than the variables in the BASE= data set
- do not have the same type as the variables in the BASE= data set.

The FORCE option can cause loss of data due to truncation or dropping of variables. The following table summarizes the consequences of using the FORCE option.

| DATA= data set contains variables that ...                          | FORCE required? | Consequences of using the FORCE option                                                   |
|---------------------------------------------------------------------|-----------------|------------------------------------------------------------------------------------------|
| are in the BASE=data set, but the BASE= data set has more variables | no              | Missing values are assigned to the extra BASE= data set variables.                       |
| are not in the BASE= data set                                       | yes             | Extra DATA= data set variables are dropped.                                              |
| are longer than the variables in the BASE= data set                 | yes             | DATA= data set variable values are truncated.                                            |
| do not have the same type as the variables in the BASE= data set    | yes             | Missing values are assigned to the DATA= data set variables with the data type mismatch. |

## Additional Features

You can also append raw data files using a SAS data set or an external file that contains the names of the raw data files to be appended.

### Syntax

#### Combining Raw Data Files Using a FILENAME Statement

```
FILENAME fileref ('external-file1' 'external-file2' 'external-filen');
DATA SAS-data-set;
    INFILE file-specification;
    INPUT variable <$> <&|:> <informat>;
RUN;
```

#### Combining Raw Data Files Using an INFILE Statement

```
DATA SAS-data=set;
DO index-variable=variable1, variable2, variablen;
    variable = "text-string"!!PUT(index-variable,format)!!"text-string";
    variable = COMPRESS (variable,'');
    DO UNTIL (variable);
        INFILE file-specification FILEVAR=variable END=variable;
        INPUT variable <$> <&|:> <informat>;
        OUTPUT;
    END;
    END;
    STOP;
RUN;
```

#### Combining SAS Data Sets Using PROC APPEND

```
PROC APPEND BASE=SAS-data-set DATA=SAS-data-set <FORCE>;
RUN;
```

### Sample Programs

#### Combining Raw Data Files Using a FILENAME Statement

```
filename qtr1 ('c:\data\month1.dat'"c:\data\month2.dat'
               'c:\data\month3.dat');
data work.firstqtr;
infile qtr1;
input Flight $ Origin $ Dest $ 
      Date : date9. RevCargo : comma15.2;
run;
```

#### Combining Raw Data Files Using an INFILE Statement

```
data quarter (drop=monthnum midmon lastmon);
monthnum=month(today());
midmon=month(intnx('month',today(),-1));
lastmon=month(intnx('month',today(),-2));
do i = monthnum, midmon, lastmon;
```

```

nextfile="c:\sasuser\month"
  !!compress(put(i,2.)!!".dat",' ');
do until (lastobs);
  infile temp filevar=nextfile end=lastobs;
  input Flight $ Origin $ Dest $ Date : date9.
    RevCargo : commal5.2;
  output;
end;
end;
stop;
run;

```

## Combining SAS Data Sets Using PROC APPEND

```

proc append base=work.acities
  data=work.airports force;
run;

```

### Points to Remember

- When you use an INFILE statement with the FILEVAR= option, the file specification is just a placeholder, not an actual filename or a fileref that has been previously assigned to a file.
- Like automatic variables, the FILEVAR= variable and the END= variable are not written to the data set.
- Using the FORCE option with PROC APPEND can cause loss of data due to truncation or dropping of variables.
- When you use the FORCE option, the structure of the BASE= data set is used for the appended data set.

## Chapter Quiz

Select the best answer for each question.

1. Which of the following statements associates the fileref **OnSale** with the raw data files **London.dat**, **Paris.dat**, and **Zurich.dat**? The files are stored in the **C:\Routes\New** directory in the Windows operating environment.
  - a. filename onsale (c:\routes\new\london.dat,c:\routes\new\paris.dat,  
c:\routes\new\zurich.dat);
  - b. filename onsale 'c:\routes\new\london.dat' 'c:\routes\new\paris.dat'  
'c:\routes\new\zurich.dat';
  - c. filename onsale ('c:\routes\new\london.dat' 'c:\routes\new\paris.dat'  
'c:\routes\new\zurich.dat');
  - d. filename onsale 'c:\routes\new\london.dat c:\routes\new\paris.dat  
c:\routes\new\zurich.dat';
2. Which of the following statements is **true**?
  - a. The FILEVAR= option can be used to dynamically change the currently opened input file to a new physical file.
  - b. The FILEVAR= variable is not written to the data set.
  - c. The FILEVAR= variable must contain a character string that is a physical filename.
  - d. all of the above
3. Given the following program, which table correctly shows the corresponding values of the variables x and readfile?

```
data work.revenue;
do x = 8, 9, 10;
readfile=compress("c:\data\month"
!!put(x,2)||".dat",'');
do until (lastobs);
infile temp filevar=nextfile
end=lastobs;
input Date : date7. Location $
Sales : dollar10.2;
output;
end;
end;
stop;
run;
```

a. **When x= readfile=**

| When x= | readfile=   |
|---------|-------------|
| 8       | month8.dat  |
| 9       | month9.dat  |
| 10      | month10.dat |

c. **When x= readfile=**

| When x= | readfile=           |
|---------|---------------------|
| 8       | c:\data\month 8.dat |
| 9       | c:\data\month 9.dat |
| 10      | c:\data\month10.dat |

b. **When x= readfile=**

| When x= | readfile=           |
|---------|---------------------|
| 8       | c:\data\month8.dat  |
| 9       | c:\data\month9.dat  |
| 10      | c:\data\month10.dat |

d. **When x= readfile=**

| When x= | readfile= |
|---------|-----------|
| 8       | month8    |
| 9       | month9    |
| 10      | month10   |

4. If the current date is March 30, 2003, which table correctly shows the corresponding values of the variables y1, y2, y3, and nextfile?

```

data work.quarter (drop=monthnum midmon lastmon);
y3=year(today());
y2=y3-1;
y1=y3-2;
do i = y3, y2, y1;
  nextfile="c:\data\Y'||put(i,4.)||.dat";
  do until (lastobs);
    infile temp filevar=nextfile
      end=lastobs;
    input Flight $ Origin $ Dest $ Date : date9. ;
    output;
  end;
end;
stop;
run;

```

| a. | When i= | nextfile=         |
|----|---------|-------------------|
| y1 |         | c:\data\Y2001.dat |
| y2 |         | c:\data\Y2002.dat |
| y3 |         | c:\data\Y2003.dat |

| c. | When i= | nextfile=         |
|----|---------|-------------------|
| y1 |         | c:\data\Y2003.dat |
| y2 |         | c:\data\Y2002.dat |
| y3 |         | c:\data\Y2001.dat |

| b. | When i= | nextfile= |
|----|---------|-----------|
| y1 |         | Y2001.dat |
| y2 |         | Y2002.dat |
| y3 |         | Y2003.dat |

| d. | When i= | nextfile=      |
|----|---------|----------------|
| y1 |         | c:\data\Y3.dat |
| y2 |         | c:\data\Y2.dat |
| y3 |         | c:\data\Y1.dat |

5. Which of the following statements is **false**?

- a. The END= variable is set to 0 when SAS processes the last data record in the input file.
- b. The END= variable is set to 1 when SAS processes the last data record in the input file.
- c. The END= variable is not written to the data set.
- d. a and c

6. Which program appends **Work.London** to **Work.Flights**?

| Data Set Description for Work.London |      |        | Data Set Description for Work.Flights |      |        |
|--------------------------------------|------|--------|---------------------------------------|------|--------|
| Variable                             | Type | Length | Variable                              | Type | Length |
| FlightNum                            | char | 4      | FlightNum                             | char | 4      |
| Destination                          | char | 3      | Destination                           | char | 3      |
| Departure                            | num  | 8      | Departure                             | num  | 8      |
| Gate                                 | char | 2      | Gate                                  | char | 2      |

- a. proc append base=work.london data=work.flights;  
run;
- b. proc append data=work.london base=work.flights;  
run;

- c. proc append data=work.london work.flights;  
run;
- d. proc append data=work.flights work.london;  
run;
7. What happens when the following program is submitted?
- ```
proc append base=staff.marketing
           data=staff.sales force;
run;
```
- | Data Set Description for Staff.Marketing |      |        | Data Set Description for Staff.Sales |      |        |
|--|------|--------|--------------------------------------|------|--------|
| Variable                                 | Type | Length | Variable                             | Type | Length |
| LastName                                 | char | 12     | LastName                             | char | 20     |
| FirstName                                | char | 10     | FirstName                            | char | 10     |
| EmpID                                    | char | 5      | EmpID                                | char | 5      |
| Office                                   | char | 4      | Office                               | char | 4      |
| Phone                                    | char | 12     | Phone                                | char | 12     |
- a. The length of LastName is converted to 20 in **Staff.Marketing**.
  - b. LastName is dropped from **Staff.Marketing**.
  - c. Missing values are assigned to LastName in observations that are read in from **Staff.Sales**.
  - d. Some of the values of LastName may be truncated in the observations that are read in from **Staff.Sales**.
8. Which program appends **Work.April** to **Work.Y2003**?

Data Set Description for Work.Y2003			Data Set Description for Work.April		
Variable	Type	Length	Variable	Type	Length
FlightNum	num	8	FlightNum	char	4
FirstClass	num	8	FirstClass	num	8
BusinessClass	num	8	BusinessClass	num	8
Coach	num	8	Coach	num	8

- a. proc append base=work.y2003 data=work.april;  
run;
- b. proc append base=work.april data=work.y2003 force;  
run;
- c. proc append base=work.y2003 data=work.april force;  
run;
- d. proc append base=work.april data=work.y2003;  
run;

9. What happens when the SAS data set **Work.NewHires** is appended to the SAS data set **Work.Employees** using PROC APPEND?

Data Set Description for Work.Employees			Data Set Description for Work.NewHires		
Variable	Type	Length	Variable	Type	Length
Division	num	8	Division	num	8
EmplID	num	8	EmplID	num	8
Name	char	20	Name	char	20
Room	char	5	Extension	num	8
Extension	num	8			

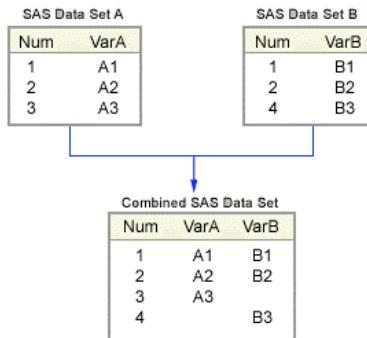
- a. Missing values are assigned to Room for the observations that are read in from **Work.NewHires**.
  - b. Missing values are assigned to Room for all of the observations in **Work.Employees**.
  - c. Room is dropped from **Work.Employees**.
  - d. The values of Name are truncated in the observations that are read in from **Work.NewHires**.
10. You do **not** need to use the FORCE option with PROC APPEND when
- a. the DATA= data set contains variables that are not in the BASE= data set.
  - b. the BASE= data set contains variables that are not in the DATA= data set.
  - c. the variables in the DATA= data set are longer than the corresponding variables in the BASE= data set.
  - d. the variables in the DATA= data set have a different type than the corresponding variables in the BASE= data set.

# Chapter 3

## Combining Data Horizontally

### Introduction

Combining data horizontally refers to the process of merging or joining multiple data sets into one data set. This process is referred to as a horizontal combination because in the final data set, each observation (or horizontal row) will have variables from more than one input data set.



It is useful to combine data horizontally if you have several data sets that contain different but related information. For example, suppose you have one data set that contains employee data with the variables IDNumber, Name, and Address, and another data set that contains employee data with the variables IDNumber and Salary. You can combine the data from these two input data sets horizontally to create an output data set that contains IDNumber, Name, Address, and Salary.

There are several methods for combining data horizontally. This lesson focuses on several methods of combining data horizontally in the DATA step, and compares a DATA step match-merge with a PROC SQL join. This lesson also covers several techniques for horizontally combining data from an input data set with values that are not stored in a SAS data set.

### Objectives

In this lesson, you learn to

- identify factors that affect which technique is most appropriate for combining data horizontally
- use the IF-THEN/ELSE statement, SAS arrays, or user-defined SAS formats to combine data horizontally
- use the DATA step with the MERGE statement to combine data sets that don't have a common variable
- use the SQL procedure to combine data sets that don't have a common variable
- identify the differences between the DATA step match-merge and the PROC SQL join
- create an output data set that contains summary statistics from PROC MEANS
- combine summary statistics in a data set with a detail data set
- calculate summary data and combine it with detail data within one DATA step
- use the SET statement with the KEY= option to combine two SAS data sets
- use an index to combine two data sets
- use \_IORC\_ to determine whether an index search was successful
- use the UPDATE statement to update a master data set with a transactional data set.

## Reviewing Terminology

Before examining the various techniques for combining data horizontally, let's review some of the terminology that this lesson uses. The table below lists important terms that you will need to know, along with their definitions.

Term	Definition
<b>combining data horizontally</b>	A technique in which information is retrieved from an auxiliary source or sources, based on the values of variables in the primary source.
<b>performing a table lookup</b>	A technique in which information is retrieved from an auxiliary source or sources, based on the values of variables in the primary source.
<b>base table</b>	The primary source in a horizontal combination. In this lesson, the base table is always a SAS data set.
<b>lookup table(s)</b>	All input data sources, except the base table, that are used in a horizontal combination.
<b>lookup values</b>	The data values that are retrieved from the lookup table(s) during a horizontal combination.
<b>key variable(s)</b>	One or more variables that reside in both the primary file and the lookup file. The values of the key variable(s) are the common elements between the files. Often, key values are unique in the base file but are not necessarily unique in the lookup file(s).
<b>key value(s)</b>	For each observation, the value(s) for the key variable(s).

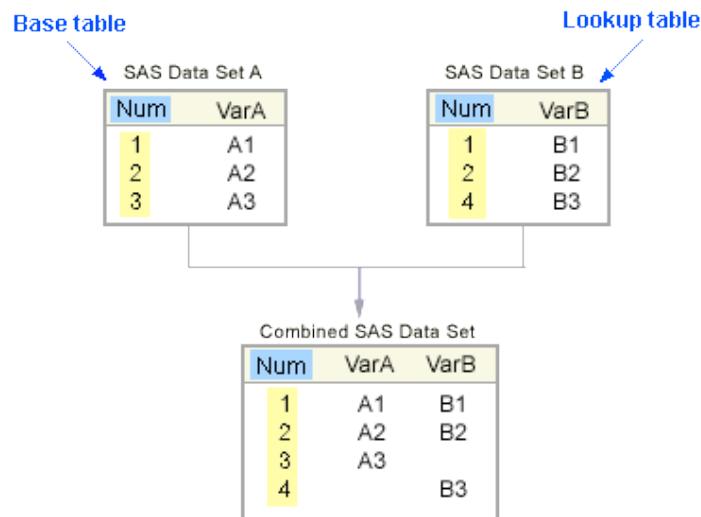


The terms **combining data horizontally** and **performing a table lookup** are synonymous and are used interchangeably throughout this lesson.



This lesson compares PROC SQL techniques with DATA step techniques. In PROC SQL terms, a **SAS data set** is usually referred to as a **table**, a **variable** is usually referred to as a **column**, and an **observation** is usually referred to as a **row**.

The following figure illustrates a base table and a lookup table that are used in an horizontal combination. The key variable, Num, is highlighted in blue in each table, and key values are highlighted in yellow.



## Relationships between Input Data Sources

One important factor to consider when you perform a table lookup is the relationship between the input data sources. In order to combine data horizontally, you must be able to match observations from each input data source. For example, there might be one or more variables that are common to each input data source. The relationship between input data sources describes how the observations in one source relate to the observations in the other source according to these key values.

The following terms describe the possible relationships between base tables and lookup tables.

- One-to-one match
- One-to-many match
- Many-to-many match
- Nonmatching data

Let's look at each of these relationships in more detail.

In a **one-to-one match**, key values in both the base table and the lookup table are unique. Therefore, for each observation in the base table, no more than one observation in the lookup table has a matching key value.

Base Table		Lookup Table	
Num	VarA	Num	VarB
1	A1	1	B1
2	A2	2	B2
3	A3	3	B3
4	A4	4	B4

In a **one-to-many match**, key values in the base table are unique, but key values in the lookup table are not unique. That is, for each observation in the base table there can be one observation or possibly multiple observations in the lookup table that have a matching key value.

Base Table		Lookup Table	
Num	VarA	Num	VarB
1	A1	1	B1
2	A2	1	B2
		1	B3
		2	B4

In a **many-to-many match**, key values are not unique in the base table or in the lookup table. That is, at least one observation in the base table matches multiple observations in the lookup table, and at least one observation in the lookup table matches multiple observations in the base table.

Base Table		Lookup Table	
Num	VarA	Num	VarB
1	A1	1	B1
1	A2	1	B2
3	A3	1	B3
4	A4	3	B4
4	A1	4	B1

Sometimes you will have a one-to-one, a one-to-many, or a many-to-many match that also includes **nonmatching data**. That is, there are observations in the base table that do not match any observations in the lookup table, or there are observations in the lookup table that do not have matching observations in the base table. If your base table or lookup table(s) include nonmatching data, you will have one of the following:

**a dense match**, in which nearly every observation has a matching observation in the corresponding table. In the following figure, the highlighted observation is unmatched.

Base Table		Lookup Table	
Num	VarA	Num	VarB
1	A1	2	B1
2	A2	3	B2
3	A3	4	B3
4	A4		



A dense match can also refer to a relationship in which every observation has a matching observation in the corresponding table and there is no nonmatching data.

**a sparse match**, in which there are more unmatched observations than matched observations in either the base table or the lookup table. In the following figure, the highlighted observations are unmatched.

Base Table		Lookup Table	
Num	VarA	Num	VarB
1	A1	2	B1
2	A2	4	B2
3	A3	5	B3
6	A5		

Examine the two data sets shown below.

Q1.

SAS Data Set A		SAS Data Set B	
Num	VarOne	Num	VarTwo
2458	07OCT2003	2458	itemA
2462	19NOV2001	2462	itemB
2501	30MAR2001	2462	itemC
		2501	itemA

Which term best describes the relationship between these data sets?

- A one-to-one match
- B one-to-many match
- C nonmatching data
- D many-to-many match

## Working with Lookup Values Outside of SAS Data Sets

Remember that it is not necessary for your lookup table to be a SAS data set. Suppose you want to combine the data from your base table with lookup values that are not stored in a SAS data set. You can use the following techniques to hard-code lookup values into your program:

- the IF-THEN/ELSE statement
- SAS arrays
- user-defined SAS formats.
- The IF-THEN/ELSE Statement

You should be familiar with the syntax and use of the IF-THEN/ELSE statement. Overall, this technique is easy to use and easy to understand. Because of its simplicity and because you can use other DATA step syntax with it, the IF-THEN/ELSE statement can be quite versatile as a technique for performing lookup operations. You can use this technique if your lookup values are not stored in a data set, and you can use it to handle any of the possible relationships between your base table and your lookup table if your lookup values are stored in a data set. You can use it to retrieve single or multiple values. For example, you can use DO groups to provide multiple values based on a condition.

Keep in mind that this technique will require maintenance. If you expect your lookup values to change and you have a large number of lookup values, or if you use the lookup values in multiple programs, the resources required for maintaining the IF-THEN/ELSE statements in your programs might make this technique inappropriate. Also, this technique might result in a prohibitively long program or even in a program that will not execute because it times out.

### Example: Using the IF-THEN/ELSE Statement to Combine Data

Suppose you have a data set, **Mylib.Employees**, that contains information about employees. **Mylib.Employees** contains a variable named IDnum that records each employee's unique identification number. If you want to combine the data from **Mylib.Employees** with a list of employees' birthdates that is not stored in a data set, you can use the IF-THEN/ELSE statement to do so.

```
data mylib.employees_new;
  set mylib.employees;
  if IDnum=1001 then Birthdate='01JAN1963'd;
  else if IDnum=1002 then Birthdate='08AUG1946'd;
  else if IDnum=1003 then Birthdate='23MAR1950'd;
  else if IDnum=1004 then Birthdate='17JUN1973'd;
run;
```



Note that **Mylib.Employees** is a fictitious data set that is used for this example and for the examples on the next two pages. The **Mylib** library and the **Employees** data set are not included in the practice data for this lesson; therefore, these sample programs will not run.

## SAS Arrays

You should be familiar with the syntax and use of the ARRAY statement. With the ARRAY statement, you can either hard-code your lookup values into the program, or you can read them into the array from a data set. Elements of a SAS array are referenced positionally. That is, you use a numeric value as a pointer to the array element, so you must be able to identify elements of the array either by position or according to another numeric value. You can use multiple values or numeric mathematical expressions to determine the array element to be returned. Exact matches are not required with this technique.

The memory requirements for loading the entire array can be a drawback to using the ARRAY statement to perform a table lookup. Also, this technique is capable of returning only a single value from the lookup operation. Finally, the dimensions of the array must be supplied at compile time either by hard-coding or through the use of macro variables.

### Example: Using the ARRAY Statement to Combine Data

Let's take another look at our example of combining the data from **Mylib.Employees** with a list of lookup values. Remember that **Mylib.Employees** contains data about employees, which includes their identification numbers (IDnum) but does not include their birthdates. You can use the ARRAY statement to hard-code the birthdates into a temporary array named birthdates, and then use the array to combine the birthdates with the data in **Mylib.Employees**.

In the following DATA step, the values that are specified as subscripts for the array correspond to values of the variable IDnum in the base table, **Mylib.Employees**. The assignment statement for the new variable Birthdate retrieves a value from the birthdates array according to the current value of IDnum.

```
data mylib.employees_new;
array birthdates[1001:1004] _temporary_ ('01JAN1963'd
  '08AUG1946'd '23MAR1950'd '17JUN1973'd);
set mylib.employees;
Birthdate=birthdates(IDnum);
run;
```



For more information about using SAS arrays, see the lesson [Processing Variables with Arrays](#).

### User-Defined SAS Formats

You should be familiar with the syntax and use of the FORMAT procedure with the VALUE statement. You can associate a format with a variable permanently by using a FORMAT or ATTRIB statement in a DATA step or PROC step that creates a SAS data set. In a DATA step, you can use a PUT statement in an assignment statement in order to use the format only while the PUT function executes. In a DATA step or PROC step, you can use the PUT function in a WHERE statement in order to use the format only during execution of the PUT function.

One advantage of using formats to combine data is that you do not have to create a new SAS data set in order to perform the lookup. Formats can be used to collapse data into categories as well as to expand data, and they can change the appearance of a report without the creation of a new variable. You can create multiple formats and use all of them in the same DATA or PROC step.

The FORMAT procedure uses a binary search (a rapid search technique) through the lookup table. Another benefit of using this technique is that maintenance is centralized; if a lookup value changes, you only have to change it in one place (in the format), and every program that uses the format will use the new value.

On the other hand, the FORMAT procedure requires the entire format to be loaded into memory for the binary search, so this technique might use more memory than others if there are a large number of lookup values.

## Example: Using the FORMAT Procedure to Combine Data

Once again, suppose the data set **Mylib.Employees** contains information about employees according to their employee identification numbers (IDnum), but does not contain employees' birthdates. You can use a format to combine employees' birthdates with the data that is stored in **Mylib.Employees**.

The following PROC FORMAT step uses a VALUE statement to hard-code the lookup values in the BIRTHDATE format. Then the DATA step uses the PUT function to associate the lookup values from the format with the values of IDnum, uses the INPUT function to associate the lookup value with the DATE9. informat, and assigns the formatted values to a new variable named Birthdate.

```
proc format;
  value birthdate 1001 = '01JAN1963'
    1002 = '08AUG1946'
    1003 = '23MAR1950'
    1004 = '17JUN1973';
run;
data mylib.employees_new;
  set mylib.employees;
  Birthdate=input(put(IDnum,birthdate.),date9.);
run;
```



For more information about user-defined SAS formats, see the lesson [Creating and Applying User-Defined Formats](#).

Select the statement below that is true.

- Q.2.**
- A If the lookup values are not stored in a SAS data set, you must use the IF-THEN/ELSE statement to perform a table lookup operation.
  - B You can use either the IF-THEN/ELSE statement or the ARRAY statement to return multiple lookup values.
  - C A user-defined format can provide easier maintenance than the IF-THEN/ELSE statement for lookup values that change often or that are used in many programs.

## Combining Data with the DATA Step Match-Merge

### The DATA Step Match-Merge

You should already know how to merge multiple data sets in the DATA step when there is a **BY variable** that is common to each of the input data sets. When you use the MERGE statement to perform a table lookup operation, your lookup values must be stored in one or more SAS data sets. Also, this technique requires that both the base table and the lookup table(s) be either sorted by or indexed on the BY variable(s).

You can specify any number of input data sets in the MERGE statement as long as all input data sets have a common BY variable. Also, the MERGE statement can combine data sets of any size. The MERGE statement is capable of returning multiple values, and you can use multiple BY variables to perform lookups that are dependent on more than one variable. The MERGE statement returns both matches and non-matches by default, but you can use DATA step syntax to return only exact matches or to include only specific values from the lookup table.



Although you can use the MERGE statement to combine data from sources that have any type of relationship, this technique might **not** produce the desired results when you are working with a many-to-many match. When the data sets are merged in a DATA step, the observations are matched and combined sequentially. Once an observation is read, it is never re-read. That is, the DATA step MERGE statement does not create a Cartesian product. Therefore, the DATA step MERGE statement is probably not an appropriate technique to use for performing lookup operations when you are working with a many-to-many match.

## Working with Multiple Lookup Tables

Sometimes you might need to combine data from three or more related SAS data sets in order to create one new data set. For example, the three data sets listed below all contain different information that relates to a fictional airline's flights and airports. **Sasuser.Acities** contains data about various airports, **Sasuser.Revenue** contains data about the revenue generated by various flights, and **Sasuser.Expenses** contains data about the expenses incurred by various flights. The variables in each of these data sets are listed here.

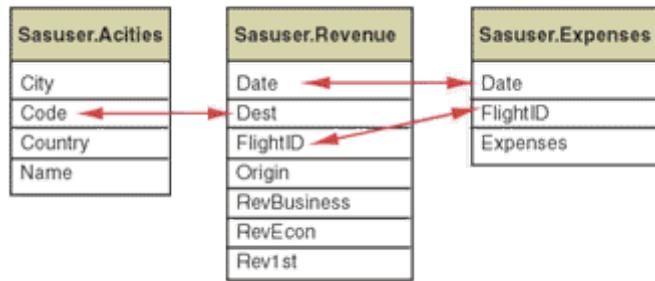
Sasuser.Acities Variables	Sasuser.Revenue Variables	Sasuser.Expenses Variables
City Code Country Name	Date Dest FlightID Origin RevBusiness RevEcon Rev1st	Date FlightID Expenses

Suppose you want to create a new data set, named **Sasuser.AllData**, that contains data from each of these three input data sets. As shown below, the **Sasuser.AllData** data set contains the variable Profit, which is calculated from the revenue values that are stored in **Sasuser.Revenue** and the expense values that are stored in **Sasuser.Expenses**.

Sasuser.AllData Variables
Date
Dest
FlightID
Origin
Profit
DestAirport
DestCity

You know that you can specify any number of input data sets in the MERGE statement as long as all input data sets have a common BY variable. However, you can see from the data set variable lists above that these three data sets do not have one common variable. Let's look at a method for performing a match-merge on these three data sets.

In order to use the MERGE statement in the DATA step to combine data sets, all input data sets must have a common variable. Although the three data sets **Sasuser.Acities**, **Sasuser.Revenue**, and **Sasuser.Expenses** do not have a common BY variable, there are several variables that are common to **two of the three** data sets. As shown below, Date and FlightID are both common to **Revenue** and **Expenses**. The variable Code in the **Acities** data set and the variable Dest in the **Revenue** data set are also common variables.



Notice that Code in **Acities** and Dest in **Revenue** are listed as corresponding to one another even though they have different names. When you are looking for common variables between data sets, the variable names are not important since they can be changed with the RENAME= option in the MERGE statement. Instead, you should look for variables that record the same information and that have the same type in each input data set. Common variables do not need to have the same length, although you should remember that the length of the variable in the first-listed data set will determine the length of the variable in the output data set.

- Any variables that have the same name in multiple data sets in the MERGE statement must also have the same type. If any variables in different input data sets have identical names but do not have identical types, ERROR and WARNING messages are written to the SAS log, and the match-merge fails.

In this case, both Code in **Acities** and Dest in **Revenue** record the three-letter abbreviation of an airport.

- You can use PROC CONTENTS to view information about variables such as type, length, and description.

Since there are variables that are common to two different pairs of the three data sets shown above, you can combine these data sets into one data set by using the MERGE statement in two subsequent DATA steps. That is, you can perform one match-merge on two of the data sets to create one new data set that combines information from the two. Then you can perform another match-merge on the new data set and the remaining original data set. Let's take a closer look.

### Example

In the following program, both **Sasuser.Expenses** and **Sasuser.Revenue** are sorted by FlightID and Date and are placed into temporary data sets in order to prepare them for being merged. Then these two sorted data sets are merged in a DATA step that creates a temporary output data set named **Revexpns**. In order to reduce the total number of variables in the output data set, a new variable named Profit is created, and the variables that are used to create Profit are dropped from **Revexpns**.

```

proc sort data=sasuser.expenses out=expenses;
  by flightid date;
run;
proc sort data=sasuser.revenue out=revenue;
  by flightid date;
run;
data revexpns (drop=revlst revbusiness revecon expenses);
  merge expenses(in=e) revenue(in=r);
  by flightid date;
  if e and r;
  Profit=sum(revlst, revbusiness, revecon, -expenses);
run;

```



The use of the temporary IN= variables E and R in the IF statement above ensures that only observations that contain data from each of the two input data sets are included in the output data set.

In the following program, the output data set named **Revexpns** is sorted by Dest. **Sasuser.Acities** is sorted by Code and is placed in a temporary data set. Remember that Dest and Code are corresponding variables even though they have different names.

The sorted data sets are then merged in a DATA step. Since two data sets must have at least one variable that matches exactly in order to be merged, the RENAME= option renames Code to Dest in the output data set. The DATA step merges **Revexpns** and **Acities** into a new output data set named **Alldata**.

```
proc sort data=revexpns;
  by dest;
run;
proc sort data=sasuser.acities out=acities;
  by code;
run;

data sasuser.alldata;
  merge revexpns(in=r) acities
    (in=a rename=(code=dest) keep=city name code);
  by dest;
  if r and a;
run;
proc print data=sasuser.alldata(obs=5) noobs;
  title 'Result of Merging Three Data Sets';
  format Date date9.;
run;
```

The PROC PRINT step prints the first five observations in the **Sasuser.Alldata** data set that is created in this example, as shown here.

Result of Merging Three Data Sets

FlightID	Date	Origin	Dest	Profit	City	Name
IA03300	06DEC1999	RDU	ANC	34010	Anchorage, AK	Anchorage International Airport
IA03300	18DEC1999	RDU	ANC	73471	Anchorage, AK	Anchorage International Airport
IA03300	30DEC1999	RDU	ANC	77755	Anchorage, AK	Anchorage International Airport
IA03301	13DEC1999	RDU	ANC	110402	Anchorage, AK	Anchorage International Airport
IA03301	25DEC1999	RDU	ANC	111151	Anchorage, AK	Anchorage International Airport

**Q.3.**

Complete the following statement to combine the **Sasuser.AllEmps**, **Sasuser.Contrib**, and **Sasuser.Newsals** data sets that are shown here. Assume that each of these data sets has been sorted by EmpID. Also assume that EmpID records employees' 4-digit numeric ID numbers in each of these data sets.

<b>Sasuser.AllEmps Variables</b>
EmpID
LastName
Phone
Location
Division

<b>Sasuser.Contrib Variables</b>
EmpID
QtrNum
Amount

<b>Sasuser.Newsals Variables</b>
EmpID
Salary
NewSalary

data combined;

-----  
by empid;  
run;

**Q.4.**

Complete the following statement to combine the **Sasuser.AllEmps**, **Sasuser.Contrib**, and **Sasuser.Newsals** data sets that are shown here. Assume that Newsals.EmpCode records the employee ID number in exactly the same type and length as AllEmps.EmpID and Contrib.EmpID. Also, assume that **AllEmps** and **Contrib** have been sorted on EmpID and that **Newsals** has been sorted on EmpCode.

<b>Sasuser.AllEmps Variables</b>
EmpID
LastName
Phone
Location
Division

<b>Sasuser.Contrib Variables</b>
EmpID
QtrNum
Amount

<b>Sasuser.Newsals Variables</b>
EmpCode
Salary
NewSalary

data joined;

-----  
by empid;  
run;

## Using PROC SQL to Join Data

### The SQL Procedure

Another method that you can use to join data sets that do not have a common variable is the SQL procedure. You should already be familiar with using PROC SQL to create a table from the results of an inner join.

In a PROC SQL step, you can choose from each input data set only the specific variables that you want to include in the new data set, and you can return multiple values. The input data sets do not need to contain a common BY variable, nor do they need to be sorted or indexed. However, if the lookup table does have an index, the SQL procedure can take advantage of the index to provide faster retrieval of lookup values.

You can join up to 32 tables with the SQL procedure, and you can use this technique to combine data horizontally from sources that have any type of relationship (one-to-one, one-to-many, many-to-many, or non-matching). Exact matches are returned by default from an inner join.



Although numerous kinds of joins are possible with PROC SQL, only **inner joins** are discussed in this lesson. Therefore, in the remainder of this lesson, "a PROC SQL join" refers to an inner join on multiple tables, with the results stored in a new table. You can learn more about PROC SQL joins in the lesson [Combining Tables Horizontally Using PROC SQL](#).

One drawback to using the SQL procedure to perform table lookups is that you cannot use DATA step syntax with PROC SQL, so complex business logic is difficult to incorporate into the join. However, by using PROC SQL you can often do in one step what it takes multiple PROC SORT and DATA steps to accomplish.

### Example: Working with Multiple Lookup Tables

The following example joins **Sasuser.Revenue**, **Sasuser.Expenses**, and **Sasuser.Acities** into a new data set named **Work.Sqljoin**:

```
proc sql;
  create table sqljoin as
    select revenue.flightid, revenue.date format=date9.,
           revenue.origin, revenue.dest,
           sum(revenue.revlst,
                revenue.revbusiness,
                revenue.revecon)
      -expenses.expenses as Profit,
       acities.city,
       acities.name
   from sasuser.expenses, sasuser.revenue,
        sasuser.acities
  where expenses.flightid=revenue.flightid
    and expenses.date=revenue.date
    and acities.code=revenue.dest
  order by revenue.dest,
           revenue.flightid,
           revenue.date;
quit;

proc print data=work.sqljoin(obs=5);
  title 'Result of Joining Three Data Sets';
run;
```

The PROC PRINT step produces the first five observations of the **Work.Sqljoin** data set that is created in the PROC SQL step above, as shown here:

**Result of Joining Three Data Sets**

Obs	FlightID	Date	Origin	Dest	Profit	City	Name
1	IA03300	06DEC1999	RDU	ANC	34010	Anchorage, AK	Anchorage International Airport
2	IA03300	18DEC1999	RDU	ANC	73471	Anchorage, AK	Anchorage International Airport
3	IA03300	30DEC1999	RDU	ANC	77755	Anchorage, AK	Anchorage International Airport
4	IA03301	13DEC1999	RDU	ANC	110402	Anchorage, AK	Anchorage International Airport
5	IA03301	25DEC1999	RDU	ANC	111151	Anchorage, AK	Anchorage International Airport



Notice that the **Work.Sqljoin** data set is identical to the **Sasuser.Alldata** data set that was previously created in the DATA step merge.

## Comparing DATA Step Match-Merges and PROC SQL Joins

You have seen that it is possible to create identical results with a DATA step match-merge and a PROC SQL inner join. Although the results might be identical, these two processes are very different, and trade-offs are associated with choosing one method over the other. The following tables summarize some of the advantages and disadvantages of each of these two methods.

**DATA Step Match-Merge**

Advantages	Disadvantages
<ul style="list-style-type: none"><li>There is no limit to the number of input data sets, other than memory.</li><li>Allows for complex business logic to be incorporated into the new data set by using DATA step processing, such as arrays and DO loops, in addition to MERGE features.</li><li>Multiple BY variables enable lookups that depend on more than one variable.</li></ul>	<ul style="list-style-type: none"><li>Data sets must be sorted by or indexed on the BY variable(s) prior to merging.</li><li>The BY variable(s) must be present in all data sets, and the names of the key variable(s) must match exactly.</li><li>An exact match on the key value(s) must be found.</li></ul>

**PROC SQL Join**

Advantages	Disadvantages
<ul style="list-style-type: none"><li>Data sets do not have to be sorted or indexed, but an index can be used to improve performance.</li><li>Multiple data sets can be joined in one step without having common variables in all data sets.</li><li>You can create data sets (tables), views, or query reports with the combined data.</li></ul>	<ul style="list-style-type: none"><li>The maximum number of tables that can be joined at one time is 32.</li><li>Complex business logic is difficult to incorporate into the join.</li><li>PROC SQL might require more resources than the DATA step with the MERGE statement for simple joins.</li></ul>

Although it is possible to produce identical results with a DATA step match-merge and a PROC SQL join, these two processes will not always produce results that are identical by default. These two techniques work differently. In order to decide which technique you should use in a particular situation, you should carefully consider both the data that you want to combine and the results that you want to produce.

Let's take a look at some simplified examples to see how each method works in various circumstances.

The following two steps show two different ways to produce the same combination of two data sets, **Data1** and **Data2**, that have a common variable, X. If **Data1** contains two variables, X and Y, and **Data2** contains two variables, X and Z, then both of the following steps produce an output data set named **Data3** that contains three variables, X, Y, and Z.



The code shown in the following two steps illustrates a simple comparison of a DATA step match-merge and a PROC SQL join. This comparison will be explored over the next few pages. However, since the **Data1** and **Data2** data sets do not exist, this code will not run successfully in your SAS session.

```

data data3;
  merge data1 data2;
  by x;
run;

proc sql;
  create table data3 as
    select data1.x, data1.y, data2.z
    from data1, data2
    where data1.x=data2.x;
quit;

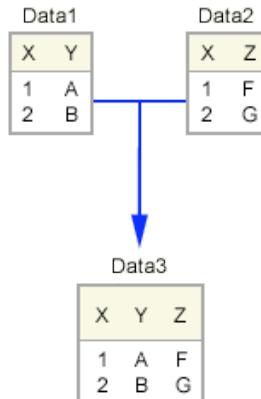
```

The contents of **Data3** will vary depending on the values that are in each input data set and on the method used for merging. Let's take a closer look at some examples.

## Examples

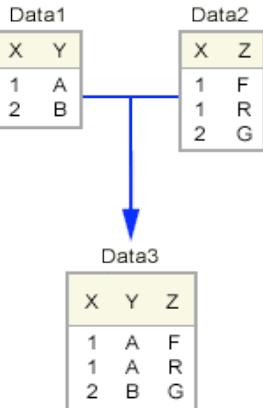
**One-to-one matches** produce **identical results** whether the data sets are merged in a DATA step or joined in a PROC SQL step. Suppose that **Data1** and **Data2** contain the same number of observations. Also, suppose that in each data set, the values of X are unique, and that each value appears in both data sets.

When these data sets are either merged in a DATA step or joined in a PROC SQL step, **Data3** will contain one observation for each unique value of X, and it will have the same number of observations as **Data1** and **Data2**.



**One-to-many matches** produce **identical results** whether the data sets are merged in a DATA step or joined in a PROC SQL step. Suppose that **Data1** contains unique values for X, but that **Data2** does not contain unique values for X. That is, **Data2** contains multiple observations that have the same value of X and therefore contains more observations than **Data1**.

When these two data sets are either merged in a DATA step or joined in a PROC SQL step, **Data3** will contain the same number of observations as **Data2**. In **Data3**, one observation from **Data1** that has a particular value for X might be matched with multiple observations from **Data2** that have the same value for X.

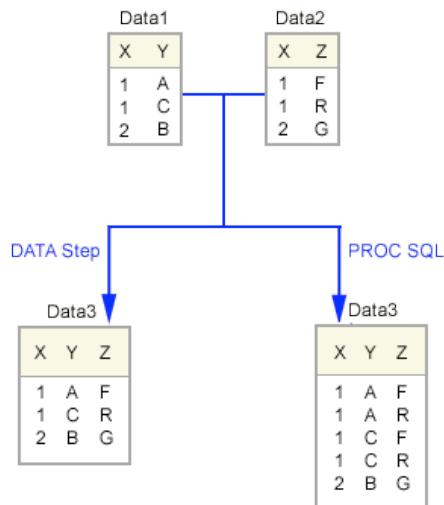


**Many-to-many matches** produce **different results** depending on whether the data sets are merged in a DATA step or joined in a PROC SQL step. Suppose the values of X are not unique in either **Data1** or in **Data2**.

When the data sets are merged in a DATA step, the observations are matched and combined sequentially. That is, an observation from the first input data set will be combined with the first observation from the second input data set that has a matching value for the BY variable. Although there might be additional observations in the second input data set that match the observation from the first input data set, these will not be included in the output data set.

In the example below, **Data3** will contain the same number of observations as the larger of the two input sets. In cases where there is a many-to-many match on the values of the BY variable, a DATA step match-merge probably does not produce the desired output because the output data set will not contain all of the possible combinations of matching observations.

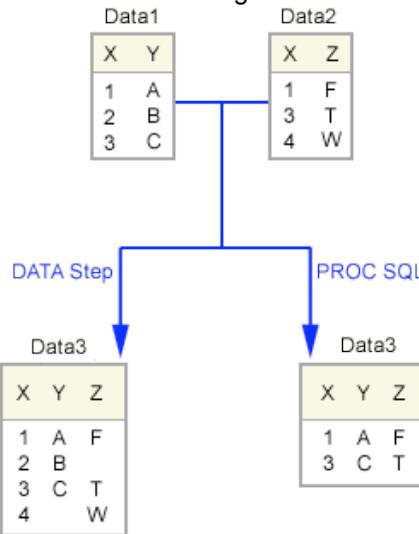
When the data sets are joined in a PROC SQL step, each match appears as a separate observation in the output data set. In the example below, the first observation that has a value of 1 for X in **Data1** is matched and combined with each observation from **Data2** that has a value of 1 for X. Then, the second observation that has a value of 1 for X in **Data1** is matched and combined with each observation from **Data2** that has a value of 1 for X, and so on.



**Nonmatching data** between the data sets produces **different results** depending on whether the data sets are merged in a DATA step or combined by using a PROC SQL inner join.

When data sets that contain nonmatching values for the BY variable are merged in a DATA step, the observations in each are processed sequentially. **Data3** will contain one observation for each unique value of X that appears in either **Data1** or **Data2**. For those values that have a nonmatching value for X, the observation in **Data3** will have a missing value for the variable that is taken from the other input data set.

When data sets that contain nonmatching values for the BY variable are joined in a PROC SQL step, the output data set will contain only those observations that have matching values for the BY variable. In the example below, **Data3** does not have any observations with missing values, because any observation from **Data1** or from **Data2** that contains a nonmatching value for X is not included in **Data3**.



You have seen the results of DATA step match-merges and PROC SQL joins in several simple scenarios. To help you understand the differences more fully, let's take a closer look at how the DATA step processes a match-merge and at how PROC SQL processes a join.

## DATA Step Match-Merge

When you merge data sets in a DATA step, the observations in each input data set are read sequentially and are matched and combined in the output data set. The animation below depicts a DATA step match-merge of two simple input data sets.

**Execution of a DATA Step Match-Merge.** The following steps shows the execution of the DATA step below.

1. This DATA step creates a new data set by performing a basic match-merge on two input data sets.
 

```
data work.data3;
merge data1 data2;
  by x;
run;
```
2. During the compilation phase, SAS reads the descriptor portions of the input data sets and creates the PDV. Also, SAS determines the BY groups in each input data set for the variables listed in the BY statement.
3. Execution begins. SAS looks at the first BY group in each input data set to determine if the BY values match. If so, SAS reads the first observation of that BY group from the first input data set and records the values in the PDV.

4. Since the BY values match, SAS reads the first observation of the same BY group from the second input data set and records the remaining values in the PDV.
5. SAS writes the merged data to the output data set.
6. SAS continues to merge observations in the same manner until it has written all observations from the first BY group to the new data set. In this example, there are two observations in the new data set that result from the first BY group ( $X = 11$ ).
7. If the BY values do not match, SAS reads the input data set with the lowest BY value. The PDV and the output data set will contain missing values for variables that are unique to the other data set.
8. If an input data set does not contain any observations in a particular BY group, the PDV and the output data set will contain missing values for the variables that are unique to that input data set.
9. SAS continues to match-merge observations until all observations from both input data sets have been read and written to the new data set. In this example, **Work.Data3** contains three variables and four observations.

## PROC SQL Join

A PROC SQL join uses a different process than a DATA step merge to combine data sets.

Conceptually, PROC SQL first creates a **Cartesian product** of all input sets. That is, PROC SQL first matches **each observation with every other observation** in the other input data sets. Then, PROC SQL eliminates any observations from the result set that do not satisfy the WHERE clause of the join statement. The PROC SQL query optimizer contains methods to minimize the Cartesian product that must be built.

**Execution of a PROC SQL Join.** The following steps shows the execution of the PROC SQL step below.

1. This PROC SQL step creates a new data set to hold the results of an inner join on two input data sets. This animation provides a conceptual view of how PROC SQL works rather than a literal depiction of the join process. In reality, SAS employs optimization routines that make the process more efficient.

```
proc sql;
  create table work.data4 as
    select *
    from data1, data 2
    where data1.x=data2.x;
quit;
```

2. Conceptually, PROC SQL first creates a Cartesian product of the two input data sets, where each observation from the first data set is combined with each observation from the second data set. PROC SQL starts by taking the first observation from **Work.Data1** and combining it with the first observation of **Work.Data2**.
3. Next, PROC SQL takes the first observation from **Work.Data1** and combines it with the second observation from **Work.Data2**.
4. PROC SQL continues in this manner until it has combined each observation from **Work.Data1** with every observation from **Work.Data2**. This is the Cartesian product of the two input data sets.
5. Finally, PROC SQL eliminates from the output data set those observations that do not satisfy the condition in the WHERE clause of the program. In this example, observations that do not have matching values for X are eliminated so that the two columns for X have identical values for each observations.
6. The results are written to the output data set. Only one of the X columns is included in the output data set; because they have identical values for each observation, it does not matter which X column is kept and which is discarded. In this example, the output data set **Work.Data4** contains three variables and four observations. None of the observations in **Work.Data4** contains any missing values.

Earlier in this lesson, you learned that a DATA step match-merge will probably not produce the desired results when the data sources that you want to combine have a many-to-many match. You also learned that PROC SQL and the DATA step match-merge do not, by default, produce the same results when you are combining data sources that contain nonmatching data. Now that you have seen how DATA step match-merges and PROC SQL joins work, let's take a look at an example of using each of these techniques to combine data from a many-to-many match that also contains nonmatching data.

### **Example: Combining Data from a Many-to-Many Match**

Suppose you want to combine the data from **Sasuser.Flightschedule** and **Sasuser.Flightattendants**. The **Sasuser.Flightschedule** data set contains information about flights that have been scheduled for a fictional airline. The data set **Sasuser.Flightattendants** contains information about the flight attendants of a fictional airline. A partial listing of each of these data sets is shown below.

**Sasuser.Flightschedule (Partial Listing)**

Date	Destination	FlightNumber	EmpID
01MAR2000	YYZ	132	1739
01MAR2000	YYZ	132	1478
01MAR2000	YYZ	132	1130
01MAR2000	YYZ	132	1390
01MAR2000	YYZ	132	1983
01MAR2000	YYZ	132	1111
01MAR2000	YYZ	182	1076
01MAR2000	YYZ	182	1118

**Sasuser.Flightattendants (Partial Listing)**

EmpID	JobCode	LastName	FirstName
1350	FA3	Arthur	Barbara
1574	FA2	Cahill	Marshall
1437	FA3	Carter	Dorothy
1988	FA3	Dean	Sharon
1983	FA2	Dunlap	Donna
1125	FA2	Eaton	Alicia
1475	FA1	Fields	Diana
1422	FA1	Fletcher	Marie

Suppose you want to combine all variables from the **Sasuser.Flightschedule** data set with the first and last names of each flight attendant who is scheduled to work on each flight. **Sasuser.Flightschedule** contains data for 45 flights, and three flight attendants are scheduled to be on each flight. Therefore, your output data set should contain 135 observations (three for each flight).

You could use the following PROC SQL step to combine **Sasuser.Flightschedule** with **Sasuser.Flightattendants**.

```
proc sql;
  create table flighttemps as
    select flightschedule.* , firstname, lastname
    from sasuser.flightschedule, sasuser.flightattendants
```

```
      where flightschedule.empid=flightattendants.empid;
      quit;
```

The resulting **Flighttemps** data set contains 135 observations. (Page 100)

Now, suppose you use the following DATA step match-merge to combine these two data sets.

```
proc sort data=sasuser.flightattendants out=fa;
  by empid;
run;
proc sort data=sasuser.flightschedule out=fs;
  by empid;
run;
data flighttemps2;
  merge fa fs;
  by empid;
run;
```

The resulting **Flighttemps2** data set contains 272 observations. (Page 100)

The DATA step match-merge does not produce the correct results because it combines the data sequentially. In the correct results, there are three observations for each unique flight from **Sasuser.Flightschedule**, and there are no missing values in any of the observations. By contrast, the results from the DATA step match-merge contain six observations for each unique flight and many observations that have missing values.

In the last example, data from two data sets that have a many-to-many match was combined. The PROC SQL join produced the correct results, but the DATA step match-merge did not. However, you can produce the correct results in a DATA step. First, let's look at using multiple SET statements to combine data.

### Using Multiple SET Statements

You can use multiple SET statements to combine observations from several SAS data sets.

For example, the following DATA step creates a new data set named **Combine**. Each observation in **Combine** contains data from one observation in **Dataset1** and data from one observation in **Dataset2**.

```
data combine;
  set dataset1;
  set dataset2;
run;
```

When you use multiple SET statements,

- processing stops when SAS encounters the end-of-file (EOF) marker on **either** data set (even if there is more data in the other data set). Therefore, the output data set contains the same number of observations as the smallest input data set.
- the variables in the program data vector (PDV) are **not** reinitialized when a second SET statement is executed.
- for any variables that are common to both input data sets, the value or values from the data set in the second SET statement will overwrite the value or values from the data set in the first SET statement in the PDV.

Keep in mind that using multiple SET statements to combine data from multiple input sources that do not have a one-to-one match can be complicated. By default, the first observation from each data set is combined, the second observation from each data set is combined, and so on until the first EOF marker is reached in one of the data sets. Therefore, if you are working with data sources that do not have a one-to-one match, or that contain nonmatching data, you will need to add additional DATA step syntax in order to produce the results that you want.

### Example: Using Multiple SET Statements with a Many-to-Many Match

Remember that in the previous example you wanted to combine **Sasuser.Flightschedule** with **Sasuser.Flightattendants**. Your resulting data set should contain all variables from the **Sasuser.Flightschedule** data set with the first and last names of each flight attendant who is scheduled to work on each flight. **Sasuser.Flightschedule** contains data for 45 flights, and three flight attendants are scheduled to be on each flight. Therefore, your output data set should contain 135 observations (three for each flight).

You can use the following DATA step to perform this table lookup operation. In this program, the first SET statement reads an observation from the **Sasuser.Flightschedule** data set. Then the DO loop executes, and the second SET statement reads each observation in **Sasuser.Flightattendants**. The EmpID variable in **Sasuser.Flightattendants** is renamed so that it does not overwrite the value for EmpID that has been read from **Sasuser.Flightschedule**. Instead, these two values are used for comparison to control which observations from **Sasuser.Flightattendants** should be included in the output data set for each observation from **Sasuser.Flightschedule**.

```
data flighttemps3(drop=empnum jobcode);
  set sasuser.flightschedule;
  do i=1 to num;
    set sasuser.flightattendants
      (rename=(empid=empnum))
      nobs=num point=i;
    if empid=empnum then output;
  end;
run;
```

The resulting **Flighttemps3** (Page 100) data set contains 135 observations and no missing values. Keep in mind that although it is possible to use a DATA step to produce the same results that a PROC SQL join creates by default, the PROC SQL step might be much more efficient.

In which of the following cases will a basic DATA step match-merge and a simple PROC SQL inner join produce identical results?

- Q.5.**
- A only when there is a one-to-one match between input data sets on the values of BY variables
  - B when there is either a one-to-one match or a one-to-many match between input data sets on the values of the BY variables
  - C when there is either a one-to-one match, a one-to-many match, or a many-to-many match between input data sets on the values of the BY variables
  - D These two methods will always produce identical results.

Which of the following is **not** an advantage of the DATA step match-merge?

- Q.6.**
- A It allows for complex business logic to be incorporated into the new data set by using DATA step processing such as arrays and DO loops in addition to merging features.
  - B Multiple BY variables enable lookups that depend on more than one variable.
  - C Data sets do not have to be sorted or indexed.

### Combining Summary Data and Detail Data

You've seen how to combine data from multiple data sets. Suppose you want to calculate percentages based on individual values from a data set as compared to a summary statistic of the data. You need to

- create a summary statistic
- combine the summary information with the detail rows of the original data set, and
- calculate the percentages.

For example, the data set **Sasuser.Monthsum** has one row for every value of SaleMon (month and year) from 1997 to 1999. Each row contains information about the revenue generated by an airline.



Note that the SaleMon variable has a label of Sales Month in the **Sasuser.Monthsum** data set.

**SAS Data Set Sasuser.Monthsum (Partial Listing)**

Sales Month	RevCargo	MonthNo
JAN1997	\$171,520,869.10	1
JAN1998	\$238,786,807.60	1
JAN1999	\$280,350,393.00	1
FEB1997	\$177,671,530.40	2
FEB1998	\$215,959,695.50	2
FEB1999	\$253,999,924.00	2

Suppose you want to produce a report that shows what percentage of the total cargo revenue for the three-year period was generated in each month. You could summarize the data to get the total revenue for cargo for the three-year period and assign that value to a new variable called Cargosum in a summary data set.

Summary data set
Cargosum
\$8,593,432,002.35

Then you would need to combine the summary data (Cargosum) with the detail data in **Sasuser.Monthsum** so that you could calculate percentages of the total cargo revenue for each month.

**Partial listing of the combined data set**

Sales Month	RevCargo	MonthNo	Cargosum	PctRev
JAN1997	\$171,520,869.10	1	\$8,593,432,002.35	<RevCargo/Cargosum>
JAN1998	\$238,786,807.60	1	\$8,593,432,002.35	<RevCargo/Cargosum>
JAN1999	\$280,350,393.00	1	\$8,593,432,002.35	<RevCargo/Cargosum>
FEB1997	\$177,671,530.40	2	\$8,593,432,002.35	<RevCargo/Cargosum>
FEB1998	\$215,959,695.50	2	\$8,593,432,002.35	<RevCargo/Cargosum>
FEB1999	\$253,999,924.00	2	\$8,593,432,002.35	<RevCargo/Cargosum>

Let's examine this task more closely.

### The MEANS Procedure

You should already know how to use the MEANS procedure for producing summary statistics. By default, PROC MEANS generates a report that contains descriptive statistics. The descriptive statistics can be routed to a SAS data set by using an **OUTPUT statement**, and the default report can be suppressed by using the **NOPRINT option**.

#### General form, PROC MEANS with OUTPUT statement:

```
PROC MEANS DATA=original-SAS-data-set NOPRINT;
  <VAR variable(s);>
  OUTPUT OUT= output-SAS-data-set
    statistic=output-variable(s);
RUN;
```

where

- *original-SAS-data-set* identifies the data set on which the summary statistic is generated
- *variable(s)* is the name(s) of the variable(s) that is being analyzed
- *output-SAS-data-set* names the data set where the descriptive statistics will be stored
- *statistic* is one of the summary statistics generated
- *output-variable(s)* names the variable(s) in which to store the value(s) of *statistic* in the output data set.

The output data set that a PROC MEANS step creates contains the requested statistics as values for *output-variable(s)*, as well as two additional variables that are automatically included, as follows:

- *\_TYPE\_* contains information about the class variables
- *\_FREQ\_* contains the number of observations that an output level represents.

#### Example

The following program creates a summary data set named **Sasuser.Summary**. **Summary** contains the sum of the values of Revcargo from **Sasuser.Monthsum**, stored in the variable Cargosum.

```
proc means data=sasuser.monthsum noprint;
  var revcargo;
  output out=sasuser.summary sum=Cargosum;
run;
proc print data=sasuser.summary;
run;
```

Because of the NOPRINT option, the PROC MEANS step does not produce any output. Printing the **Sasuser.Summary** data set produces the following output.

Obs	_TYPE_	_FREQ_	Cargosum
1	0	36	\$8593432002.35

Once you have created the summary statistic, you need to combine this summary information with the detail rows of the data set so that you can calculate the percentages. Remember that you can use multiple SET statements to combine data horizontally. Let's take a closer look at how this works by using multiple set statements to combine the detail rows of **Sasuser.Monthsum** with the detail data that we created in **Sasuser.Summary**.

### Example

The example in the animation below creates a new data set named **Percent1** that combines

- summary data (total revenue for cargo from the three-year period) from **Sasuser.Summary**
- detail data (month and total cargo for the month) from **Sasuser.Monthsum**.

**Percent1** also contains a new variable named PctRev that records the calculated percentage of the total revenue that each observation represents.

Remember, the automatic variable **\_N\_** keeps track of how many times the DATA step has begun to execute. The following DATA step uses **\_N\_** to keep SAS from reaching the EOF marker for **Sasuser.Summary** after the first iteration of the step. Since the variables in the PDV will not be reinitialized on each iteration, the first value of **Summary.Cargosum** will be retained in the PDV for each observation that is read from **Sasuser.Monthsum**.

### How it works

- This DATA step creates a new data set that combines summary data from one input data set (**Sasuser.Summary**) and detail data from a second input data set (**Sasuser.Monthsum**).

```
data sasuser.percent1(drop=cargosum);
  if _N_=1 then set sasuser.summary(keep=cargosum);
  set sasuser.monthsum(keep=salemon revcargo);
  PctRev=revcargo/cargosum;
run;
```

- During the compilation phase, SAS reads the descriptor portion of the input data set and creates the PDV. **\_N\_** is a temporary variable that is included in the PDV although it will not be included in the output data set.
- Execution begins. On the first iteration of the DATA step, **\_N\_** has a value of 1. The IF statement evaluates as true, so the first SET statement executes and SAS reads the value of Cargosum from **Sasuser.Summary** and records it in the PDV.
- Next, the second SET statement executes. SAS reads the first observation in **Sasuser.Monthsum** and records the values in the PDV.
- SAS calculates the value of PctRev and records it in the PDV.
- At the end of the DATA step, SAS writes the values in the PDV to the output data set. **\_N\_** is not included in the output data set since it is a temporary variable. CargoSum is dropped from the output data set as well.

7. On the second iteration of the DATA step, the value of `_N_` is 2, so the IF statement evaluates to false and the first SET statement does not execute. However, the value of CargoSum is retained in the PDV.
8. The second SET statement executes. SAS reads the second observation from **Sasuser.Monthsum** and records the values in the PDV.
9. The value for PctRev is calculated and recorded in the PDV. SAS will write the values in the PDV to the output data set (except for `_N_` and CargoSum).
10. The DATA step will continue to execute until all observations have been read from **Sasuser.Monthsum**.

Complete the following step to combine the data from **Sasuser.Summary** and **Sasuser.Cargorev**. Assume that there is one observation in **Sasuser.Summary** that contains the total revenue generated by cargo, and that there are fifty observations in **Sasuser.Cargorev** that contain data on the revenue generated by cargo on specific flights.

## Q.7.

```
data RevbyFlt;
-----  
set sasuser.cargorev;  
PctRev=cargo$um/revcargo;  
run;
```

Another method of combining summary data and detail data is to create the summary statistic in a DATA step and combine it with the detail data in the same step. To do this you must

- read the data once and calculate the summary statistic
- re-read the data to combine the summary statistic with the detail data and calculate the percentages.

## The Sum Statement

If you took the lesson **Creating and Managing Variables**, you learned how to use the sum statement to obtain a summary statistic within a DATA step. The sum statement adds the result of an expression to an **accumulator variable**.

### General form, sum statement:

*variable+expression;*

where

- *variable* specifies the name of the accumulator variable. This variable must be numeric. The variable is automatically set to 0 before the first observation is read. The variable's value is retained from one DATA step execution to the next.
- *expression* is any valid SAS expression.



If the *expression* produces a missing value, the sum statement ignores it. (Remember, however, that assignment statements assign a missing value if the *expression* produces a missing value.)



The sum statement is one of the few SAS statements that doesn't begin with a keyword.

The sum statement adds the result of the expression that is on the right side of the plus sign (+) to the numeric variable that is on the left side of the plus sign. At the top of the DATA step, the value of the numeric variable is not set to missing as it usually is when reading raw data. Instead, the variable retains the new value in the program data vector for use in processing the next observation.

## Example

The program in the following animation uses a sum statement to generate the summary statistic in a DO UNTIL loop. On the first execution of the DATA step, the DO UNTIL loop reads each observation of **Sasuser.Monthsum** and keeps a running tally of the total value of RevCargo from each observation. On each subsequent execution of the DATA step, this tally (stored in the variable TotalRev) is divided into RevCargo in order to calculate the new variable PctRev.



Remember that the END= data set option creates a temporary variable that contains an end-of-file indicator.

## How it works

1. This animation shows the execution of the DATA step below. This DATA step reads the same data set, **Sasuser.Monthsum**, twice: first, to create a summary statistic; second, to merge the summary statistic back into the detail data to create a new data set, **Sasuser.Percent2**.

```
data sasuser.percent2(crop=totalrev);
if _N_=1 then do until (LastObs);
  set sasuser.monthsum(keep=revcargo) end=lastobs;
  TotalRev+revcargo;
end;
set sasuser.monthsum(keep=salemon revcargo);
PctRev=revcargo/totalrev;
run;
```

2. During the compilation phase, SAS reads the descriptor portion of the input data set and creates the PDV. `_N_`, `LastObs`, and `TotalRev` are temporary variables that are included in the PDV although they will not be included in the output data set.
3. Execution begins. The temporary variables are initialized with values. The IF statement resolves to true on the first iteration of the DATA step, so the DO UNTIL loop begins to execute. Remember, in a DO UNTIL loop, the condition is evaluated at the bottom of the loop.
4. SAS reads the first observation from **Sasuser.Monthsum** and writes the value for RevCargo to the PDV.
5. The value of TotalRev is increased by the value of RevCargo and recorded in the PDV.
6. At the bottom of the DO loop, SAS evaluates the UNTIL expression. It resolves to false since the value of LastObs is 0, so the loop continues to execute.
7. SAS reads the second observation from **Sasuser.Monthsum**, overwriting the value for RevCargo in the PDV and adding this value to the accumulator variable TotalRev.
8. The DO UNTIL loop continues to execute until SAS reads the last observation from **Sasuser.Monthsum** and the value of LastObs is set to 1. At this point, the value for TotalRev in the PDV is the sum of all values for RevCargo in **Sasuser.Monthsum**. The loop is satisfied.
9. The second SET statement reads data from the same data set as the first SET statement did. However, this time values for both SaleMon and RevCargo are recorded in the PDV. There is already a value for TotalRev in the PDV.
10. A value for PctRev is calculated for the observation and recorded in the PDV. Then, SAS writes the values in the PDV to the output data set **Sasuser.Percent2**, except for the temporary variables and the variable TotalRev.
11. On the second iteration of the DATA step, the value of `_N_` increases to 2, so the IF expression is false. The second SET statement executes and values from the second observation of **Sasuser.Monthsum** are read into the PDV.
12. The value for the accumulator variable TotalRev is retained from the last iteration and is used to calculate a new value for PctRev, which is recorded in the PDV. SAS writes the values in the PDV to the output data set and the DATA step iterates.

13. The DATA step iterates until SAS has read the last observation from **Sasuser.Monthsum** and has written the new observation to the output data set **Sasuser.Percent2**.

Complete the following DATA step to create a new variable named TotalExp that records the total of the values contained in the Expenses variable in **Sasuser.Expenses**. The DATA step should also calculate the percentage of this total that each value of Expenses represents.

**Q.8.**

```

data exspndat;
  if _n_=1 then do until(lastobs);
    set sasuser.expenses(keep=expenses) end=lastobs;

  _____
  end;
  set sasuser.expenses;
  PctRev=expenses/totalexp;
run;
```

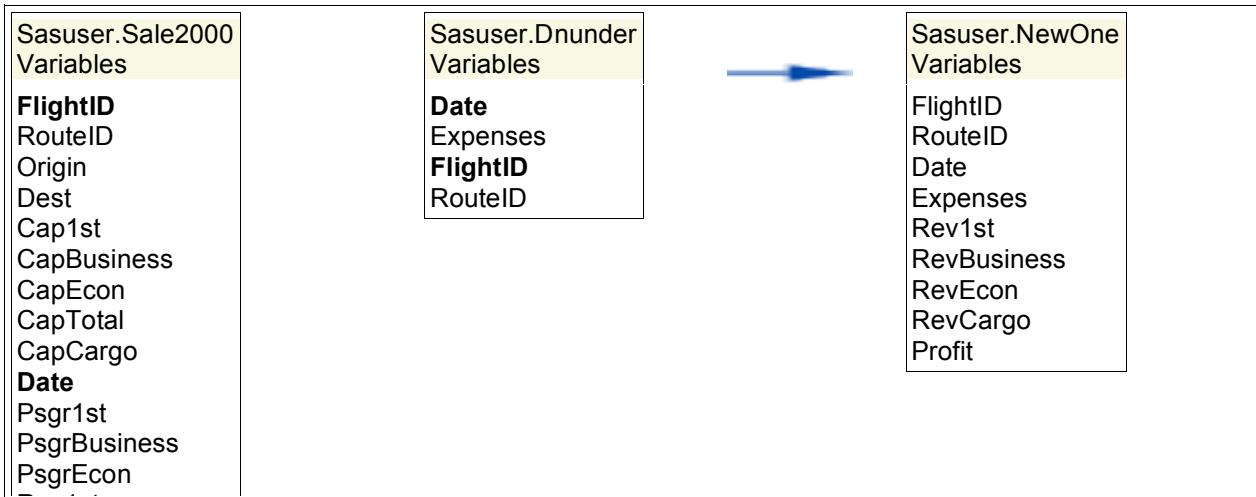
## Using an Index to Combine Data

Suppose you want to combine data from two data sets, and one of the data sets is much larger than the other. Also, suppose you want to use only those observations from the larger data set that match an observation from the smaller data set according to the value of one or more common variables.

You should already know how to create an index on a SAS data set. You have learned that PROC SQL can take advantage of an index to improve performance on a join. You can also take advantage of an index in a DATA step to combine data from matching observations in multiple data sets if the index is built on variables that are common to all input data sets.

For example, suppose you want to combine data from the matching observations in **Sasuser.Dnunder** and **Sasuser.Sale2000**. Only a portion of the flights that are in **Sasuser.Sale2000** (which has 156 observations) are also in **Sasuser.Dnunder** (which has only 57 observations). Suppose you want to use only data from **Sasuser.Sale2000** about flights that are in both data sets.

Assume that **Sasuser.Sale2000** has a composite index named **Flightdate** associated with it. The values for **Flightdate** are unique and are based on the values of the variables FlightID and Date. You can use **Flightdate** to combine data from only the observations in both data sets that have matching values for FlightID and Date.



Rev1st
RevBusiness
RevEcon
SaleMon
CargoWgt
RevCargo

The next few pages show how to use the **Flightdate** index to combine matching observations from the **Sasuser.Sale2000** data set and the **Sasuser.Dnunder** data set.

### The KEY= Option

You have seen how to use multiple SET statements in a DATA step in order to combine summary data and detail data in a new data set. You can also use multiple SET statements to combine data from multiple data sets if you want to combine only data from observations that have matching values for particular variables.

You specify the KEY= option in the SET statement to use an index to retrieve observations from the input data set that have key values equal to the key variable value that is currently in the program data vector (PDV).

#### General form, SET statement with KEY= option:

**SET SAS-data-set-name KEY= index-name;**

where *index-name* is the name of an index that is associated with the **SAS-data-set-name** data set.

To use the SET statement with the KEY= option to perform a lookup operation, your lookup values must be stored in a SAS data set that has an index. This technique is appropriate only when you are working with one-to-one matches, and you can use it with a lookup table of any size. It is possible to return multiple values with this technique, and you can use other DATA step syntax with it as well.

When SAS encounters the SET statement that includes the KEY= option, there must already be a value in the PDV for the value or values of the key variable(s) on which the KEY= index is built. SAS can then use the index to retrieve an observation that has a value for the key variable that matches the key value from the PDV.

For example, if the **Sasuser.Sale2000** data set has an index named **Flightdate** associated with it, the following SET statement uses the **Flightdate** index to locate observations in **Sale2000** that have specific values for FlightID and Date:

```
set sasuser.sale2000 key=flightdate;
```

When the SET statement in the example above begins to execute, there must already be a value for FlightID and a value for Date in the PDV. SAS then uses the **Flightdate** index to retrieve an observation from **Sasuser.Sale2000** that has values for FlightID and Date that match the values for FlightID and Date that are already in the PDV.

In order to assign a key value in the PDV before the SET statement with the KEY= option executes, you precede that SET statement with another SET statement in the DATA step. Let's look more closely at this example in context.

## Example

Remember that you want to combine **Sasuser.Sale2000** and **Sasuser.Dnunder**, and that **Sasuser.Sale2000** has an index named **Flightdate** that is based on the values of the FlightID and the Date variables. You can use two SET statements to combine these two data sets, and you can use the KEY= option on the second SET statement to take advantage of the index.

In the following example,

- the first SET statement reads an observation sequentially from the **Sasuser.Dnunder** data set. SAS writes the values from this observation to the PDV, and then moves to the second SET statement.
- SAS will use the **Flightdate** index on **Sasuser.Sale2000** to find an observation in **Sasuser.Sale2000** that has values for FlightID and Date that match the values of FlightID and Date that are currently in the PDV.
- **Work.Profit** is the output data set. (Page 100)



If you use the KEY= option to read a SAS data set, you cannot use WHERE processing on that data set in the same DATA step.

**How it works.** The following steps shows the execution of a DATA step that uses two SET statements to combine data from two input data sets (**Sasuser.Sale2000** and **Sasuser.Dnunder**) into one output data set (**Work.Profit**).

1. The DATA step also uses an index on the larger of the two input data sets, which is **Sasuser.Sale2000**, to find matching observations.

```
data work.profit;
  set sasuser.dnunder;
  set sasuser.sale2000(keep=routeid flightid date revlst
                      revbusiness revecon revcargo)
    key=flightdate;
  Profit=sum(revlst, revbusiness, revecon, revcargo,
             -expenses);
run;
```

2. SAS sets up the new data set by reading the descriptor portions of the input data sets and creating the PDV.
3. The first SET statement executes. SAS reads the first observation in **Sasuser.Dnunder** and records the values in the PDV.
4. When the second SET statement executes, the KEY= option causes SAS to use the **Flightdate** index to directly access the observation in **Sasuser.Sale2000** that has values for FlightID and Date that match the values already in the PDV. SAS reads the observation and records the values to the PDV.
5. SAS calculates the value for Profit and records it in the PDV. Then, SAS writes the values from the PDV to the output data set.
6. The DATA step continues to iterate. Only the variable Profit is reinitialized to missing. SAS reads the second observation in **Sasuser.Dnunder** and records the values in the PDV, overwriting the values that have been retained.
7. SAS uses the **Flightdate** index to find a matching observation in **Sasuser.Sale2000**. Then, SAS records the values from that observation in the PDV, overwriting the values that have been retained. A new value for Profit is calculated and recorded, and the values are written to the output data set.

8. The DATA step continues to iterate until all observations have been read from **Sasuser.Dnunder**.

Remember that when SAS encounters the SET statement that includes the KEY= option, there must already be a value in the PDV for the value or values of the key variable(s) on which the KEY= index is built. Otherwise, the step will generate errors in the output data set.

### Example

The following step is identical to the last example except that the order of the SET statements has been reversed:

```
data work.profit2;
  set sasuser.sale2000(keep=routeid flightid date
    revlst revbusiness revecon revcargo)
    key=flightdate;
  set sasuser.dnunder;
  Profit=sum(revlst, revbusiness, revecon, revcargo,
    -expenses);
run;
```

On the first iteration of this DATA step, there are no values for the key values in the PDV when SAS encounters the SET statement with the KEY= option. Therefore, SAS does not know what to look up in the index, and no observation is read from the **Sasuser.Sale2000** data set. SAS proceeds to the second SET statement, reads an observation from the **Sasuser.Dnunder** data set, and writes the values to the PDV and to the **Work.Profit2** data set. (Page 100)

Since no data was read from the **Sasuser.Sale2000** data set, there are missing values in the first observation of the output data set. Also, if you examine the values for Rev1st, RevBusiness, RevEcon and RevCargo in **Work.Profit2** and compare them with the values for these variables in **Work.Profit**, you will notice that there are differences between these two data sets.

Remember that the values in the PDV are not reinitialized after each iteration of the DATA step. On the second iteration of the DATA step, SAS uses the values from the first observation of **Sasuser.Dnunder** to match an observation from **Sasuser.Sale2000**. But before these values are written to the **Work.Profit2** data set, a new observation is read from **Sasuser.Dnunder** and written to the PDV. Therefore, none of the observations in **Work.Profit2** actually contains correctly matched data from the two input data sets.

Complete the following DATA step to create the **Empsals** data set. Use the **EmplD** index on **Sasuser.Empdata** to read only those observations from the **Sasuser.Empdata** data set that match observations in the **Sasuser.Newsals** data set according to the **EmplD** variable.

**Q.9.**

```
data Empsals;
  set sasuser.newsals;
  _____
run;
```

You have seen how to use a SET statement with the KEY= option in conjunction with a second SET statement to create a data set that combines data from matching observations of two input data sets. Remember that when you use multiple SET statements, the variables in the PDV are **not** reinitialized when the second SET statement is executed. This can lead to problems in the output data set.

Suppose SAS reads an observation from the first input data set on the second iteration of the DATA step (that is, when **\_N\_=2**) and does not find a matching observation in the second input data set. Because the

DATA step has already iterated once, and the values in the PDV have not been reinitialized, there are already values in the PDV for all variables. Therefore, the resulting observation in the output data set will contain values from the second observation of the first input data set, combined with values from the first observation of the second input data set.

### Example

If you examine the **Work.Profit** output data set (Page 100) closely, you will notice that the final observation in the output data set contains values for several variables that are identical to values in the previous observation. This duplication of data is incorrect, although the error might not be obvious.

The error in the output data set is caused by a data error in one of the input data sets. If you examine the **Sasuser.Dnunder** data set closely, you will find that all of the values for FlightID begin with the characters *I*A10 except the value in the last observation of the data set. Instead, the value for FlightID in the last observation begins with the characters *I*A11. This is a data error. Because of the data error, when the DATA step executes SAS will not be able to find a matching observation in **Sasuser.Sale2000** for the last observation in **Sasuser.Dnunder**, and will write an observation to the output data set that contains data from the last observation in **Sasuser.Dnunder** and data from the previous DATA step iteration for **Sasuser.Sale2000**.

The SAS log provides an additional indication that the final observation in the output data set contains nonmatching data. The observation that contains unmatched data is printed to the log. As you can see in the log sample below, the unmatched observation includes an `_Error_` variable whose value is 1, which indicates that there is an error. The `_N_` variable indicates the iteration of the DATA step in which the error occurred.

### SAS Log

```
FlightID=IA11802 RouteID=0000108 Date=30DEC2000 Expenses=3720
Revlst=1270 RevBusiness=. RevEcon=5292 RevCargo=1940 Profit=4782
ERROR =1 _IORC_=1230015 _N_=57
NOTE: There were 57 observations read from the data set
      SASUSER.DNUNDER.
NOTE: The data set WORK.PROFIT has 57 observations and 9 variables.
NOTE: DATA statement used (Total process time):
      real time      0.38 seconds
      cpu time       0.04 seconds
```

You could check the SAS log for observations that contain errors in order to ensure that your output data set does not contain bad data, but there is a better way. Notice that the observation that is printed in the SAS log above also contains a variable named `_IORC_`. You can use the value of the `_IORC_` variable to prevent the observations that contain errors from being written to your output data set.

### The `_IORC_` Variable

When you use the KEY= option, SAS creates an automatic variable named `_IORC_`, which stands for **INPUT/OUTPUT Return Code**. You can use `_IORC_` to determine whether the index search was successful. If the value of `_IORC_` is zero, SAS found a matching observation. If the value of `_IORC_` is not zero, SAS did not find a matching observation.



The `_IORC_` variable is also automatically created when you use the MODIFY statement with the KEY= option in the DATA step.

On the previous page, you saw an example in which a data error was included in the output data set and was written to the SAS log. To prevent writing the data error to the log (and to your output data set),

- check the value of `_IORC_` to determine whether a match has been found
- set `_ERROR_` to 0 if there is no match
- delete the nonmatching data or write the nonmatching data to an errors data set.

### Example

The following example uses the **Flightdate** index to combine data from **Sasuser.Sale2000** with data from **Sasuser.Dnunder**, and writes the combined data to a new data set named **Work.Profit3**. (Page 100) If any unmatched observations are read from **Sasuser.Dnunder**, the resulting combined observation will be written to **Work.Errors**. No observations should be written to the SAS log.

```
data work.profit3 work.errors;
set sasuser.dnunder;
set sasuser.sale2000(keep=routeid flightid date revlst
revbusiness revecon revcargo)key=flightdate;
if _iorc_=0 then do;
Profit=sum(revlst, revbusiness, revecon, revcargo,
-expenses);
output work.profit3;
end;
else do;
_error_=0;
output work.errors;
end;
run;
```

If you examine the results from the program above, you will notice that there is one fewer observation in the **Work.Profit3** output data set than there was in the **Work.Profit** output data set. The extra observation has been written to the **Work.Errors** data set because it contains a data error.

### Example Results

**SAS Data Set Work.Errors**

Obs	FlightID	RouteID	Date	Expenses	Rev1st	RevBusiness	RevEcon	RevCargo	Profit
1	IA11802	0000108	30DEC2000	3720	1524	.	5124	2020	.

### SAS Data Set Work.Profit3 (Page 100)

During execution of the DATA step shown below, what does a value of 0 for `_IORC_` indicate?

**Q.10.**

```
data newsset;
set sasuser.empdata;
set sasuser.newsals key=empid;
if _iorc_=0 then do;
output work.newsals;
end;
run;
```

- A The most recent SET statement with the KEY= option did not successfully find a matching observation.
- B The most recent SET statement with the KEY= option did successfully find a matching observation.
- C You must also check the value of `_ERROR_` to know if the SET statement executed successfully.

## Using a Transactional Data Set

Sometimes, rather than just combining data from two data sets, you might want to update the data in one data set with data that is stored in another data set. That is, you might want to update a master data set by overwriting certain values in it with values that are stored in a transactional data set.

For example, suppose the data set **Mylib.Empmaster** contains some information that is outdated. You have the current data stored in another data set named **Mylib.Empchanges**. **Mylib.Empmaster** contains 148 observations, and **Mylib.Empchanges** contains 6 observations. The variable EmpID contains unique values in both data sets.

A partial listing of **Mylib.Empmaster** and the full listing of **Mylib.Empchanges** is shown below. Notice that there is one observation's in each data set with a value of 1065 for EmpID, and that the values of JobCode and Salary are different in for this observation.

**Mylib.Empmaster (Partial Listing)**

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
05MAR1957	30MAR1990	1009	M	TA1	\$40,432
01JAN1956	20OCT1979	1017	M	TA3	\$57,201
23MAY1963	27OCT1982	1036	F	TA3	\$55,149
14APR1962	17SEP1990	1037	F	TA1	\$39,981
13NOV1967	26NOV1989	1038	F	TA1	\$37,146
17JUL1961	27AUG1984	1050	M	ME2	\$49,234
<b>29JAN1942</b>	<b>10JAN1985</b>	<b>1065</b>	<b>M</b>	<b>ME2</b>	<b>\$49,126</b>
18OCT1970	06OCT1989	1076	M	PT1	\$93,181

**Mylib.Empchanges**

DateOfBirth	DateOfHire	EmpID	Gender	JobCode	Salary
30JUN1955	31JAN1982	1639	F	TA3	\$59,164
<b>29JAN1942</b>	<b>10JAN1985</b>	<b>1065</b>	<b>M</b>	<b>ME3</b>	<b>\$53,326</b>
03DEC1961	10Oct1985	1561	M	TA3	\$51,120
25SEP1965	07OCT1989	1221	F	FA3	\$41,854
11AUG1970	01NOV2000	1447	F	FA1	\$30,340
13SEP1968	05NOV2000	1998	M	SCP	\$32,240

If you could see the full listing of **Mylib.Empmaster**, you would see that each of the observations in **Mylib.Empchanges** has a matching observation in **Mylib.Empmaster** based on the values of EmpID. There are also many observations in **Mylib.Empmaster** that do not have a matching observation in **Mylib.Empchanges**. To update **Mylib.Empmaster**, you want to find all of the matching observations and change their values for JobCode and Salary with the new values from **Mylib.Empchanges**. You can use the UPDATE statement to make these changes.



Note that **Mylib.Empmaster** and **Mylib.Empchanges** are fictitious data sets that are used for this example. The **Mylib** library and the **Empmaster** and **Empchanges** data sets are not included in the practice data for this lesson; therefore, the sample programs on the next page will not run.

## Using the UPDATE Statement

You use the UPDATE statement to update a master data set with a transactional data set. The UPDATE statement can perform the following tasks.

- change the values of variables in the in the master data set.
- add observations to the master data set.
- add variables to the master data set.

### General form, UPDATE statement:

```
DATA master-data-set;
UPDATE master-data-set transaction-data-set;
BY by-variable(s);
RUN;
```

where

- *master-data-set* names the SAS data set used as the master file
- *transaction-data-set* names the SAS data set that contains the changes to be applied to the master data set
- *by-variable(s)* names a variable that appears in both *master-data-set* and in *transaction-data-set*. Each observation in *master-data-set* must have a unique value for *by-variable*, but *transaction-data-set* can contain more than one observation with the same *by-variable* value.

The UPDATE statement replaces values in the master data set with values from the transactional data set for each observation with a matching value of the BY variable. Any observations in either the master data set or the transactional data set that have non-matching values for the BY variable are included in the output data set. Also, by default, SAS does not replace existing values in the master data set with missing values if those values are coded as periods (for numeric variables) or blanks (for character variables) in the transaction data set.

When you use the UPDATE statement, keep in mind the following restrictions.

- Only two data set names can appear in the UPDATE statement.
- The master data set must be listed first.
- A BY statement that gives the matching variable must be used.
- Both data sets must be sorted by or have indexes based on the BY variable.
- In the master data set, each observation must have a unique value for the BY variable.

### Example

Remember that you want to update the master data set **Mylib.Empmaster** with the transactional data set **Mylib.Empchanges**. You can use the UPDATE statement to accomplish this task, as shown the program below. Remember, both data sets must be sorted by or indexed on the BY variable.

```
proc sort data=mylib.empmaster;
  by empid;
run;
proc sort data=mylib.empchanges;
  by empid;
run;
data mylib.empmaster;
  update mylib.empmaster mylib.empchanges;
  by empid;
run;
```

The first 8 observations of the updated **Mylib.Empmaster** data set are shown below. Notice that the observation that has a value of 1065 for EmpID now contains the updated values for JobCode and Salary.

DateofBirth	DateofHire	EmplID	Gender	JobCode	Salary
05MAR1957	30MAR1990	1009	M	TA1	\$40,432
01JAN1956	20OCT1979	1017	M	TA3	\$57,201
23MAY1063	27OCT1982	1036	F	TA3	\$55,149
14APR1962	17SEP1990	1037	F	TA1	\$39.981
13NOV1967	26NOV1989	1038	F	TA1	\$37,146
17JUL1961	27AUG1984	1050	M	ME2	\$49.234
<b>29JAN1942</b>	<b>10JAN1985</b>	<b>1065</b>	<b>M</b>	<b>ME3</b>	<b>\$53,326</b>
18OCT1970	06OCT1989	1076	M	PT1	\$93.181

Which of the following statements is true about using the UPDATE statement to update a master data set with a transactional data set?

**Q11.**

- A The UPDATE statement can change the values of existing variables in the master data set, but it cannot add new variables or observations to the master data set.
- B The master data set and transactional data set must contain at least one common variable.
- C The master data set must be sorted by the variable that is listed in the BY statement, but the transactional data set does not need to be sorted.

## **Chapter Summary**

### **Reviewing Terminology**

You can review definitions of terms that are important in this lesson. You can also review diagrams and descriptions of the various relationships between input sources for a table lookup operation.

### **Working with Lookup Values Outside of SAS Data Sets**

You can use the IF-THEN/ELSE statement in the DATA step to combine data from a base table with lookup values that are not stored in a SAS data set. You can also use the FORMAT procedure or the ARRAY statement to combine data from a base table with lookup values that are not stored in a SAS data set.

### **Combining Data with the DATA Step Match-Merge**

You can use the MERGE statement in the DATA step to combine data from multiple data sets as long as the input data sets have a common variable. You can merge more than two data sets that lack a common variable in multiple DATA steps if each input data set contains at least one variable in it that is also in at least one other input data set.

### **Using PROC SQL to Join Data**

You can also use PROC SQL to join data from multiple data sets if there is no single variable that is common to all input data sets. In a PROC SQL step, you can choose only the specific variables from each input data set that you want to include in the new data set. If you create a new table with the results of an inner join in a PROC SQL step, the results can be very similar to the results of a DATA step match-merge.

### **Comparing DATA Step Match-Merges and PROC SQL Joins**

It is possible to create identical results with a basic DATA step match-merge and a PROC SQL join. However, there are significant differences between these two methods, as well as advantages and disadvantages to each. In some cases, such as when there is a one-to-one or a one-to-many match on values of the BY variables in the input data sets, these two methods produce identical results.

In other cases, such as when there is a many-to-many match on values of the BY variables, or if there are nonmatching values of the BY variables, these two methods will produce different results. These differences reflect the fact that the processing is different for a DATA step match-merge and a PROC SQL join. Even if you are working with many-to-many matches or nonmatching data, it is possible to use other DATA step techniques such as multiple SET statements to create results that are identical to the results that a PROC SQL step creates.

### **Combining Summary Data and Detail Data**

In order to perform tasks such as calculating percentages based on individual values from a data set as compared to a summary statistic of the data, you need to combine summary data and detail data. One way to create a summary data set is to use PROC MEANS. Once you have a summary data set, you can use multiple SET statements to combine the summary data with the detail data in the original data set. It is also possible to create summary data with a sum statement and to combine it with detail data in one DATA step.

### **Using an Index to Combine Data**

You can use an index to combine data from matching observations in multiple data sets if the index is built on variables that are common to all input data sets. Especially if one of the input data sets is very large, an index can improve the efficiency of the merge. You use the KEY= option in a SET statement in conjunction with another SET statement to use an index to combine data. However, this method might result in data errors in the output data set. You can use the \_IORC\_ variable to prevent unmatched data from being included in the output data set.

## Using a Transactional Data Set

Sometimes, you might want to update the data in one data set with data that is stored in another data set. You use the UPDATE statement to update a master data set with a transactional data set. The UPDATE statement replaces values in the master data set with values from the transactional data set for each observations with a matching value of the BY variable.

### Syntax

```
PROC MEANS DATA=original-SAS-data-set NOPRINT;
  <VAR variable(s);>
  OUTPUT OUT=output-SAS-data-set statistic=output-variable(s);
RUN;
DATA libref.data-set-name;
  SET SAS-data-set-name;
  SET SAS-data-set-name KEY=index-name;
  variable+expression;
RUN;
DATA master-data-set;
  UPDATE master-data-set transaction-data-set;
  BY by-variables;
RUN;
```

## Sample Programs

### Combining Data with the IF-THEN/ELSE Statement

```
data mylib.employees_new;
set mylib.employees;
if IDnum=1001 then Birthdate='01JAN1963'd;
else if IDnum=1002 then Birthdate='08AUG1946'd;
else if IDnum=1003 then Birthdate='23MAR1950'd;
else if IDnum=1004 then Birthdate='17JUN1973'd;
run;
```

### Combining Data with the ARRAY Statement

```
data mylib.employees_new;
array birthdates{1001:1004} _temporary_ ('01JAN1963'd
  '08AUG1946'd '23MAR1950'd '17JUN1973'd);
set mylib.employees;
Birthdate=birthdates(IDnum);
run;
```

### Combining Data with the FORMAT Procedure

```
proc format;
  value $birthdate '1001' = '01JAN1963'
    '1002' = '08AUG1946'
    '1003' = '23MAR1950'
    '1004' = '17JUN1973';
run;
data mylib.employees_new;
  set mylib.employees;
  Birthdate=input(put(IDnum,$birthdate.),date9.);
run;
```

## Performing a DATA Step Match-Merge

```
proc sort data=sasuser.expenses out=expenses;
  by flightid date;
run;
proc sort data=sasuser.revenue out=revenue;
  by flightid date;
run;
data revexpns (drop=revlst revbusiness revecon
  expenses);
  merge expenses(in=e) revenue(in=r);
  by flightid date;
  if e and r;
  Profit=sum(revlst, revbusiness, revecon, -expenses);
run;
proc sort data=revexpns;
  by dest;
run;
proc sort data=sasuser.acities out=acities;
  by code;
run;
data sasuser.alldata;
  merge revexpns(in=r) acities
    (in=a rename=(code=dest)
     keep=city name code);
  by dest;
  if r and a;
run;
```

## Performing a PROC SQL Join

```
proc sql;
  create table sqljoin as
  select revenue.flightid,
    revenue.date format=date9., revenue.origin, revenue.dest,
    sum(revenue.revlst, revenue.revbusiness, revenue.revecon) -expenses.expenses as Profit,
    acities.city, acities.name
  from sasuser.expenses, sasuser.revenue, sasuser.acities
  where expenses.flightid=revenue.flightid
    and expenses.date=revenue.date
    and acities.code=revenue.dest
  order by revenue.dest, revenue.flightid, revenue.date;
quit;
```

## Working with a Many-to-Many Match

```
proc sql;
  create table flighttemp as
  select flightschedule.*, firstname, lastname
  from sasuser.flightschedule, sasuser.flightattendants
  where flightschedule.empid=flightattendants.empid;
quit;
data fighttemps3(drop=empnum jobcode)
  set sasuser.flightschedule;
  do i=1 to num;
    set sasuser.flightattendants
      (rename=(empid=empnum))
```

```

nob=num point=i;
if empid=empnum then output;
end;
run;

```

## Combining Summary Data and Detail Data

```

proc means data=sasuser.monthsum noprint;
var revcargo;
output out=sasuser.summary sum=Cargosum;
run;
data sasuser.percent1;
if _n_=1 then set sasuser.summary
(keep=cargosum);
set sasuser.monthsum
(keep=salemon revcargo);
PctRev=revcargo/cargosum;
run;
data sasuser.percent2(drop=totalrev);
if _n_=1 then do until(lastobs);
set sasuser.monthsum(keep=revcargo)
end=lastobs;
totalrev+revcargo;
end;
set sasuser.monthsum (keep=salemon revcargo);
PctRev=revcargo/totalrev;
run;

```

## Using an Index to Combine Data

```

data work.profit work.errors;
set sasuser.dnunder;
set sasuser.sale2000(keep=routeid
flightid date revlst revbusiness
revecon revcargo)key=flightdate;
if _iorc_=0 then do;
Profit=sum(revlst, revbusiness, revecon,
revcargo, -expenses);
output work.profit;
end;
else do;
_error_=0;
output work.errors;
end;
run;

```

## Using a Transactional Data Set

```

proc sort data=mylib.empmaster;
by empid;
run;
proc sort data=mylib.empchanges;
by empid;
run;
data mylib.empmaster;
update mylib.empmaster mylib.empchanges;
by empid;
run;

```

### **Points to Remember**

- In a DATA step match-merge, you can use the RENAME= option to give identical names to variables in input data sets if those variables record the same information in values that have the same type and length.
- You use the OUTPUT statement and the NOPRINT option with the MEANS procedure if you want the results to be routed to an output data set and the default report to be suppressed.
- The automatic variable \_N\_ keeps track of how many times a DATA step has iterated. The \_N\_ variable is useful when you are combining data from a summary data set with data from a larger detail data set.
- When you use the UPDATE statement, both data sets must be sorted by or have indexes based on the BY variable.

## Chapter Quiz

Select the best answer for each question.

1. According to the data set descriptions below, which of the variables listed qualify as BY variables for a DATA step match-merge?

Variable	Type	Length	Description
Code	char	5	Department code
Totemps	num	3	Total number of employees
Region	char	4	Location of the department
Manager	num	5	Employee ID number

Variable	Type	Length	Description
IDnum	num	5	Employee ID number
Name	char	20	Employee name
Division	char	3	Division abbreviation
Hiredate	num	8	Date of hire
Supervisor	char	20	Name of supervisor

- a. Code and IDnum  
b. Manager and Supervisor  
c. Manager and IDnum  
d. There are no variables that are common to both of these data sets.
2. Suppose you want to merge **Dataset1**, **Dataset2**, and **Dataset3**. Also suppose that **Dataset1** and **Dataset2** have the common variable Startdate, **Dataset2** and **Dataset3** have the common variable Instructor, and that these data sets have no other common variables. How can you use a DATA step to merge these three data sets into one new data set?
- a. You use a MERGE statement in one DATA step to merge **Dataset1**, **Dataset2**, and **Dataset3** by Startdate and Instructor.  
b. You sort **Dataset1** and **Dataset2** by Startdate and merge them into a temporary data set in a DATA step. Then you sort the temporary data set and **Dataset3** by Instructor and merge them into a new data set in a DATA step.  
c. You can merge these data sets only with a PROC SQL step.  
d. You cannot merge these three data sets at all because they do not have a common variable.
3. Which of the following programs correctly creates a table with the results of a PROC SQL inner join matched on the values of empcode?
- a.
- ```
proc sql;
  select newsals.empcode allemps.lastname
         newsals.salary contrib.amount
    from sasuser.alldeps, sasuser.contrib,
         sasuser.newsals
   where empcode=allemps.empid
         and empcode=contrib.empid;
quit;
```

```

b. proc sql;
create table usesql as
select newsals.empcode allemps.lastname
      newsals.salsry contrib.amount
   from sasuser.alllemps, sasuser.contrib,
        sasuser.newsals
quit;

c. proc sql;
create table usesql as;
select newsals.empcode, allemps.lastname,
      newsals.salary, contrib.amount;
   from sasuser.alllemps, sasuser.contrib,
        sasuser.newsals;
where empcode=allemps.empid
      and empcode=contrib.empid;
quit;

d. proc sql;
create table usesql as
select newsals.empcode, allemps.lastname,
      newsals.salary, contrib.amount
   from sasuser.alllemps, sasuser.contrib,
        sasuser.newsals
where empcode=allemps.empid
      and empcode=contrib.empid;
quit;

```

4. To process a default DATA step match-merge, SAS first reads the descriptor portion of each data set and sets up the PDV and the descriptor portion of the new data set. Which of the following accurately describes the rest of this process?
- a. Next, SAS sequentially match-merges observations and writes the new observation to the PDV, then to the new data set. When the BY value changes in all the input data sets, the PDV is initialized to missing. Missing values for variables, as well as missing values that result from unmatched observations, are written to the new data set.
  - b. Next, SAS sequentially match-merges observations and writes the new observation to the PDV, then to the new data set. After each DATA step iteration, the PDV is initialized to missing. Missing values for variables, as well as missing values that result from unmatched observations, are omitted from the new data set.
  - c. Next, SAS creates a Cartesian product of all possible combinations of observations and writes them to the PDV, then to the new data set. Then SAS goes through the new data set and eliminates all observations that do not have matching values of the BY variable.
  - d. Next, SAS creates a Cartesian product of all possible combinations of observations and writes them to the PDV, then to the new data set. The new data set is then ordered by values of the BY variable.

5. Which of the following statements is **false** about using multiple SET statements in one DATA step?
- You can use multiple SET statements to combine observations from several SAS data sets.
  - Processing stops when SAS encounters the end-of-file (EOF) marker on either data set (even if there is more data in the other data set).
  - You can use multiple SET statements in one DATA step only if the data sets in each SET statement have a common variable.
  - The variables in the PDV are not reinitialized when a second SET statement is executed.
6. Select the program that correctly creates a new data set named **Sasuser.Summary** that contains one observation with summary data created from the Salary variable of the **Sasuser.Empdata** data set.
- ```
proc sum data=sasuser.empdata noprint;
output out=sasuser.summary sum=Salarysum; run;
```
  - ```
proc means data=sasuser.empdata noprint;
var salary;
output out=sasuser.summary sum=Salarysum; run;
```
  - ```
proc sum data=sasuser.empdata noprint;
var salary;
output out=sasuser.summary sum=Salarysum; run;
```
  - ```
proc means data=sasuser.empdata noprint;
output=sasuser.summary sum=Salarysum; run;
```
7. If the value of Cargosum is \$1000 at the end of the first iteration of the DATA step shown below, what is the value of Cargosum in the PDV when the DATA step is in its third iteration?
- ```
data sasuser.percent1;
if _n_=1 then set sasuser.summary (keep=cargosum);
set sasuser.monthsum (keep=salemon revcargo);
PctRev=revcargo/cargosum;
run;
```
- \$1000
  - \$3000
  - The value is missing.
  - The value cannot be determined without seeing the data that is in **Sasuser.Summary**.
8. According to the data set shown, what is the value of Totalrev in the PDV at the end of the fourth iteration of the DATA step?

Obs	SaleMon	RevCargo
1	JAN1997	\$520.00
2	JAN1998	\$230.00
3	JAN1999	\$350.00
4	FEB1997	.

```

data sasuser.percent2(drop=totalrev);
  if _n_=1 then do until(lastobs);
    set sasuser.monthsum2(keep=revcargo) end=lastobs;
    totalrev+revcargo;
  end;
  set sasuser.monthsum2 (keep=salemon revcargo);
  PctRev=revcargo/totalrev;
run;

```

- a. The value is missing.
- b. \$350.00
- c. \$520.00
- d. \$1100.00

9. Which of the following programs correctly uses an index to combine data from two input data sets?

- a. 

```

data work.profit;
set sasuser.sale2000(keep=routeid flightid date
                      revlst revbusiness revecon revcargo)
      key=flightdate;
set sasuser.dnunder;
Profit=sum(revlst, revbusiness, revecon, revcargo,
           -expenses);
run;
```
- b. 

```

data work.profit;
set sasuser.dnunder;
set sasuser.sale2000(keep=routeid flightid date
                      revlst revbusiness revecon revcargo)
      key=flightdate;
where routeid='0000103';
Profit=sum(revlst, revbusiness, revecon, revcargo,
           -expenses);
run;
```
- c. 

```

data work.profit;
set sasuser.dnunder;
set sasuser.sale2000(keep=routeid flightid date
                      revlst revbusiness revecon revcargo);
      key=flightdate;
Profit=sum(revlst, revbusiness, revecon, revcargo,
           -expenses);
run;
```
- d. 

```

data work.profit;
set sasuser.dnunder;
set sasuser.sale2000(keep=routeid flightid date
                      revlst revbusiness revecon revcargo)
      key=flightdate;
Profit=sum(revlst, revbusiness, revecon, revcargo,
           -expenses);
run;
```

10. Which of the following statements about the \_IORC\_ variable is **false**?

- a. It is automatically created when you use either a SET statement with the KEY= option or the MODIFY statement with the KEY= option in a DATA step.
- b. A value of zero for \_IORC\_ means that the most recent SET statement with the KEY= option (or MODIFY statement with the KEY= option) did not execute successfully.
- c. A value of zero for \_IORC\_ means that the most recent SET statement with the KEY= option (or MODIFY statement with the KEY= option) executed successfully.
- d. You can use the \_IORC\_ variable to prevent nonmatching data from being included when you use an index to combine data from multiple data sets.

# Chapter 4

## Using Lookup Tables to Match Data

### Introduction

Sometimes, you need to combine data from two or more sets into a single observation in a new data set according to the values of a common variable. When the data sources are two or more data sets that have a common structure, you can use a match-merge to combine the data sets. However, in some cases the data sources do not share a common structure. When data sources do not have a common structure, you can use a **lookup table** to match them. A lookup table is a table that contains key values.

Key	Var_X	Key	Var_Y	Var_Z	Year	Var_X	Var_Y	Var_Z
1998	X1	1998	Y1	Z1	1998	X1	Y1	Z1
1999	X2	1999	Y2	Z2	1999	X2	Y2	Z2
2000	X3	2000	Y3	Z3	2000	X3	Y3	Z3
2001	X4	2001	Y4	Z4	2001	X4	Y4	Z4
2002	X5	2002	Y5	Z5	2002	X5	Y5	Z5

The technique that you use to perform a table lookup is dependent on your data. This lesson focuses on using multidimensional arrays to perform table lookups and transposing SAS data sets in preparation for a match-merge.

### Objectives

In this lesson, you learn to

- use a multidimensional array to match data
- use stored array values to match data
- use PROC TRANSPOSE to transpose a SAS data set and prepare it for a table lookup
- merge a transposed SAS data set
- use a hash object as a lookup table (for SAS®9 and higher).

## Using Multidimensional Arrays

### Review of the Multidimensional Array Statement

When a lookup operation depends on more than one numerical factor, you can use a multidimensional array. As you learned in the lesson **Processing Variables with Arrays**, you use an ARRAY statement to create an array. The ARRAY statement defines a set of elements that you plan to process as a group.

#### General form, multidimensional ARRAY statement:

```
ARRAY array-name {rows,cols,...} <$> <length>
    <array-elements> <(initial values)>;
```

where

- *array-name* names the array
- *rows* specifies the number of array elements in a row arrangement
- *cols* specifies the number of array elements in a column arrangement
- *array-elements* names the variables that make up the array
- *initial values* specifies initial values for the corresponding elements in the array that are separated by commas or spaces.



The keyword `_TEMPORARY_` may be used instead of *array-elements* to avoid creating new data set variables. Only temporary array elements are produced as a result of using `_TEMPORARY_`.

When you are working with arrays, remember that

- The name of the array must be a SAS name that is not the name of a SAS variable in the same DATA step.
- The variables listed as array elements must all be the same type (either all numeric or all character).
- The initial values specified can be numbers or character strings. You must enclose all character strings in quotation marks.



If you use the `_TEMPORARY_` keyword in an array statement, remember that temporary data elements behave like DATA step variables with the following exceptions.

- They do not have names. Refer to temporary data elements by the array name and dimension.
- They do not appear in the output data set.
- You cannot use the special subscript asterisk (\*) to refer to all the elements.
- Temporary data element values are always automatically retained, rather than being reset to missing at the beginning of the next iteration of the DATA step.

#### Example

Suppose you need to determine the wind chill values for the flights represented in the SAS data set `Sasuser.Flights`. The data set contains three variables: Flight (the flight number), Temp (the average outdoor temperature during the flight), and Wspeed (the average wind speed during the flight).

SAS Data Set `Sasuser.Flights`

Obs	Flight	Temp	Wspeed
1	IA2736	-8	9
2	IA6352	-4	16

Wind chill values are derived from the air temperature and wind speed as shown in the following wind chill lookup table. To determine the wind chill for each flight, you can create a multidimensional array that stores the wind chill values shown in the table. You can then match the values of Temp and Wspeed with the wind chill values stored in the array.

Wind Speed (in miles per hour)	Temperature (in degrees Fahrenheit)									
	-10	-5	0	5	10	15	20	25	30	
	5	-22	-16	-11	-5	1	7	13	19	25
	10	-28	-22	-16	-10	-4	3	9	15	21
	15	-32	-26	-19	-13	-7	0	6	13	19
	20	-35	-29	-22	-15	-9	-2	4	11	17
	25	-37	-31	-24	-17	-11	-4	3	9	16
	30	-39	-33	-26	-19	-12	-5	1	8	15
	35	-41	-34	-27	-21	-14	-7	0	7	14
	40	-43	-36	-29	-22	-15	-8	-1	6	13

In the following program, the ARRAY statement creates the two-dimensional array WC and specifies the dimensions of the array: four rows and two columns. No variables are created from the array because the keyword \_TEMPORARY\_ is used. The initial values specified correspond to the values in the wind chill lookup table. For this example, only the values in the first two columns and four rows in the wind chill lookup table are included in the array.

```
data work.wndchill (drop = column row);
array WC{4,2} _temporary_
  (-22,-16,-28,-22,-32,-26,-35,-29);
set sasuser.flights;
row = round(wspeed,5)/5;
column = (round(temp,5)/5)+3;
WindChill= wc{row,column};
run;
```

Wind Speed (in miles per hour)	Temperature (in degrees Fahrenheit)									
	-10	-5	0	5	10	15	20	25	30	
	5	-22	-16	-11	-5	1	7	13	19	25
	10	-28	-22	-16	-10	-4	3	9	15	21
	15	-32	-26	-19	-13	-7	0	6	13	19
	20	-35	-29	-22	-15	-9	-2	4	11	17
	25	-37	-31	-24	-17	-11	-4	3	9	16
	30	-39	-33	-26	-19	-12	-5	1	8	15
	35	-41	-34	-27	-21	-14	-7	0	7	14
	40	-43	-36	-29	-22	-15	-8	-1	6	13

The value of WindChill for each flight is determined by referencing the array based on the values of Wspeed and Temp in the Sasuser.Flights data set. The row number for the array reference is determined by the value of Wspeed. The column number for the array reference is determined by the value of Temp.

```
data work.wndchill (drop = column row);
array WC{4,2} _temporary_
```

```

        (-22,-16,-28,-22,-32,-26,-35,-29);
set sasuser.flights;
row = round(wspeed,5)/5;
column = (round(temp,5)/5)+3;
WindChill= wc{row,column};
run;

```

**Table Representation of the WC Array**

-22	-16
-28	-22
-32	-26
-35	-29

The rounding unit for the value of Wspeed is 5 because the values for wind speed in the wind chill table are rounded to every 5 miles-per-hour. Wspeed is then divided by 5 to derive the row number for the array reference.

Like the value for Wspeed, the value of Temp is rounded to the nearest five, then divided by 5. The offset of 3 is added to the value because the third column in the wind chill lookup table represents 0 degrees.

```

data work.wndchill (drop = column row);
array WC{4,2} _temporary_
        (-22,-16,-28,-22,-32,-26,-35,-29);
set sasuser.flights;
row = round(wspeed,5)/5;
column = (round(temp,5)/5)+3;
WindChill= wc{row,column};
run;

```

The following page illustrates how the program is processed, step by step.

## Array Processing

1. The following program is submitted to SAS:

```

data work.wndchill (drop = column row);
array WC{4,2} _temporary_
        (-22,-16,-28,-22,-32,-26,-35,-29);
set sasuser.flights;
row = round(wspeed,5)/5;
column = (round(temp,5)/5)+3;
WindChill= wc{row,column};
run;

```

2. During the compilation phase, the WC array is populated with the values specified in the ARRAY statement. Also, SAS sets up the new data set by reading the descriptor portion of the input data set and creating the PDV. Since the \_TEMPORARY\_ keyword is specified in the ARRAY statement, there are no variables included in the PDV for array elements.
3. Execution begins. SAS reads the first observation from Sasuser.Flights and records values for Flight, Temp, and Wspeed in the PDV.
4. SAS calculates the value of row based on the value of Wspeed. The value of Wspeed is 9, which is rounded to 10 (the nearest multiple of 5) and then divided by 5, giving row a value of 2. SAS records this value in the PDV.
5. SAS calculates the value of column based on the value of Temp. The value of Temp is -8, which is rounded to -10 (the nearest multiple of 5) and then divided by 5, leaving a value of -2. The

offset value of 3 is then added to -2, giving column a value of 1. SAS records this value in the PDV.

6. SAS determines the value of WindChill by looking up the value of the array element WC{2,1}. SAS records this value, 28, in the PDV.
7. SAS writes the values from the PDV to the output data set Work.Wndchill except for the temporary variables row and column.
8. The DATA step iterates. SAS reads the second observation of the Sasuser.Flights data set and records the values in the PDV. Then, SAS calculates the values for row and column and uses these values to look up the value of WindChill.
9. SAS writes the values in the PDV to the output data set Work.Wndchill, except for the temporary variables row and column. Work.Wndchill contains four variables and two observations.

PROC PRINT output shows the completed data set.

```
proc print data=work.wndchill;
run;
```

Obs	Flight	Temp	Wspeed	WindChill
1	IA2736	-8	9	-28
2	IA6352	-4	16	-26

Write a statement to store the values in the table below in the temporary, two-dimensional array sales.

**Q1.**

Weekend Sales

	Sat	Sun
Wk 1	1621	1243
Wk 2	1740	1425

What is the value of Score, given the following array reference? Score=results{2,3};

**Q2.**

Table Representation of Results Array

98	87	85	79	91
86	92	81	83	77
96	93	88	84	89

## Using Stored Array Values

In the previous section, the wind chill values were loaded into the WC array when the array was created. In some cases, you may need to store array values in a SAS data set rather than loading them in an ARRAY statement. Array values should be stored in a SAS data set when

- there are too many values to initialize easily in the array
- the values change frequently
- the same values are used in many programs.

### Example

Suppose you want to compare the **actual** cargo revenue values in the SAS data set **Sasuser.Monthsum** to the target cargo revenue values in the SAS data set **Sasuser.Cttargets**.

**Sasuser.Monthsum** contains the actual cargo and passenger revenue figures for each month from 1997 through 1999.

**SAS Data Set Sasuser.Monthsum**  
(first five observations of selected variables)

Obs	SaleMon	RevCargo	MonthNo
1	JAN1997	\$171,520,869.10	1
2	JAN1998	\$238,786,807.60	1
3	JAN1999	\$280,350,393.00	1
4	FEB1997	\$177,671,530.40	2
5	FEB1998	\$215,959,695.50	2

The SAS data set **Sasuser.Ctargs** contains the target cargo revenue figures for each month from 1997 through 1999.

**SAS Data Set Sasuser.Ctargs**

Obs	Year	Jan	Feb	Mar	Apr	May	Jun
1	1997	192284420	86376721	28526103	260386468	109975326	102833104
2	1998	108645734	147656369	202158055	41160707	264294440	267135485
3	1999	85730444	74168740	39955768	312654811	318149340	187270927

Obs	Jul	Aug	Sep	Oct	Nov	Dec
1	196728648	236996122	112413744	125401565	72551855	136042505
2	208694865	83456868	286846554	275721406	230488351	24901752
3	123394421	34273985	151565752	141528519	178043261	181668256

You want to create a new SAS data set, **Work.Lookup1**, that lists the actual and target values for each month. Work.Lookup1 should have the same structure as **Sasuser.Monthsum**: an observation for each month and year, as well as a new variable, Ctarget (target cargo revenues). The value of Ctarget is derived from the target values in **Sasuser.Ctargs**.

**SAS Data Set Work.Lookup1**  
(first five observations of selected variables)

Obs	SaleMon	RevCargo	Ctarget
1	JAN1997	\$171,520,869.10	\$192,284,420.00
2	JAN1998	\$238,786,807.60	\$108,645,734.00
3	JAN1999	\$280,350,393.00	\$85,730,444.00
4	FEB1997	\$177,671,530.40	\$86,376,721.00
5	FEB1998	\$215,959,695.50	\$147,656,369.00

**Sasuser.Monthsum** and **Sasuser.Ctargs** cannot be merged because they have different structures:

- **Sasuser.Monthsum** has an observation for each month and year.
- **Sasuser.Ctargs** has one column for each month and one observation for each year.

However, the data sets have two common factors: month and year. You can use a multidimensional array to match the actual values for each month and year in **Sasuser.Monthsum** with the target values for each month and year in **Sasuser.Ctargs**.

## Creating an Array

The first step is to create an array to hold the values in the target data set, **Sasuser.Ctargs**. The array needs two dimensions: one for the year values and one for the month values. In the following program, the first ARRAY statement creates the two-dimensional array, Targets.

Remember that the index of an array does not have to range from one to the number of elements. You can specify a range for the values for the index when you define the array. In this case, the dimensions of the array are specified as three rows (one for each year: 1997, 1998, and 1999) and 12 columns (one for each month).

```
data work.lookup1;
array Targets{1997:1999,12} _temporary_;
if _n_=1 then do i= 1 to 3;
    set sasuser.ctargs;
    array mnth{*} Jan--Dec;
    do j=1 to dim(mnth);
        targets{year,j}=mnth{j};
    end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
year=input(substr(salemon,4),4.);
Ctarget=targets{year,monthno};
format ctarget dollar15.2;
run;
```

The following table represents the Targets array. Notice that the array is not populated. The next step is to load the array elements from **Sasuser.Ctargs**.

**Table Representation of Targets Array**

	1	2	3	4	5	6	7	8	9	10	11	12
1997												
1998												
1999												



The row dimension for the Targets array could have been specified using the value 3. For example,

```
array Targets{3,12} _temporary_;
```

However, using the notation 1997:1999 simplifies the program by eliminating the need to map numeric values to the year values.

## Loading the Array Elements

The Targets array needs to be loaded with the values in **Sasuser.Ctargs**. One method for accomplishing this task is to load the array within a DO loop.

**SAS Data Set Sasuser.Ctargs**

Year	Jan	Feb	Mar	Apr	May	Jun
------	-----	-----	-----	-----	-----	-----

Year	Jan	Feb	Mar	Apr	May	Jun
1997	192284420	86376721	28526103	260386468	109975326	102833104
1998	108645734	147656369	202158055	41160707	264294440	267135485
1999	85730444	74168740	39955768	312654811	318149340	187270927

Jul	Aug	Sep	Oct	Nov	Dec
196728648	236996122	112413744	125401565	72551855	136042505
208694865	83456868	286846554	275721406	230488351	24901752
123394421	34273985	151565752	141528519	178043261	181668256

The IF-THEN statement specifies that the Targets array is loaded only once, during the first iteration of the DATA step. The DO loop executes three times, once for each observation in **Sasuser.Ctargs**.

The ARRAY statement within the DO loop creates the Mnth array, which will be used to store the elements from **Sasuser.Ctargs**. The dimensions of the Mnth array are specified using an asterisk, which enables SAS to automatically count the array elements.



If you use an asterisk to specify the dimensions of an array, you must list the array elements. You cannot use an asterisk to specify an array's dimensions if the elements of the array are specified with the \_TEMPORARY\_ keyword.

The array elements Jan through Dec are listed using a double hyphen (--). The double hyphen (--) is used to read the specified values based on their positions in the PDV instead of alphabetically.

```
data work.lookup1;
array Targets{1997:1999,12} _temporary_;
if _n_=1 then do i= 1 to 3;
  set sasuser.ctargs;
  array Mnth{*} Jan--Dec;
  do j=1 to dim(mnth);
    targets{year,j}=mnth{j};
  end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
year=input(substr(salemon,4),4.);
Ctarget=targets{year,monthno};
format ctarget dollar15.2;
run;
```

The following table shows the values in the Mnth array after the first iteration of the DO loop.

**Table Representation of Mnth Array  
(after the first iteration of the DO loop)**

Jan	Feb	Mar...	...Oct	Nov	Dec
192284420	86376721	260386468	125401565	72551855	136042505

Within the nested DO loop, the Targets array reference is matched to the Mnth array reference in order to populate the Targets array. The DIM function returns the number of elements in the Mnth array (in this case 12) and provides an ending point for the nested DO loop.

```

data work.lookup1;
array Targets{1997:1999,12} _temporary_;
if _n_=1 then do i= 1 to 3;
  set sasuser.ctargets;
  array Mnth{*} Jan--Dec;
  do j=1 to dim(mnth);
    targets{year,j}=mnth{j};
  end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
year=input(substr(salemon,4),4.);
Ctarget=targets{year,monthno};
format ctarget dollar15.2;
run;

```

**Table Representation of Mnth Array**  
(after the second iteration of the DO loop)

Jan	Feb	Mar...	...Oct	Nov	Dec
108645734	147656369	202158055	275721406	230488351	24901752

**Table Representation of Mnth Array**  
(after the third iteration of the DO loop)

Jan	Feb	Mar...	...Oct	Nov	Dec
85730444	74168740	39955768	141528519	178043261	181668256

**Table Representation of Populated Targets Array**

	1	2	3...	...10	11	12
1997	192284420	86376721	260386468	125401565	72551855	136042505
1998	108645734	147656369	202158055	275721406	230488351	24901752
1999	85730444	74168740	39955768	141528519	178043261	181668256



The dimension of the Mnth array could also be specified using the numeric value 12. However, the asterisk notation enables the program to be more flexible. For example, using the asterisk, the program would not need to be edited if the target data set contained data for only eleven months. Remember that if you use an asterisk to count the array elements, you must list the array elements.

## Reading the Actual Values

The last step is to read the actual values stored in **Sasuser.Monthsum**. Remember that you need to know the month and year values for each observation in order to locate the correct target revenue values.

**SAS Data Set Sasuser.Monthsum**  
(first five observations of selected variables)

SaleMon	RevCargo	MonthNo
JAN1997	\$171,520,869.10	1
JAN1998	\$238,786,807.60	1

SaleMon	RevCargo	MonthNo
JAN1999	\$280,350,393.00	1
FEB1997	\$177,671,530.40	2
FEB1998	\$215,959,695.50	2

The values for month are read in from MonthNo. The year values are contained within the values of SaleMon and can be extracted using the SUBSTR function. In this example, the SUBSTR function brings in four characters from SaleMon, starting at the fourth character. Note that the INPUT function is used to convert the value that is extracted from SaleMon from character to numeric in the assignment statement for Year. A numeric format must be used because the value of Year will be used as an array reference.

The values of Ctarget are then read in from the Targets array based on the value of Year and MonthNo.

```
data work.lookup1;
array Targets{1997:1999,12} _temporary_;
if _n_=1 then do i= 1 to 3;
  set sasuser.ctargets;
  array Mnth{*} Jan--Dec;
  do j=1 to dim(mnth);
    targets{year,j}=mnth{j};
  end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
year=input(substr(salemon,4),4.);
Ctarget=targets{year,monthno};
format ctarger dollar15.2;
run;
```

Table Representation of Targets Array

	1	2	3...	...10	11	12
1997	192284420	86376721	260386468	125401565	72551855	136042505
1998	108645734	147656369	202158055	275721406	230488351	24901752
1999	85730444	74168740	39955768	141528519	178043261	181668256

PROC PRINT output shows the new data set **Work.Lookup1**, which contains the actual cargo values (RevCargo) and the target cargo values (Ctarget).

```
proc print data=work.lookup1;
var salemon revcargo ctarget;
run;
```

Work.Lookup1 (first ten observations)

Obs	SaleMon	RevCargo	Ctarget
1	JAN1997	\$171,520,869.10	\$192,284,420.00
2	JAN1998	\$238,786,807.60	\$108,645,734.00
3	JAN1999	\$280,350,393.00	\$85,730,444.00
4	FEB1997	\$177,671,530.40	\$86,376,721.00
5	FEB1998	\$215,959,695.50	\$147,656,369.00
6	FEB1999	\$253,999,924.00	\$74,168,740.00
7	MAR1997	\$196,591,378.20	\$28,526,103.00
8	MAR1998	\$239,056,025.55	\$202,158,055.00

Obs	SaleMon	RevCargo	Ctarget
9	MAR1999	\$281,433,310.00	\$39,955,768.00
10	APR1997	\$380,804,120.20	\$260,386,468.00

The following illustrates how the program is processed, step by step.

## Array Processing

The following steps shows how stored array values are used to match the values for each month and year in **Sasuser.Monthsum** with the values for each month and year in **Sasuser.Ctargets**.

1. The following program is submitted to SAS:

```
data sasuser.lookup1 (drop=i j);
array Targets{1997:1999, 12} _temporary_;
if _N_=1 then do i=1 to 3;
  set sasuser.ctargets;
  array Mnth{*} Jan--Dec;
  do j=1 to dim(Mnth);
    targets{year, j}=mnth{j};
  end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
Year=input(substr(salemon,4),4.);
Ctargets=targets{year,monthno};
format ctarget dollar15.2;
run;
```

2. During the compilation phase, SAS creates the Targets and Mnth arrays. Also, SAS sets up the new data set by reading the descriptor portion of the input data sets and creating the PDV.
3. Execution begins. During the first iteration of the DATA step, the value of `_N_` is 1. Therefore the DO loops executes three times: once for each year in the target data set, **Sasuser.Ctargets**.
4. SAS reads the first observation in **Sasuser.Ctargets** and records the values in the Mnth array.
5. Within the nested DO loop, the Targets array reference is matched with the Mnth array reference in order to populate the Targets array. Year equals 1997 and j equals 1. So, Targets{1997,1} is assigned the value of Mnth{1}.
6. The values for the 11 remaining months in 1997 are assigned to the Targets array as the inner DO loop iterated 11 more times.
7. After the value for Mnth{12} is read into the Targets array, control returns to the top of the outer DO loop. The values for the second observation (year) are read from **Sasuser.Ctargets**. The process continues until the values for all three observations (years) are read and assigned to the Targets array.
8. After the third iteration of the outer DO loop, control drops down to the second SET statement and the values from the first observation of **Sasuser.Monthsum** are read into the PDV.
9. The new value of Year is extracted from the value of Salemon using the SUBSTR function and is converted to a numeric value by the INPUT function.
10. The value of Ctarget is pulled from the Targets array. For the first observation, the value of Ctarget equals the value of Targets{1997,1}, which is 198284420.
11. The values in the PDV are written to the output data set, **Sasuser.Lookup1**, as the first observation and control returns to the top of the DATA step.
12. During the second iteration of the DATA step, the condition `_N_=1` is false, so the Targets array is not populated again. Control drops to the second SET statement, which reads the second

observation from **Sasuser.Monthsum**, which is processed as before and is written to **Sasuser.Lookup1** as the second observation.

13. The program continues to execute until all observations from **Sasuser.Monthsum** have been read.

If the same array values are used in many programs, it is best to  
**Q.3.** store the array values in a SAS data set.  
initialize the values in an array within each program.

## Using PROC TRANSPOSE

In the previous section, we compared actual revenue values to target revenue values using an array as a lookup table. Remember that

- **Sasuser.Monthsum** has an observation for each month and year.

**SAS Data Set Sasuser.Monthsum**  
(first five observations of selected variables)

SaleMon	RevCargo	MonthNo
JAN1997	\$171,520,869.10	1
JAN1998	\$238,786,807.60	1
JAN1999	\$280,350,393.00	1
FEB1997	\$177,671,530.40	2
FEB1998	\$215,959,695.50	2

- **Sasuser.Ctargs** has one variable for each month and one observation for each year.

**SAS Data Set Sasuser.Ctargs**  
(selected variables)

Year	Jan	Feb	Mar	Apr	May	Jun
1997	192284420	86376721	28526103	260386468	109975326	102833104
1998	108645734	147656369	202158055	41160707	264294440	267135485
1999	85730444	74168740	39955768	312654811	318149340	187270927

Using arrays was a good solution because the orientation of the data sets differed. An alternate solution is to transpose **Sasuser.Ctargs** using PROC TRANSPOSE, and then merge the transposed data set with **Sasuser.Monthsum** by the values of Year and Month.

### General form, PROC TRANSPOSE:

```
PROC TRANSPOSE <DATA=input-data-set>
  <OUT=output-data-set>
  <NAME=variable-name>
  <PREFIX=variable-name>;
  BY <DESCENDING> variable-1
    <...<DESCENDING> variable-n>
    <NOTSORTED>;
  VAR variable(s);
  RUN;
```

where

- **DATA=***input-data-set* names the SAS data set to transpose.
- **OUT=***output-data-set* names the output data set.
- **NAME=***variable-name* specifies the name for the variable in the output data set that contains the name of the variable that is being transposed to create the current observation.
- **PREFIX=***variable-name* specifies a prefix to use in constructing names for transposed variables in the output data set. For example, if PREFIX=VAR, the names of the variables are VAR1, VAR2, ..., VARn.
- the **BY** statement is used to transpose each BY group
- **VAR** *variable(s)* names one or more variables to transpose.



If *output-data-set* does not exist, PROC TRANSPOSE creates it by using the DATA*n* naming convention.



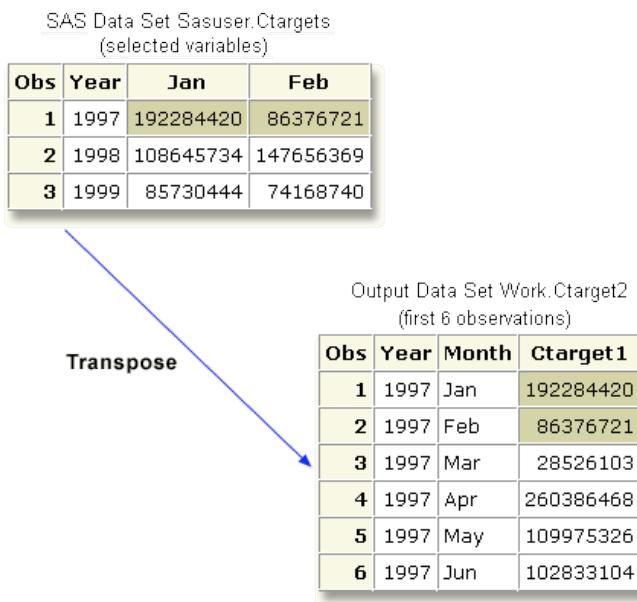
If you omit the **VAR** statement, the TRANSPOSE procedure transposes all of the numeric variables in the input data set that are not listed in another statement.



You must list character variables in a **VAR** statement if you want to transpose them.

The TRANSPOSE procedure creates an output data set by restructuring the values in a SAS data set. When the data set is restructured, selected variables are transposed into observations. The TRANSPOSE procedure can often eliminate the need to write a lengthy DATA step to achieve the same result. The output data set can be used in subsequent DATA or PROC steps for analysis, reporting, or further data manipulation.

PROC TRANSPOSE does not print the output data set. Use PROC PRINT, PROC REPORT, or some other SAS reporting tool to print the output data set.



### Example

The following program transposes the SAS data set **Sasuser.Cttargets**. The OUT= option specifies the name of the output data set, **Work.Cttarget2**. All of the variables in **Sasuser.Cttargets** are transposed because all of the variables are numeric and a VAR statement is not used in the program.

```
proc transpose data=sasuser.cttargets
```

```

out=work.ctarget2;
run;

```

**Input Data Set: Sasuser.Ctargets**  
(selected variables)

Year	Jan	Feb	Mar	Apr	May	Jun
1997	192284420	86376721	28526103	260386468	109975326	102833104
1998	108645734	147656369	202158055	41160707	264294440	267135485
1999	85730444	74168740	39955768	312654811	318149340	187270927

**Output Data Set: Work.Ctarget2**

Obs	_NAME_	COL1	COL2	COL3
1	Year	1997	1998	1999
2	Jan	192284420	108645734	85730444
3	Feb	86376721	147656369	74168740
4	Mar	28526103	202158055	39955768
5	Apr	260386468	41160707	312654811
6	May	109975326	264294440	318149340
7	Jun	102833104	267135485	187270927
8	Jul	196728648	208694865	123394421
9	Aug	236996122	83456868	34273985
10	Sep	112413744	286846554	151565752
11	Oct	125401565	275721406	141528519
12	Nov	72551855	230488351	178043261
13	Dec	136042505	24901752	181668256

Notice that in the output data set, the variables are named `_NAME_`, `COL1`, `COL2`, and `COL3`.

`_NAME_` is the default name of the variable that PROC TRANSPOSE creates to identify the source of the values in each observation in the output data set. This variable is a character variable whose values are the names of the variables that are transposed from the input data set. For example, in **Work.Ctarget2** the values in the first observation in the output data set come from the values of the variable `Year` in the input data set.

The remaining transposed variables are named `COL1`...`COLn` by default. In **Work.Ctarget2**, the values of the variables `COL1`, `COL2`, and `COL3` represent the target cargo revenue for each month in the years 1997, 1998, and 1999.

### Adding Descriptive Variable Names

You can use PROC TRANSPOSE options to give the variables in the output data set more descriptive names. The **NAME=** option specifies a name for the `_NAME_` variable.

The **PREFIX=** option specifies a prefix to use in constructing names for transposed variables in the output data set. For example, if `PREFIX=Ctarget`, the names of the variables are `Ctarget1`, `Ctarget2`, and `Ctarget3`.

```

proc transpose data=sasuser.ctargets
  out=work.ctarget2

```

```

name=Month
prefix=Ctarget;
run;

```

Output Data Set: Work.Ctarget2				
Obs	Month	Ctarget1	Ctarget2	Ctarget3
1	Year	1997	1998	1999
2	Jan	192284420	108645734	85730444
3	Feb	86376721	147656369	74168740
4	Mar	28526103	202158055	39955768
5	Apr	260386468	41160707	312654811
6	May	109975326	264294440	318149340
7	Jun	102833104	267135485	187270927
8	Jul	196728648	208694865	123394421
9	Aug	236996122	83456868	34273985
10	Sep	112413744	286846554	151565752
11	Oct	125401565	275721406	141528519
12	Nov	72551855	230488351	178043261
13	Dec	136042505	24901752	181668256



The RENAME=data set option can also be used with PROC TRANSPOSE to change variable names.

```

proc transpose data=sasuser.ctargets
  out=work.ctarget2 (rename=(col1=Ctarget1 col2=Ctarget2 col3=Ctarget3))
  name=Month;
run;

```



The default label for the \_NAME\_ variable is NAME OF FORMER VARIABLE. To see this, print the transposed data set using PROC PRINT with the LABEL option. You can use a LABEL statement to override the default label.

```

proc transpose data=sasuser.ctargets
  out=work.ctarget2
  name=Month
  prefix=Ctarget;
run;
proc print data=work.ctarget2 label;
label Month=MONTH;
run;

```

Write a statement to transpose the SAS data set **Work.Sales**. The output data set should be named **Work.Sales2**. Both data sets are shown below.

SAS Data Set **Work.Sales**

**Q.4.**

Obs	Quarter	Month1	Month2	Month3
1	1	8293.56	7986.54	8173.98
2	2	8173.89	8056.82	7954.87
3	3	8364.27	8124.36	7892.81

4	4	8422.9	8255.67	8385.34
---	---	--------	---------	---------

SAS Data Set Work.Sales2

Obs	_NAME_	COL1	COL2	COL3	COL4
1	Quarter	1	2	3	4
2	Month1	8293.56	8173.89	8364.27	8422.9
3	Month2	7986.54	8056.82	8124.36	8255.67
4	Month3	8173.98	7954.87	7892.81	8385.34

---

run;

Add an option to specify the name Meal for the \_NAME\_ variable.

**Q.5.** proc transpose data=flight.service  
out=flight.service2

---

run;

## Merging the Transposed Data Set

### Structuring the Data for a Merge

Remember that the transposed data set, **Work.Cttarget2**, needs to be merged with **Sasuser.Monthsum** by the values of Year and Month. Neither data set is currently structured correctly for the merge.

SAS Data Set Work.Cttarget2  
(first five observations)

Obs	Month	Ctarget1	Ctarget2	Ctarget3
1	Year	1997	1998	1999
2	Jan	192284420	108645734	85730444
3	Feb	86376721	147656369	74168740
4	Mar	28526103	202158055	39955768
5	Apr	260386468	41160707	312654811

SAS Data Set Sasuser.Monthsum  
(first five observations of selected variables)

Obs	SaleMon	RevCargo	MonthNo
1	JAN1997	\$171,520,869.10	1
2	JAN1998	\$238,786,807.60	1
3	JAN1999	\$280,350,393.00	1
4	FEB1997	\$177,671,530.40	2
5	FEB1998	\$215,959,695.50	2

### Using a BY Statement with PROC TRANSPOSE

In order to correctly structure **Work.Cttarget2** for the merge, a BY statement needs to be used with PROC TRANSPOSE. For each BY group, PROC TRANSPOSE creates one observation for each variable that it transposes. The BY variable itself is not transposed.

The following program transposes **Sasuser.Cttargets** by the value of Year. The resulting output data set, **Work.Cttarget2**, now contains twelve observations for each each year (1997, 1998, and 1999).

```
proc transpose data=sasuser.cttargets
  out=work.cttarget2
  name=Month
  prefix=Ctarget;
  by year;
run;
```

**Input Data Set Sasuser.Cttargets**  
(selected variables)

Obs	Year	Jan	Feb	Mar	Apr	May	Jun
1	1997	192284420	86376721	28526103	260386468	109975326	102833104
2	1998	108645734	147656369	202158055	41160707	264294440	267135485
3	1999	85730444	74168740	39955768	312654811	318149340	187270927

**Output Data Set Work.Cttarget2**  
(first 12 observations)

Obs	Year	Month	Ctarget1
1	1997	Jan	192284420
2	1997	Feb	86376721
3	1997	Mar	28526103
4	1997	Apr	260386468
5	1997	May	109975326
6	1997	Jun	102833104
7	1997	Jul	196728648
8	1997	Aug	236996122
9	1997	Sep	112413744
10	1997	Oct	125401565
11	1997	Nov	72551855
12	1997	Dec	136042505



The original SAS data set must be sorted or indexed prior to using a BY statement with PROC TRANSPOSE unless you use the NOTSORTED option.

### Sorting the Work.Cttarget2 Data Set

The last step in preparing **Work.Cttarget2** for the merge is to use the SORT procedure to sort the data set by Year and Month as shown in the following program:

```
proc sort data=work.cttarget2;
  by year month;
run;
```

Notice that in the sorted version of **Work.Cttarget2**, the values of month are sorted alphabetically by year.

### SAS Data Set Work.Ctarget2

(sorted, first 12 observations)

Obs	Year	Month	Ctarget1
1	1997	Apr	260386468
2	1997	Aug	236996122
3	1997	Dec	136042505
4	1997	Feb	86376721
5	1997	Jan	192284420
6	1997	Jul	196728648
7	1997	Jun	102833104
8	1997	Mar	28526103
9	1997	May	109975326
10	1997	Nov	72551855
11	1997	Oct	125401565
12	1997	Sep	112413744

### Reorganizing the Sasuser.Monthsum Data Set

The data in **Sasuser.Monthsum** must also be reorganized for the merge because the month and year values in that data set are combined in the variable SaleMon.

### SAS Data Set Sasuser.Monthsum

(first five observations of selected variables)

Obs	SaleMon	RevCargo	MonthNo
1	JAN1997	\$171,520,869.10	1
2	JAN1998	\$238,786,807.60	1
3	JAN1999	\$280,350,393.00	1
4	FEB1997	\$177,671,530.40	2
5	FEB1998	\$215,959,695.50	2

The following program creates two new variables, Year and Month, to hold the year and month values. The values for Year are created from MonthSum using the INPUT and SUBSTR functions. The values for Month are extracted from MonthSum using the LOWCASE and SUBSTR functions.

```
data work.mnthsum2;
  set sasuser.monthsum(keep=SaleMon RevCargo);
  length Month $ 8;
  Year=input(substr(SaleMon,4),4.);
  Month=substr(SaleMon,1,1)
    || lowcase(substr(SaleMon,2,2));
run;
```

### SAS Data Set Work.Mnthsum2

(first six observations)

Obs	SaleMon	RevCargo	Month	Year
1	JAN1997	\$171,520,869.10	Jan	1997
2	JAN1998	\$238,786,807.60	Jan	1998
3	JAN1999	\$280,350,393.00	Jan	1999

Obs	SaleMon	RevCargo	Month	Year
4	FEB1997	\$177,671,530.40	Feb	1997
5	FEB1998	\$215,959,695.50	Feb	1998
6	FEB1999	\$253,999,924.00	Feb	1999

### Sorting the Work.Mnthsum2 Data Set

As with **Work.Cttarget2**, the last step in preparing for the merge is to sort the data set by the values of Year and Month as shown in the following program:

```
proc sort data=work.mnthsum2;
  by year month;
run;
```

Notice that in the sorted version of **Work.Mnthsum2**, the values of month are sorted alphabetically by year.

**SAS Data Set Work.Mnthsum2**  
(sorted, first twelve observations)

Obs	SaleMon	RevCargo	Month	Year
1	APR1997	\$380,804,120.20	Apr	1997
2	AUG1997	\$196,639,501.10	Aug	1997
3	DEC1997	\$196,504,413.00	Dec	1997
4	FEB1997	\$177,671,530.40	Feb	1997
5	JAN1997	\$171,520,869.10	Jan	1997
6	JUL1997	\$197,163,278.20	Jul	1997
7	JUN1997	\$190,560,828.50	Jun	1997
8	MAR1997	\$196,591,378.20	Mar	1997
9	MAY1997	\$196,261,573.20	May	1997
10	NOV1997	\$190,228,066.70	Nov	1997
11	OCT1997	\$196,957,153.40	Oct	1997
12	SEP1997	\$190,535,012.50	Sep	1997

### Completing the Merge

When the data is structured correctly, **Work.Mnthsum2** and **Work.Cttarget2** can be merged by the values of Year and Month as shown in the following program:

```
data work.merged;
  merge work.mnthsum2 work.cttarget2;
  by year month;
run;
```

**SAS Data Set Work.Mnthsum2**  
(first five observations)

Obs	SaleMon	RevCargo	Month	Year
1	APR1997	\$380,804,120.20	Apr	1997
2	AUG1997	\$196,639,501.10	Aug	1997

Obs	SaleMon	RevCargo	Month	Year
3	DEC1997	\$196,504,413.00	Dec	1997
4	FEB1997	\$177,671,530.40	Feb	1997
5	JAN1997	\$171,520,869.10	Jan	1997

**SAS Data Set Work.Ctarget2**  
(first five observations)

Obs	Year	Month	Ctarget1
1	1997	Apr	260386468
2	1997	Aug	236996122
3	1997	Dec	136042505
4	1997	Feb	86376721
5	1997	Jan	192284420

PROC PRINT output shows the resulting data set **Work.Merged**. The values of RevCargo represent the actual cargo revenue for each month. The values of Ctarget1 represent the target cargo values for each month.

```
proc print data=work.merged;
  format ctarget1 dollar15.2;
  var month year revcargo ctarget1;
run;
```

**SAS Data Set Work.Merged (partial output)**

Obs	Month	Year	RevCargo	CTarget1
1	Apr	1997	\$380,804,120.20	\$260,386,468.00
2	Aug	1997	\$196,639,501.10	\$236,996,122.00
3	Dec	1997	\$196,504,413.00	\$136,042,505.00
4	Feb	1997	\$177,671,530.40	\$86,376,721.00
5	Jan	1997	\$171,520,869.10	\$192,284,420.00
6	Jul	1997	\$197,163,278.20	\$196,728,648.00
7	Jun	1997	\$190,560,828.50	\$102,833,104.00
8	Mar	1997	\$196,591,378.20	\$28,526,103.00
9	May	1997	\$196,261,573.20	\$109,975,326.00
10	Nov	1997	\$190,228,066.70	\$72,551,855.00

Add a statement to group the observations in the output data set by Date.

**Q.6.**

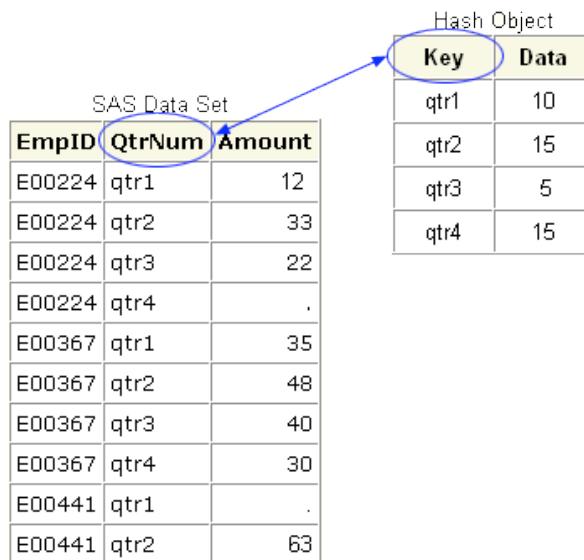
```
proc transpose data=flight.service
  out=flight.service2
  name=Meal
  prefix=Flight;
  _____
run;
```

## Using Hash Objects as Lookup Tables

Beginning with SAS®9, the hash object is available for use in a DATA step. The hash object provides an efficient, convenient mechanism for quick data storage and retrieval.

Unlike an array, which uses a series of consecutive integers to address array elements, a hash object can use any combination of numeric and character values as addresses. A hash object can be loaded from hard-coded values or a SAS data set, is sized dynamically, and exists for the duration of the DATA step.

The hash object is a good choice for lookups using unordered data that can fit into memory because it provides in-memory storage and retrieval and does not require the data to be sorted or indexed.



### The Structure of a Hash Object

When a lookup operation depends on one or more key values, you can use the hash object. A hash object resembles a table with rows and columns and contains a key component and a data component.

#### The key component

- might consist of numeric and character values
- maps key values to data rows
- must be unique
- can be a composite.

#### The data component

- can contain multiple data values per key value
- can consist of numeric and character values.

### Example

Suppose you have a data set, named **Sasuser.Contrib**, that lists the quarterly contributions to a retirement fund. You can use the hash object to calculate the difference between the actual contribution and the goal amount.

SAS Data Set Sasuser.Contrib  
(Partial Listing)

EmpID	QtrNum	Amount
E00224	qtr1	12
E00224	qtr2	33
E00224	qtr3	22
E00224	qtr4	.
E00367	qtr1	35
E00367	qtr2	48
E00367	qtr3	40
E00367	qtr4	30
E00441	qtr1	.
E00441	qtr2	63

Hash Object

Key: QtrNum	Data: Goal Amount
qtr1	10
qtr2	15
qtr3	5
qtr4	15

actual contribution  
 - goal amount  
calculate the difference

The following program creates a hash object that stores the quarterly goal for employee contributions to the retirement fund. To calculate the difference between actual contribution and the goal amount, the program retrieves the goal amount from the hash object based on the key values.

```

data work.difference (drop= goalamount);
length goalamount 8;
if _N_ = 1 then do;
  declare hash goal();
  goal.definekey("QtrNum");
  goal.definedata("GoalAmount");
  goal.definedone();
  call missing(qtrnum, goalamount);
  goal.add(key:'qtr1', data:10 );
  goal.add(key:'qtr2', data:15 );
  goal.add(key:'qtr3', data: 5 );
  goal.add(key:'qtr4', data:15 );
end;
set sasuser.contrib;
goal.find();
Diff = amount - goalamount;
run;

```

Let's see how the hash object is set up.

### Data Step Component Objects

The hash object is a DATA step component option. Component objects are data elements that consist of attributes and methods. **Attributes** are the properties that specify the information that is associated with an object. An example is size. **Methods** define the operations that an object can perform.

To use a DATA step component object in your SAS program, you must first declare and create (instantiate) the object.

### Declaring the Hash Object

You declare a hash object using the DECLARE statement.

### General form, DECLARE statement:

**DECLARE** *object variable* <(<*argument\_tag-1: value-1*<, ...*argument\_tag-n: value-n*>>)>;

where

1. *object* specifies the component object
2. *variable* specifies the variable name for the component object
3. *arg\_tag* specifies the information that is used to create an instance of the component object
4. *value* specifies the value for an argument tag. Valid values depend on the component object.
5. Valid values for *object* are as follows:
6. *hash* indicates a hash object.
7. *hiter* indicates a hash iterator object.



The hash iterator object retrieves data from the hash object in ascending or descending key order.

The following DECLARE statement creates a hash object named **Goal**.

```
data work.difference (drop= goalamount);
length goalamount 8;
if _N_ = 1 then do;
  declare hash goal;
```

At this point, you have only declared the variable **Goal**. It has the potential to hold a component object of the type hash.



The DECLARE statement is an executable statement.

### Instantiating the Hash Object

You use the **\_NEW\_** statement to instantiate the hash object.

### General form, \_NEW\_ statement

*variable* = **\_NEW\_** *object*(<*argument\_tag-1: value-1*<, ...*argument\_tag-n: value-n*>>);

where

- *variable* specifies the variable name for the component object
- *object* specifies the component object
- *argument\_tag* specifies the information that is used to create an instance of the component object
- *value* specifies the value for an argument tag. Valid values depend on the component object.
- Valid values for *object* are as follows:
- *hash* indicates a has object.
- *hiter* indicates a hash iterator object.

The following **\_NEW\_** statement creates an instance of the hash object and assigns it to the variable **Goal**.

```
data work.difference (drop= goalamount);
length goalamount 8;
if _N_ = 1 then do;
  declare hash goal;
  goal= _new_ hash();
```

## Declaring and Instantiating the Hash Object in a Single Step

As an alternative to the two-step process of using the DECLARE and \_NEW\_ statements to declare and instantiate a component object, you can use the DECLARE statement to declare and instantiate the component object in one step.

```
data Work.Difference (drop= goalamount);
length goalamount 8;
if _N_ = 1 then do;
  declare hash Goal();
```

## Defining Keys and Data

Remember that the hash object uses lookup keys to store and retrieve data. The keys and the data are DATA step variables that you use to initialize the hash object by using dot notation method calls.

### General form, dot notation method calls:

*object.method(<argument\_tag-1: value-1<, ...argument\_tag-n: value-n>>);*

where

- *object* specifies the variable name for the DATA step component object
- *method* specifies the name of the method to invoke
- *argument-tag* identifies the arguments that are passed to the method
- *value* specifies the argument value.

A key is defined by passing the key variable name to the DEFINEKEY method. Data is defined by passing the data variable name to the DEFINEDATA method. When all key and data variables are defined, the DEFINEDONE method is called. Keys and data can consist of any number of character or numeric data step variables.

The following code initializes the key variable QtrNum and the data variable GoalAmount.

```
data work.difference (drop= goalamount);
length goalamount 8;
if _N_ = 1 then do;
  declare hash goal();
  goal.definekey ("QtrNum");
  goal.definedata ("GoalAmount");
  goal.definedone();
```

## Using the Call Missing Routine

The hash object does not assign values to key variables, and the SAS compiler cannot detect the implicit key and data variable assignments done by the hash object. Therefore, if no explicit assignment to a key or data variable appears in the program, SAS issues a note stating that the variables are uninitialized.

To avoid receiving these notes, n use the CALL MISSING routine with the key and data variables as parameters. The CALL MISSING routine assigns a missing value to the specified character or numeric variables.

```
data Work.Difference (drop= goalamount);
length GoalAmount 8;
if _N_ = 1 then do; declare hash goal();
  goal.definekey("QtrNum");
  goal.definedata("GoalAmount");
  goal.definedone();
call missing(qtrnum, goalamount);
```



Another way to avoid receiving notes stating that the variables are uninitialized is to provide an initial assignment statement that assigns a missing value to each key and data variable.

## Loading Key and Data Values

So far, you've declared and instantiated the hash object, and initialized the hash object's key and data variables. You are now ready to store data in the hash object using the ADD method. The following code uses the ADD method to load the key values *qtr1*, *qtr2*, *qtr3*, and *qtr4* and the corresponding data values 10, 15, 5, and 15 into the hash object.

```
data work.difference (drop= goalamount);
length goalamount 8;
if _N_ = 1 then do; declare hash goal();
  declare hash goal();
  goal.definekey("QtrNum");
  goal.definedata("GoalAmount");
  goal.definedone();
  call missing(qtrnum, goalamount);
  goal.add(key:'qtr1', data:10);
  goal.add(key:'qtr2', data:15);
  goal.add(key:'qtr3', data: 5);
  goal.add(key:'qtr4', data:15);
end;
```

## Retrieving Matching Data

You use the FIND method to retrieve matching data from the hash object. The FIND method returns a value that indicates whether the key is in the hash object. If the key is in the hash object, then the FIND method also sets the data variable to the value of the data item so that it is available for use after the method call.

```
data work.difference (drop= goalamount);
length goalamount 8;
if _N_ = 1 then do;
  declare hash goal();
  goal.definekey("QtrNum");
  goal.definedata("GoalAmount");
  goal.definedone();
  call missing(qtrnum, goalamount);
  goal.add(key:'qtr1', data:10);
  goal.add(key:'qtr2', data:15);
```

```

goal.add(key:'qtr3', data: 5 );
goal.add(key:'qtr4', data:15 );
end;
set sasuser.contrib;
goal.find();
Diff = amount - goalamount;
run;

```

## Hash Object Processing

Let's take a closer look at what happens when the program is submitted for execution.

```

data Work.Difference (drop= goalamount);
length goalamount 8;
if _N_ = 1 then do;
  declare hash goal();
  goal.definekey("QtrNum");
  goal.definedata("GoalAmount");
  goal.definedone();
  call missing(qtrnum, goalamount);
  goal.add(key:'qtr1', data:10 );
  goal.add(key:'qtr2', data:15 );
  goal.add(key:'qtr3', data: 5 );
  goal.add(key:'qtr4', data:15 );
end;
set sasuser.contrib;
goal.find();
Diff = amount - goalamount;
run;

```

The following steps show the execution of a DATA step that uses a hash object as a lookup table.

1. During the first iteration of the data step, the value of `_N_` equals 1, so the DO loop begins.
2. The hash object is declared and instantiated. A key is defined by passing the key variable name `QtrNum` to the `DEFINEKEY` method. Data is defined by passing the data variable name `GoalAmount` to the `DEFINEDATA` method.
3. The `ADD` method loads the key values `qtr1`, `qtr2`, `qtr3`, and `qtr4` and the corresponding data values 10, 15, 5, and 15 into the hash object.
4. The values from the first observation in the input data set are loaded into the PDV.
5. The `FIND` method retrieves matching data from the hash object based on the value of `QtrNum`.
6. The difference between the actual contribution and the goal amount is calculated by subtracting the value of `GoalAmount` from the value of `Amount`.
7. Control returns to the top of the DO loop. At the start of the second iteration, the value of `_N_` equals 2, so the DO loop is exited and the values for the second observation in the input data set are loaded into the PDV.
8. The `FIND` method retrieves matching data from the hash object based on the value of `QtrNum`.
9. The difference between the actual contribution and the goal amount is calculated by subtracting the value of `GoalAmount` from the value of `Amount`.

The program continues to execute until the DATA step encounters the end of the line. PROC PRINT output shows the completed data set.

```

proc print data=work.difference;
run;

```

Partial Output				
Obs	EmpID	QtrNum	Amount	Diff
1	E00224	qtr1	12	2
2	E00224	qtr2	33	18
3	E00224	qtr3	22	17
4	E00224	qtr4	.	.
5	E00367	qtr1	35	25
6	E00367	qtr2	48	33
7	E00367	qtr3	40	35
8	E00367	qtr4	30	20
9	E00441	qtr1	.	.
10	E00441	qtr2	63	48

Which statement or set of statements declares and instantiates the hash object **Discount**?

**Q.7.**

```
declare discount hash();
discount= _new_ hash();
```

```
declare hash discount;
discount= _new_ hash();
```

Given the following program, what is the value of SalePrice if the value of Price is 100 and the value of Week is week3?

**Q.8.**

```
data work.fallsale;
length Discount 8;
if _N_=1 then do;
  declare hash sale();
  sale.definekey("Week");
  sale.definedata("Discount");
  sale.definedone();
  call missing(discount);
  sale.add(key:'week1', data:10);
  sale.add(key:'week2', data:15);
  sale.add(key:'week3', data:20);
  sale.add(key:'week4', data:25);
end;
set sasuser.contrib;
goal.find();
SalePrice = price - discount;
run;
```

20  
data:20  
80

## Creating a Hash Object From a SAS Data Set

Suppose you need to create a report that shows revenue, expenses, profits, and airport information. You have two data sets that contain portions of the required data. The SAS data set **Sasuser.Revenue**

contains flight revenue information. The SAS data set **Sasuser.Acities** contains airport data including the airport code, location, and name.

**SAS Data Set Sasuser.Revenue (first five observations)**

Origin	Dest	FlightID	Date	Rev1st	RevBusiness	RevEcon
ANC	RDU	IA03400	02DEC1999	15829	28420	68688
ANC	RDU	IA03400	14DEC1999	20146	26460	72981
ANC	RDU	IA03400	26DEC1999	20146	23520	59625
ANC	RDU	IA03401	09DEC1999	15829	22540	58671
ANC	RDU	IA03401	21DEC1999	20146	22540	65826

**SAS Data Set Sasuser.Acities (first five observations)**

City	Code	Name	Country
Auckland	AKL	International	New Zealand
Amsterdam	AMS	Schiphol	Netherlands
Anchorage, AK	ANC	Anchorage International Airport	USA
Stockholm	ARN	Arlanda	Sweden
Athens (Athinai)	ATH	Hellenikon International Airport	Greece

To create the report, you can use a hash object to retrieve matching airport data from **Sasuser.Acities**.

## Using a Non-Executing SET Statement

To initialize the attributes of hash variables that originate from an existing SAS data set, you can use a non-executing SET statement.

Because the IF condition is false during execution, the SET statement is compiled but not executed. The PDV is created with the variable Code, City, and Name from **Sasuser.Acities**.

```
data work.report;
if _N_=1 then do;
  if 0 then
    set sasuser.acities (keep=Code City Name);
```

When you use this technique, the MISSING routine is not required.

## Working with Multiple Data Variables

The hash object that you worked with earlier in this lesson contains one key variable and one data variable. In this example, you need to associate more than one data value with each key.

In the following code, the DECLARE statement creates the **Airports** hash object and loads it from **Sasuser.Acities**. The DEFINEKEY method call defines the key to be the value of the variable Code. The DEFINEDATA method call defines the data to be the values of the variables City and Name.

```
data work.report;
if 0 then
  set sasuser.acities (keep=Code City Name);
if _N_=1 then do;
```

```

declare hash airports (dataset: "sasuser.acities")
airports.definekey ("Code");
airports.definedata ("City", "Name");
airports.definedone();
end;

```

**Hash Object Airports**

<b>Key: Code</b>	<b>Data: City</b>	<b>Data: Name</b>
ANC	Anchorage, AK	Anchorage International Airport
BNA	Nashville, TN	Nashville International Airport
CDG	Paris	Charles de Gaulle
LAX	Los Angeles, CA	Los Angeles International Airport
RDU	Raleigh-Durham, NC	Raleigh-Durham International Airport



To define all data set variables as data variables for the hash object, use the ALL: "YES" option.

```
hashobject.DEFINEDATA (ALL: "YES");
```



The hash object can store multiple key variables as well as multiple data variables.

## Retrieving Multiple Data Values

You can use multiple FIND method calls to retrieve multiple data values. In the following program, the FIND method calls retrieve the value of City and Name from the Airports hash object based on the value of Origin.

```

data work.report;
if _N_=1 then do;
  if 0 then set sasuser.acities (keep=Code City Name);
  declare hash airports (dataset: "sasuser.acities");
  airports.definekey ("Code");
  airports.definedata ("City", "Name");
  airports.definedone();
end;
set sasuser.revenue;
airports.find(key:origin);
OriginCity=city;
OriginAirport=name;
airports.find(key:dest);
DestCity=city;
DestAirport=name;
run;

```

Let's take a closer look at what happens when the program is submitted for execution.

The following steps show the execution of a DATA step that uses multiple FIND method calls to retrieve values from a hash object.

1. The values of Origin and Dest are read into the PDV from the input data set.
2. The first FIND method call retrieves the value of City and Name from the **Airports** hash object based on the value of Origin.
3. The first set of assignment statements then assigns the value of City to OriginCity and the value of Name to OriginAirport.

4. The second FIND method call retrieves the value of City and Name from the **Airports** hash object based on the value of Dest.
5. The second set of assignment statements then assigns the value of City to DestCity and the value of Name to DestAirport.

### Using Return Codes with the FIND Method

Method calls create a return code that is a numeric value. The value specifies whether the method succeeded or failed. A value of 0 indicates that the method succeeded. A non-zero value indicates that the method failed.

If the program does not contain a return code variable for the method call and the method fails, then an appropriate error message is written to the log.

To store the value of the return code in a variable, specify the variable name `rc` at the beginning of the method call. For example:

```
rc=hashobject.find(key:keyvalue);
```

The return code variable value can be used in conditional logic to ensure that the FIND method found a KEY value in the hash object that matches the KEY value from the PDV.

### Example

Error messages are written to the log when the following program (also shown in the previous example) is submitted.

```
data work.report;
if _N_=1 then do;
  if 0 then set sasuser.acities (keep=Code City Name);
  declare hash airports (dataset: "sasuser.acities");
  airports.definekey ("Code");
  airports.definedata ("City", "Name");
  airports.definedone();
end;
set sasuser.revenue;
airports.find(key:origin);
OriginCity=city;
OriginAirport=name;
airports.find(key:dest);
DestCity=city;
DestAirport=name;
run;
```

### SAS Log

NOTE: There were 50 observations read from the data set SASUSER.ACITIES.

ERROR: Key not found.

NOTE: The SAS System stopped processing this step because of errors.

NOTE: There were 142 observations read from the data set SASUSER.REVENUE.

WARNING: The data set WORK.REPORT1 may be incomplete. When this step was

stopped there were 142 observations and 14 variables.

A closer examination of the output shows that the data set **Work.Report** contains errors. For example, notice that in observations 6-8 the value of both OriginCity and DestCity is *Canberra, Australian C* and the values of OriginAirport and DestAirport are missing.

The errors occur because the **Airports** hash object does not include the key value *WLG* or a corresponding Name value for the key value *CBR*.

**SAS Data Set Work.Report (observations 6-8 of selected variables)**

Obs	Origin	Dest	FlightID	Date	OriginCity	OriginAirport	DestCity	DestAirport
6	CBR	WLG	IA10500	04DEC1999	Canberra, Australian C		Canberra, Australian C	
7	CBR	WLG	IA10500	16DEC1999	Canberra, Australian C		Canberra, Australian C	
8	CBR	WLG	IA10500	28DEC1999	Canberra, Australian C		Canberra, Australian C	

Conditional logic can be added to the program to create blank values if the values loaded from the input data set, **Sasuser.Revenue**, cannot be found in the **Airports** hash object:

If the return code for the FIND method call has a value of 0, indicating that the method succeeded, the value of City and Name are assigned to the appropriate variables (OriginCity and OriginAirport or DestCity and DestAirport).

If the return code for the FIND method call has a non-zero value, indicating the method failed, the value of City and Name are assigned blank values.

```
data work.report;
if _N_=1 then do;
  if 0 then set sasuser.acities(keep=Code City Name);
  declare hash airports (dataset: "sasuser.acities");
  airports.definekey ("Code");
  airports.definedata ("City", "Name");
  airports.definedone();
end;
set sasuser.revenue;
rc=airports.find(key:origin);
if rc=0 then do;
  OriginCity=city;
  OriginAirport=name;
end;
else do;
  OriginCity="";
  OriginAirport="";
end;
rc=airports.find(key:dest);
if rc=0 then do;
  DestCity=city;
  DestAirport=name;
end;
else do;
  DestCity="";
end;
```

```

DestAirport='';
end;
run;

```

PROC PRINT output shows the completed data set. Notice that in observations 6-8, the value of DestCity is now blank and no error messages appear in the log.

```

proc print data=work.report;
var origin dest flightid date origincity originairport
    destcity destairport;
run;

```

#### SAS Data Set Work.Report (first eight observations of selected variables)

Obs	Origin	Dest	FlightID	Date	OriginCity	OriginAirport	DestCity	DestAirport
1	ANC	RDU	IA03400	02DEC1999	Anchorage, AK	Anchorage International Airport	Raleigh-Durham, NC	Raleigh-Durham International Airport
2	ANC	RDU	IA03400	14DEC1999	Anchorage, AK	Anchorage International Airport	Raleigh-Durham, NC	Raleigh-Durham International Airport
3	ANC	RDU	IA03400	26DEC1999	Anchorage, AK	Anchorage International Airport	Raleigh-Durham, NC	Raleigh-Durham International Airport
4	ANC	RDU	IA03401	09DEC1999	Anchorage, AK	Anchorage International Airport	Raleigh-Durham, NC	Raleigh-Durham International Airport
5	ANC	RDU	IA03401	21DEC1999	Anchorage, AK	Anchorage International Airport	Raleigh-Durham, NC	Raleigh-Durham International Airport
6	CBR	WLG	IA10500	04DEC1999	Canberra, Australian C			
7	CBR	WLG	IA10500	16DEC1999	Canberra, Australian C			
8	CBR	WLG	IA10500	28DEC1999	Canberra, Australian C			

#### SAS Log

NOTE: There were 50 observations read from the data set SASUSER.ACITIES.  
 NOTE: There were 142 observations read from the data set SASUSER.REVENUE.  
 NOTE: The data set WORK.REPORT2 has 142 observations and 15 variables.

Which return code value indicates that a method call succeeded?

**Q.9.**

0  
1  
any non-zero value

## **Chapter Summary**

### **Introduction**

Sometimes, you need to combine data from two or more sets into a single observation in a new data set according to the values of a common variable. When data sources do not have a common structure, you can use a lookup table to match them.

### **Using Multidimensional Arrays**

When a lookup operation depends on more than one numerical factor, you can use a multidimensional array. You use an ARRAY statement to create an array. The ARRAY statement defines a set of elements that you plan to process as a group.

### **Using Stored Array Values**

In some cases, you might need to store array values in a SAS data set rather than loading them in an ARRAY statement. Array values should be stored in a SAS data set when

- there are too many values to initialize easily in the array
- the values change frequently
- the same values are used in many programs.

The first step in using stored array values is to create an array to hold the values in the target data set. The next step is to load the array elements. One method for accomplishing this task is to load the array within a DO loop. The last step is to read the actual values stored in the other source data set.

### **Using PROC TRANSPOSE**

The TRANSPOSE procedure can also be used to match data when the orientation of the data sets differs. PROC TRANSPOSE creates an output data set by restructuring the values in a SAS data set, thereby transposing selected variables into observations. The transposed (output) data set can then be merged with the other data set in order to match the data.

- The output data set contains several default variables.
- \_NAME\_ is the default name of the variable that PROC TRANSPOSE creates to identify the source of the values in each observation in the output data set. This variable is a character variable whose values are the names of the variables that are transposed from the input data set. To override the default name, use the NAME= option.
- The remaining transposed variables are named COL1...COLn by default. To override the default names, use the PREFIX= option.

### **Merging the Transposed Data Set**

You might need to use a BY statement with PROC TRANSPOSE in order to correctly structure the data for a merge. For each BY group, PROC TRANSPOSE creates one observation for each variable that it transposes. The BY variable itself is not transposed. In order to structure the data for a merge, you might also need to sort the output data set. Any other source data sets might need to be reorganized and sorted as well. When the data is structured correctly, the data sets can be merged.

### **Using Hash Objects as Lookup Tables**

Beginning with SAS®9, the hash object is available for use in a DATA step. The hash object provides an efficient, convenient mechanism for quick data storage and retrieval.

A hash object resembles a table with rows and columns; it contains a key component and a data component. Unlike an array, which uses a series of consecutive integers to address array elements, a hash object can use any combination of numeric and character values as addresses.

The hash object is a DATA step component option. Component objects are data elements that consist of attributes and methods. To use a DATA step component object in your SAS program, you must first declare and create (instantiate) the object. After you define the hash object's key and data variables, you can load the variables from hard-coded values or a SAS data set.

You use the FIND method call to retrieve matching data from the hash object. Method calls create a return code that is a numeric value. The value specifies whether the method succeeded or failed. A value of 0 indicates that the method succeeded. A nonzero value indicates that the method failed. The return code variable value can be used in conditional logic to ensure that the FIND method found KEY value in the hash object that matches the KEY value from the PDV.

## Syntax

### Using a Multidimensional Array

```
LIBNAME libref 'SAS-data-library';
DATA libref.sas-data-set (DROP=variable(s));
 array-name {rows,cols,...} <$> <length>
    <array-elements> <(initial values)>;
SET<SAS-data-set(s) <(data-set-options(s))>> <options>;
variable=expression;
variable=array-name {rows,cols,...};
RUN;
```

### Using Stored Array Values

```
LIBNAME libref 'SAS-data-library';
DATA libref.sas-data-set;
 array-name1 {rows,cols,...} <$> <length>
    <array-elements> <(initial values)>;
IF expression THEN DO index-variable=specification;
    SET<SAS-data-set(s) <(data-set-options(s))>> <options>;
    ARRAY array-name2 {rows,cols,...} <$> <length>
        <array-elements> <(initial values)>;
    DO index-variable=specification;
        array-name1 {rows,cols,...}=array-name2 {rows,cols,...};
    END;
END;
RUN;
```

### Using PROC TRANSPOSE and a Merge

```
LIBNAME libref 'SAS-data-library';
PROC TRANSPOSE <DATA=input-data-set>
    <OUT=output-data-set>
    <NAME=name>
    <PREFIX=prefix>;
BY <DESCENDING> variable-1
    <...<DESCENDING> variable-n> <NOTSORTED>;
RUN;
PROC SORT;
    BY <DESCENDING> variable-1
        <...<DESCENDING> variable-n>;
RUN;
DATA libref.sas-data-set;
    SET<SAS-data-set(s) <(data-set-options(s))>> <options>;
```

```

LENGTH variable(s) <$$> length;
variable=expression;
variable=expression;
RUN;
PROC SORT;
    BY <DESCENDING> variable-1 <...<DESCENDING> variable-n>;
RUN;
DATA libref.sas-data-set;
MERGE SAS-data-set-1 <(data-set-options)>
    SAS-data-set-2 <(data-set-options)> ;
    BY <DESCENDING> variable-1 <...<DESCENDING> variable-n>;
RUN;

```

## Using a Hash Object as a Lookup Table

```

DATA libref.sas-data-set;
IF expression THEN statement;
DECLARE object variable <(<argument tag-1: value-1<, ...argument tag-n: value-n >>)>;
variable = NEW object(<argument tag-1: value-1<, ...argument tag-n: value-n >>);
object.DEFINEKEY('keyvarname-1'<...,'keyvarname-n'>);
object.DEFINEDATA('datavarname-1'<...,'datavarname-n'>);
object.DEFINEDONE();
CALL MISSING(varname1<, varname2, ...>);
object.ADD(<KEY: keyvalue-1, ..., KEY: keyvalue-n, DATA: datavalue-1, ..., DATA: datavalue-n>);
END;
SET<SAS-data-set(s) <(data-set-options(s))>> <options>;
object.FIND(<KEY: keyvalue-1, ..., KEY: keyvalue-n>);
RUN;

```

## Sample Programs

### Using a Multidimensional Array

```

data work.wndchill (drop = column row);
array WC{4,2} _temporary_
(-22,-16,-22,-32,-26,-35,-29);
set sasuser.flights;
row = round(wspeed,5)/5;
column = (round(temp,5)/5)+3;
WindChill= wc{row,column};
run;

```

### Using Stored Array Values

```

data work.lookup1;
array Targets{1997:1999,12} _temporary_;
if _n_=1 then do i= 1 to 3;
    set sasuser.ctargets;
    array Mnth{*} Jan-Dec;
    do j=1 to dim(mnth);
        targets{year,j}=mnth{j};
    end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
year=input(substr(salemon,4),4.);
Ctarget=targets{year,monthno};

```

```

format ctarger dollar15.2;
run;

```

## Using PROC TRANSPOSE and a Merge

```

proc transpose data=sasuser.ctargets
  out=work.ctarget2
  name=Month
  prefix=Ctarget;
  by year;
run;
proc sort data=work.ctarget2;
  by year month;
run;
data work.mnthsu2;
  set sasuser.monthsum(keep=SaleMon RevCargo);
  length Month $ 8;
  Year=input(substr(SaleMon,4),4.);
  Month=substr(SaleMon,1,1)
    || lowercase(substr(SaleMon,2,2));
run;
proc sort data=work.mnthsu2;
  by year month;
run;
data work.merged;
  merge work.mnthsu2 work.ctarget2;
  by year month;
run;

```

## Using a Hash Object That is Loaded From Hard-Coded Values

```

data work.difference (drop= goalamount);
length goalamount 8;
if _N_= 1 then do;
  declare hash goal();
  goal.definekey("QtrNum");
  goal.definedata("GoalAmount");
  goal.definedone();
  call missing(qtrnum, goalamount);
  goal.add(key:'qtr1', data:10 );
  goal.add(key:'qtr2', data:15 );
  goal.add(key:'qtr3', data: 5 );
  goal.add(key:'qtr4', data:15 );
end;
set sasuser.contrib;
goal.find();
Diff = amount - goalamount;
run;

```

## Using a Hash Object That is Loaded From a SAS Data Set

```

data work.report;
if _N_=1 then do;
  if 0 then set sasuser.acities(keep=Code City Name);
declare hash airports (dataset: "sasuser.acities");
airports.definekey ("Code");
airports.definedata ("City", "Name");

```

```

airports.definedone());
end;
set sasuser.revenue;
rc=airports.find(key:origin);
if rc=0 then do;
  OriginCity=city;
  OriginAirport=name;
end;
else do;
  OriginCity="";
  OriginAirport="";
end;
rc=airports.find(key:dest);
if rc=0 then do;
  DestCity=city;
  DestAirport=name;
end;
else do;
  DestCity="";
  DestAirport="";
end;
run;

```

## Points to Remember

- The name of an array must be a SAS name that is not the name of a SAS variable in the same DATA step.
- Array elements must be either all numeric or all character.
- The initial values specified for an array can be numbers or character strings. You must enclose all character strings in quotation marks.
- The original SAS data set must be sorted or indexed prior to using a BY statement with PROC TRANSPOSE unless you use the NOTSORTED option.
- The hash object is a good choice for lookups using unordered data that can fit into memory because it provides in-memory storage and retrieval and does not require the data to be sorted.
- The hash object is sized dynamically, and exists for the duration of the DATA step.

## Chapter Quiz

Select the best answer for each question.

1. Which SAS statement correctly specifies the array Sales as illustrated in the following table?

Table Representation of Sales Array

m1	m2	m3	m4
m5	m6	m7	m8
m9	m10	m11	m12

- a. array Sales{3,4} m1-m12;  
b. array Sales{4,3} m1-m12;  
c. array {3,4} Sales m1-m12;  
d. array {4,12} Sales m1-m12;
2. Which of the following statements creates temporary array elements?
  - a. array new {\*} \_temporary\_;
  - b. array new {6} \_temporary\_;
  - c. array new {\*} \_temporary\_ Jan Feb Mar Apr May Jun;
  - d. array \_temporary\_ new {6} Jan Feb Mar Apr May Jun;
3. Which DO statement processes all of the elements in the Yearx array?

```
array Yearx{12} Jan--Dec;
```

  - a. do i=1 to dim(yearx);
  - b. do i=1 to 12;
  - c. do i=Jan to Dec;
  - d. a and b
4. Given the following program, what is the value of Points in the fifth observation in the data set **Work.Results**?

```
data work.results;
  array score{2,4} _temporary_ (40,50,60,70,40,50,60,70);
  set work.contest;
  Points=score{week,finish};
run;
```

SAS Data Set Work.Contest

Obs	Name	Week	Finish
1	Tuttle	1	1
2	Gomez	1	2
3	Chapman	1	3
4	Venter	1	4
5	Vandeusen	2	1
6	Tuttle	2	2
7	Venter	2	3
8	Gomez	2	4

- a. 40  
 b. 50  
 c. 60  
 d. 70
5. Array values should be stored in a SAS data set when
- a. there are too many values to initialize easily in an array.
  - b. the values change frequently.
  - c. the same values are used in many programs.
  - d. all of the above
6. Given the following program, which statement is **not** true?
- ```

data work.lookup1;
array Targets{1997:1999,12} _temporary_;
if _n_=1 then do i= 1 to 3;
  set sasuser.cttargets;
  array Mnth{*} Jan-Dec;
  do j=1 to dim(mnth);
    targets{year,j}=mnth{j};
  end;
end;
set sasuser.monthsum(keep=salemon revcargo monthno);
year=input(substr(salemon,4),4.);
Ctarget=targets{year,monthno};
run;
```
- a. The IF-THEN statement specifies that the Targets array is loaded once.
  - b. During the first iteration of the DATA step, the outer DO loop executes three times.
  - c. After the first iteration of the DO loop, the pointer drops down to the second SET statement.
  - d. During the second iteration of the DATA step, the condition `_N_=1` is false. So, the DO loop doesn't execute.

7. Given the following program, which variable names will appear in the data set **Work.New**?

```
proc transpose data=work.revenue
   out=work.new;
run;
```

SAS Data Set Work.Revenue

| Obs | Year | Jan       | Feb       | Mar       | Apr       |
|-----|------|-----------|-----------|-----------|-----------|
| 1   | 2000 | 1.003.561 | 922.080   | 836.068   | 973.016   |
| 2   | 2001 | 1.078.552 | 1.013.798 | 1.047.812 | 1.005.575 |
| 3   | 2002 | 1.182.442 | 1.657.323 | 1.079.866 | 1.466.640 |

- a. Year, Jan, Feb, Mar, Apr
- b. Year, 2000, 2001, 2002
- c. \_NAME\_, Col1, Col2, Col3
- d. \_NAME\_, Jan, Feb, Mar, Apr

8. Which program creates the output data set **Work.Temp2**?

SAS Data Set Work.Temp

| Obs | Month1   | Month2   | Month3   |
|-----|----------|----------|----------|
| 1   | 13604250 | 24901752 | 18166825 |
| 2   | 72551855 | 23048835 | 17804326 |
| 3   | 12540156 | 27572140 | 14152851 |
| 4   | 11241374 | 28684655 | 15156575 |

SAS Data Set Work.Temp2

| Obs | Month  | Quarter1 | Quarter2 | Quarter3 | Quarter4 |
|-----|--------|----------|----------|----------|----------|
| 1   | Month1 | 13604250 | 72551855 | 12540156 | 11241274 |
| 2   | Month2 | 24901752 | 23048835 | 27572140 | 28684655 |
| 3   | Month3 | 18166825 | 17804326 | 14152851 | 15156575 |

- a. proc transpose data=work.temp out=work.temp2  
prefix=Quarter;  
run;
- b. proc transpose data=work.temp out=work.temp2  
name=Month prefix=Quarter;  
run;
- c. proc transpose data=work.temp out=work.temp2  
prefix=Month name=Quarter;  
run;
- d. proc transpose data=work.temp out=work.temp2  
prefix=Month index=Quarter;  
run;

9. Which version of the data set **Work.Sales2** is created by the following program?

```
proc transpose data=work.sales
  out=work.sales2
  name=Week;
  by employee;
run;
```

SAS Data Set Work.Sales

| Obs | Employee    | Week1   | Week2   |
|-----|-------------|---------|---------|
| 1   | Almers      | 3393.50 | 2192.25 |
| 2   | Bonaventure | 5093.75 | 2247.50 |
| 3   | Johnson     | 1813.30 | 2082.75 |
| 4   | LaMance     | 1572.50 | 2960.00 |

a.

| Obs | Week  | COL1    | COL2    | COL3    | COL4   |
|-----|-------|---------|---------|---------|--------|
| 1   | Week1 | 3393.50 | 5093.75 | 1813.30 | 1572.5 |
| 2   | Week2 | 2192.25 | 2247.50 | 2028.75 | 2960.0 |

b.

| Obs | Employee     | Week  | COL1    |
|-----|--------------|-------|---------|
| 1   | Almers       | Week1 | 3393.50 |
| 2   | Almers       | Week2 | 2192.25 |
| 3   | Bonnaventure | Week1 | 5093.75 |
| 4   | Bonnaventure | Week2 | 2247.50 |
| 5   | Johnson      | Week1 | 1813.30 |
| 6   | Johnson      | Week2 | 2028.75 |
| 7   | LaMance      | Week1 | 1572.50 |
| 8   | LaMance      | Week2 | 2960.00 |

c.

| Obs | Week  | Almers  | Bonnaventure | Johnson | LaMance |
|-----|-------|---------|--------------|---------|---------|
| 1   | Week1 | 3393.50 | 5093.75      | 1813.30 | 1572.5  |
| 2   | Week2 | 2192.25 | 2247.50      | 2028.75 | 2960.0  |

d.

| Obs | Employee     | _NAME_ | Week    |
|-----|--------------|--------|---------|
| 1   | Almers       | Week1  | 3393.50 |
| 2   | Bonnaventure | Week1  | 5093.75 |
| 3   | Johnson      | Week1  | 1813.30 |
| 4   | LaMance      | Week1  | 1572.50 |
| 5   | Almers       | Week2  | 2192.25 |
| 6   | Bonnaventure | Week2  | 2247.50 |
| 7   | Johnson      | Week2  | 2028.75 |
| 8   | LaMance      | Week2  | 2960.00 |

10. Which program creates the data set **Work.Fishsize**?

SAS Data Set Work.Fishdata

| Obs | Location   | Date    | Length1 | Weight1 | Length2 | Weight2 |
|-----|------------|---------|---------|---------|---------|---------|
| 1   | Cole Pond  | 02JUN95 | 31      | 0.25    | 32      | 0.30    |
| 2   | Cole Pond  | 04AUG95 | 29      | 0.23    | 30      | 0.25    |
| 3   | Eagle Lake | 02JUN95 | 32      | 0.35    | 32      | 0.25    |
| 4   | Eagle Lake | 04AUG95 | 33      | 0.30    | 33      | 0.28    |

SAS Data Set Work.Fishsize

| Obs | Location   | Date    | _NAME_  | Measurement1 |
|-----|------------|---------|---------|--------------|
| 1   | Cole Pond  | 02JUN95 | Length1 | 31.00        |
| 2   | Cole Pond  | 02JUN95 | Weight1 | 0.25         |
| 3   | Cole Pond  | 02JUN95 | Length2 | 32.00        |
| 4   | Cole Pond  | 02JUN95 | Weight2 | 0.30         |
| 5   | Cole Pond  | 04AUG95 | Length1 | 29.00        |
| 6   | Cole Pond  | 04AUG95 | Weight1 | 0.23         |
| 7   | Cole Pond  | 04AUG95 | Length2 | 30.00        |
| 8   | Cole Pond  | 04AUG95 | Weight2 | 0.25         |
| 9   | Eagle Lake | 02JUN95 | Length1 | 32.00        |
| 10  | Eagle Lake | 02JUN95 | Weight1 | 0.35         |
| 11  | Eagle Lake | 02JUN95 | Length2 | 32.00        |
| 12  | Eagle Lake | 02JUN95 | Weight2 | 0.25         |
| 13  | Eagle Lake | 04AUG95 | Length1 | 33.00        |
| 14  | Eagle Lake | 04AUG95 | Weight1 | 0.30         |
| 15  | Eagle Lake | 04AUG95 | Length2 | 33.00        |
| 16  | Eagle Lake | 04AUG95 | Weight2 | 0.28         |

- a. proc transpose data=work.fishdata out=work.fishsize prefix=Measurement;  
run;
- b. proc transpose data=work.fishdata out=work.fishsize prefix=Measurement;  
by location;  
run;
- c. proc transpose data=work.fishdata out=work.fishsize prefix=Measurement;  
by date;  
run;
- d. proc transpose data=work.fishdata out=work.fishsize prefix=Measurement;  
by location date;  
run;

# Chapter 5

## Formatting Data

### Introduction

Custom formats are used to display variable values in certain ways, such as formatting a product number so that it is displayed as descriptive text. You should already be familiar with using the FORMAT procedure to create and store formats.



In this lesson you learn to document formats, use formats located in any catalog, create formats with overlapping ranges, and use the PICTURE statement to create a format for printing numbers. You also learn an easy way to update formats by creating a SAS data set from a format, updating the data set, and then creating a format from the SAS data set.

### Objectives

In this lesson, you learn to

- create formats with overlapping ranges
- create custom formats using the PICTURE statement
- use the FMTLIB keyword with the FORMAT procedure to document formats
- use the CATALOG procedure to manage format entries
- use the DATASETS procedure to associate a format with a variable
- use formats located in any catalog
- substitute formats to avoid errors
- create a format from a SAS data set
- create a SAS data set from a custom format.

## Creating Custom Formats Using the VALUE Statement

### Review of Creating Non-Overlapping Formats

As you learned in [Creating and Applying User-Defined Formats](#), you can use the VALUE statement in the FORMAT procedure to create a custom format for displaying data in a particular way. For example, suppose you have airline data and you want to create several custom formats that you can use for your report-writing tasks. You need formats that

- group airline routes into zones
- label airport codes as International or Domestic
- group cargo revenue figures into ranges.

The following PROC FORMAT step creates these three formats:

```
libname library 'c:\sas\newfnts';
proc format lib=library;
  value $routes
    'Route1' = 'Zone 1'
    'Route2' - 'Route4' = 'Zone 2'
    'Route5' - 'Route7' = 'Zone 3'
    '' = 'Missing'
    other = 'Unknown';
  value $dest
    'AKL','AMS','ARN','ATH','BKK','BRU',
    'CBR','CCU','CDG','CPH','CPT','DEL',
    'DXB','FBU','FCO','FRA','GLA','GVA',
    'HEL','HKG','HND','JED','JNB','JRS',
    'LHR','LIS','MAD','NBO','PEK','PRG',
    'SIN','SYD','VIE','WLG' = 'International'
    'ANC','BHM','BNA','BOS','DFW','HNL',
    'IAD','IND','JFK','LAX','MCI','MIA',
    'MSY','ORD','PWM','RDU','SEA','SFO' = 'Domestic';
  value revfmt
    . = 'Missing'
    low - 10000 = 'Up to $10,000'
    10000 <- 20000 = '$10,000+ to $20,000'
    20000 <- 30000 = '$20,000+ to $30,000'
    30000 <- 40000 = '$30,000+ to $40,000'
    40000 <- 50000 = '$40,000+ to $50,000'
    50000 <- 60000 = '$50,000+ to $60,000'
    60000 <- HIGH = 'More than $60,000';
run;
```



If you choose to run this example code, be sure to change the path in the LIBNAME statement to a storage location in your operating environment.

The PROC FORMAT step creates three formats: \$ROUTES. and \$DEST., which are character formats, and REVFMT., which is numeric.

\$ROUTES. groups airline routes into zones. In \$ROUTES.,

- both single values and ranges are assigned labels
- missing values are designated by a space in quotation marks and are assigned the label "Missing"

- the keyword **OTHER** is used to assign the label "Unknown" to any values that are not addressed in the range.

\$DEST. labels airport codes as either International or Domestic. In \$DEST.,

- unique character values are enclosed in quotation marks and separated by commas
- missing values and values not included in the range are not handled in this format.

REVFMT. groups cargo revenue figures into ranges. In REVFMT.,

- the "less than" operator (<) is used to show a **non-inclusive range** (10000<-20000 indicates that the first value is not included in the range)
- the keyword **LOW** is used to specify the lower limit of a variable's value range, but it does not include missing values
- missing values are designated with a period (.) and assigned the label "Missing"
- the keyword **HIGH** is used to specify the upper limit of a variable's value range.

Which VALUE statement below specifies a valid range?

**Q.1.**

```
value $itemfmt
  105AR='twill shorts'
  931BT='denim skirt'
  034QE='knit dress'
  .='no label'
  other='mislabeled';

value $itemfmt
  '105AR'='twill shorts'
  '931BT'='denim skirt'
  '034QE'='knit dress'
  '' ='no label'
  other='mislabeled';
```

Given the following FORMAT procedure, how would the value 10 be displayed when the GRADES. format is applied?

**Q.2.**

```
proc format lib=library;
  value grades 20-<70='U'
    70-<90='S'
    90-100='C';
run;
U
10
Mislabeled
```

## Creating a Format with Overlapping Ranges

There are times when you need to create a format that groups the same values into different ranges. To create overlapping ranges, use the **MULTILABEL** option in the VALUE statement in PROC FORMAT.

**General form, VALUE statement with the MULTILABEL option:**

**VALUE** *format-name* (**MULTILABEL**);

where *format-name* is the name of the character or numeric format that is being created.

**Example**

Suppose you want to create a format that groups dates into overlapping categories. In the table below, notice that each month appears in two groups.

| Value     | Label               |
|-----------|---------------------|
| Jan - Mar | 1st Quarter         |
| Apr - Jun | 2nd Quarter         |
| Jul - Sep | 3rd Quarter         |
| Oct - Dec | 4th Quarter         |
| Jan - Jun | First Half of Year  |
| Jul - Dec | Second Half of Year |

In the PROC FORMAT step below, the MULTILABEL option has been added to indicate that the DATES. format will have values with overlapping ranges:

```
proc format;
  value dates (multilabel)
    '01jan2000'd - '31mar2000'd = '1st Quarter'
    '01apr2000'd - '30jun2000'd = '2nd Quarter'
    '01jul2000'd - '30sep2000'd = '3rd Quarter'
    '01oct2000'd - '31dec2000'd = '4th Quarter'
    '01jan2000'd - '30jun2000'd = 'First Half of Year'
    '01jul2000'd - '31dec2000'd = 'Second Half of Year';
run;
```

Multilabel formatting allows an observation to be included in multiple rows or categories. To use the multilabel formats, you can specify the MLF option on class variables in procedures that support it:

- PROC TABULATE
- PROC MEANS
- PROC SUMMARY.

The MLF option activates multilabel format processing when a multilabel format is assigned to a class variable. For example, in the following TABULATE procedure,

- the FORMAT= option specifies DOLLAR15.2 as the default format for the value in each table cell
- the FORMAT statement references the new format DATES. for the variable Date
- the CLASS statement identifies Date as the class variable and uses the MLF option to activate multilabel format processing
- the row dimension of the TABLE statement creates a row for each formatted value of Date.

```
proc tabulate data = sasuser.sale2000 format = dollar15.2;
  format Date dates.;
  class Date / mlf;
  var RevCargo;
```

```
table Date, RevCargo*(mean median);
run;
```

|                     | RevCargo    |            |
|---------------------|-------------|------------|
|                     | Mean        | Median     |
| Date                |             |            |
| 1st Quarter         | \$24,839.08 | \$4,939.00 |
| 2nd Quarter         | \$14,949.77 | \$4,579.50 |
| 3rd Quarter         | \$19,836.00 | \$3,591.00 |
| 4th Quarter         | \$20,403.13 | \$1,940.00 |
| First Half of Year  | \$19,894.42 | \$4,823.00 |
| Second Half of Year | \$20,261.35 | \$2,100.00 |



For more information about using the MULTILABEL option, see the SAS documentation for the FORMAT procedure.

## Creating Custom Formats Using the PICTURE Statement

You have learned that you can use the VALUE statement to convert output values to a different form. Suppose one of the variables in your data set had numeric values that you wanted to format a certain way. For example, you might have a phone number listed in your data set as 1111231234 and you want to format it as (111) 123-1234. You can use the **PICTURE statement** to create a template for printing numbers.

### General form, PROC FORMAT with the PICTURE statement:

```
PROC FORMAT;
  PICTURE format-name
    value-or-range='picture';
RUN;
```

where

- *format-name* is the name of the format you are creating.
- *value-or-range* is the individual value or range of values you want to label.
- *picture* specifies a template for formatting values of numeric variables. The template is a sequence of characters enclosed in quotation marks. The maximum length for a picture is 40 characters.

## Ways to Specify Pictures

Pictures are specified with three types of characters:

- digit selectors
- message characters
- directives.

Let's look at using digit selectors and message characters first. You'll learn about directives a little later.

**Digit selectors** are numeric characters (0 through 9) that define positions for numeric values. If you use **nonzero** digit selectors, zeros are added to the formatted value as needed. If you use **zeros** as digit selectors, no zeros are added to the formatted value.

In the picture definitions below you can see the difference between using nonzero digit selectors (99) and zero digit selectors (00) on the formatted values.

| Picture Definition       | Data Values | Formatted Values |
|--------------------------|-------------|------------------|
| picture month 1-12='99'; | 01          | 01               |
|                          | 1           | 01               |
|                          | 12          | 12               |
| picture month 1-12='00'; | 01          | 1                |
|                          | 1           | 1                |
|                          | 12          | 12               |

**Message characters** are nonnumeric characters that print as specified in the picture. They are inserted into the picture after the numeric digits are formatted. Digit selectors must come before message characters in the picture definition. In the picture definition below, the text string JAN is made up of message characters.

| Picture Definition        | Data Value | Formatted Value |
|---------------------------|------------|-----------------|
| picture month 1='99 JAN'; | 1          | 01 JAN          |

### Example

The following PICTURE statement contains both digit selectors and message characters. Because the RAINAMT. format has nonzero digit selectors, values are printed with leading zeros. The keyword OTHER is used to print values and message characters for any values that do not fall into the specified range.

```

proc format;
  picture rainamt
    0-2='9.99 slight'
    2<-4='9.99 moderate'
    4<-<10='9.99 heavy'
    other='999 check value';
run;
data rain;
  input Amount;
  datalines;
  4
  3.9
  20
  .5
  6
  ;
run;
proc print data=rain;
  format amount rainamt.;
run;

```

The following output shows the values with the RAINAMT. format applied.

| Obs | Amount        |
|-----|---------------|
| 1   | 4.00 moderate |
| 2   | 3.90 moderate |

| Obs | Amount          |
|-----|-----------------|
| 3   | 020 check value |
| 4   | 0.50 slight     |
| 5   | 6.00 heavy      |

Next you learn about using **directives** to specify picture formats.

The final way to specify a picture is with a directive. **Directives** are special characters that you can use in the picture to format **date**, **time**, or **datetime** values. If you use a directive, you must specify the **DATATYPE= option** in the PICTURE statement. This option specifies that the picture applies to a SAS date, SAS time, or SAS datetime value.

#### General form, PICTURE statement with the DATATYPE= option:

```
PICTURE format-name
      value-or-range='picture' (DATATYPE=SAS-date-value-type);
```

where

- *format-name* is the name of the format you are creating
- *value-or-range* is the individual value or range of values you want to label
- *picture* specifies a template with directives for formatting values of numeric variables
- *SAS-date-value-type* is either **DATE**, **TIME** or **DATETIME**.

#### Guidelines for Specifying Directives

The percent sign (%) followed by a letter indicates a directive. Directives that you can use to create a picture format are listed in the table below.

| Directive | Result                                                                                               |
|-----------|------------------------------------------------------------------------------------------------------|
| %a        | abbreviated weekday name                                                                             |
| %A        | full weekday name                                                                                    |
| %b        | abbreviated month name                                                                               |
| %B        | full month name                                                                                      |
| %d        | day of the month as a number 1-31, with no leading zero                                              |
| %H        | 24-hour clock as a number 0-23, with no leading zero                                                 |
| %I        | 12-hour clock as a number 1-12, with no leading zero                                                 |
| %j        | day of the year as a number 1-366, with no leading zero                                              |
| %m        | month as a number 1-12, with no leading zero                                                         |
| %M        | minute as a decimal number 0-59, with no leading zero                                                |
| %p        | AM or PM                                                                                             |
| %S        | second as a number 0-59, with no leading zero                                                        |
| %U        | week number of the year (Sunday is the first day of the week) as a number 0-53, with no leading zero |
| %w        | weekday as a number (1=Sunday, to 7)                                                                 |
| %y        | year without century as a number 0-99, with no leading zero                                          |
| %Y        | year with century as a number                                                                        |

Although directives generally return numeric values with no leading zeros, you can add 0 in the directive so that if a one-digit numeric value is returned, it is preceded by a 0.

As shown below, when you create a picture with directives, the number of characters inside quotation marks is the maximum length of the formatted value. You must add trailing blanks to the directive if your values will contain more characters than the picture. The formatted value will be truncated if you do not.

| Formatted Result                             | Picture Definition                  |
|----------------------------------------------|-------------------------------------|
| d d - m m m y y y y <br>1 2 3 4 5 6 7 8 9 10 | '%0d-%b%Y'<br>1 2 3 4 5 6 7 8 9 10' |

### Example

Suppose you want to display values for employee hire dates in the format *dd-mmmmyyyy* (such as 25-JAN2000). This format requires spaces for 10 characters.

The following code creates this format. There are a few things you should notice about the picture definition:

- The keywords LOW and HIGH are used to include all values.
- The 0 in the %d directive indicates that if the day of the month is one digit, it should be preceded by a 0.
- Because there are only eight characters inside the single quotation marks, you must add two blank spaces to set the length to 10.

```
proc format;
picture mydate
    low-high='%0d-%b%Y' (datatype=date);
run;
proc print data=sasuser.empdata
(keep=division hireDate lastName obs=5);
format hiredate mydate. ;
run;
```

The output below shows the values for HireDate formatted with the MYDATE. picture format.

| Obs | Division                     | HireDate   | LastName |
|-----|------------------------------|------------|----------|
| 1   | FLIGHT OPERATIONS            | 11-MAR1992 | MILLS    |
| 2   | FINANCE & IT                 | 19-DEC1983 | BOWER    |
| 3   | HUMAN RESOURCES & FACILITIES | 12-MAR1985 | READING  |
| 4   | HUMAN RESOURCES & FACILITIES | 16-OCT1989 | JUDD     |
| 5   | AIRPORT OPERATIONS           | 19-DEC1981 | WONSILD  |



For more information about using the PICTURE statement, see the SAS documentation for the FORMAT procedure.

**Q.3.**

How would the value 9 be displayed when the format GRADE. is applied?

```

proc format;
picture grade
  0-5='9 Elementary School'
  6-8='9 Middle School'
  9-12='99 High School';
run;
  99 High School
  9 High School
  09 High School

```

## Managing Custom Formats

### Using FMTLIB with PROC FORMAT to Document Formats

When you have created a large number of permanent formats, it can be easy to forget the exact spelling of a specific format name or its range of values. Remember that adding the keyword **FMTLIB** to the PROC FORMAT statement displays a list of all the formats in the specified catalog, along with descriptions of their values.

```

libname library 'c:\sas\newfmt';
proc format lib=library fmtlib;
run;

```

You can also use the **SELECT** and **EXCLUDE** statements to process specific formats rather than an entire catalog.

#### General form, PROC FORMAT with FMTLIB and the SELECT and EXCLUDE statements:

```

PROC FORMAT LIB=library FMTLIB;
  SELECT format-name;
  EXCLUDE format-name;
RUN;

```

where

- *library* is the name of the library where the formats are stored. If you do not specify the LIB= option, formats in the **Work** library are listed.
- *format-name* is the name of the format you want to select or exclude.

### Example

The following code displays only the documentation for the \$ROUTES. format. Notice that you do not use a period at the end of the format name when you specify the format in the SELECT statement.

```

libname library 'c:\sas\newfmt';
proc format lib=library fmtlib;
  select $routes;
run;

```

#### SAS Output

|                                                                                                                |     |                                          |  |
|----------------------------------------------------------------------------------------------------------------|-----|------------------------------------------|--|
| FORMAT NAME : \$ROUTES LENGTH: 7 NUMBER OF VALUES: 5<br>MIN LENGTH: 1 MAX LENGTH: 40 DEFAULT LENGTH: 7 FUZZ: 0 |     |                                          |  |
| START                                                                                                          | END | LABEL (VER. V7 V8<br>29AUG2002:11:13:14) |  |

|           |           |         |
|-----------|-----------|---------|
|           |           | Missing |
| Route1    | Route1    | Zone 1  |
| Route2    | Route4    | Zone 2  |
| Route5    | Route7    | Zone 3  |
| **OTHER** | **OTHER** | Unknown |



If you specify more than one format on the SELECT or EXCLUDE statement, separate each format name with a space as follows:

```
select $routes newdate;
```

Add a statement to the PROC FORMAT step below to document all the formats in the **Sasuser.Formats** catalog EXCEPT the \$GEOAREA. and GEOPOP. formats.

**Q4.**

```
proc format lib=sasuser fmtlib;
-----  
run;
```

### Using PROC CATALOG to Manage Formats

Because formats are saved as catalog entries, you can use the CATALOG procedure to manage your formats. Using PROC CATALOG, you can

- create a listing of the contents of a catalog
- copy a catalog or selected entries within a catalog
- delete or rename entries within a catalog.

#### General form, PROC CATALOG step:

```
PROC CATALOG CATALOG=libref.catalog;  
CONTENTS <OUT=SAS-data-set>;  
COPY OUT=libref.catalog <options>;  
SELECT entry-name.entry-type(s);  
EXCLUDE entry-name.entry-type(s);  
DELETE entry-name.entry-type(s);
```

**RUN;**

**QUIT;**

where

- *libref.catalog* with the CATALOG= argument is the SAS catalog to be processed
- *SAS-data-set* is the name of the data set that will contain the list of the catalog contents
- *libref.catalog* with the OUT= argument is the SAS catalog to which the catalog entries will be copied
- *entry-name.entry-type(s)* are the full names of catalog entries (in the form *name.type*) that you want to process.



If you are using the SAS Learning Edition, you will not be able to submit a PROC CATALOG step because the CATALOG procedure is not included in the software.

### Example

The first PROC CATALOG step below copies the \$ROUTES. format from the **Library.Formats** catalog to the **Work.Formats** catalog. Notice that in the SELECT statement, you specify the \$ROUTES. character format using the full catalog entry name, ROUTES.FORMATC.

```
proc catalog catalog=library.formats;
  copy out=work.formats;
  select routes.formatc;
run;
proc catalog cat=work.formats;
  contents;
run;
quit;
```

The second PROC CATALOG step displays the contents of the **Work.Formats** catalog. A note is written to the log when the format is copied from one catalog to another.

| Contents of Catalog WORK.FORMATS |        |         |                    |                    |             |
|----------------------------------|--------|---------|--------------------|--------------------|-------------|
| #                                | Name   | Type    | Create Date        | Modified Date      | Description |
| 1                                | ROUTES | FORMATC | 17OCT2002:10:30:24 | 17OCT2002:10:30:24 |             |

 For more information about PROC CATALOG, including other statements and options that you can use, see the SAS documentation.

What does the following code do?

```
proc catalog catalog=work.formats;
  copy out=sasuser.formats;
  select newdate.formatc;
run;
quit;
```

**Q5.**

copies all formats from the **Work.Formats** catalog to the **Sasuser.Formats** catalog

copies all formats except NEWDATE. from the **Work.Formats** catalog to the **Sasuser.Formats** catalog

copies the NEWDATE. format from the **Work.Formats** catalog to the **Sasuser.Formats** catalog

## Using Custom Formats

After you have created a custom format, you can use SAS statements to permanently assign the format to a variable in a DATA step, or you can temporarily specify a format in a PROC step to determine the way that the data values appear in output. You should already be familiar with referencing a format in a FORMAT statement.

Another way to assign, change, or remove the format associated with a variable in an existing SAS data set is to use the DATASETS procedure to modify the descriptor portion of a data set.

#### General form, DATASETS procedure with the MODIFY and FORMAT statements:

```
PROC DATASETS LIB=SAS-library <NOLIST>;
  MODIFY SAS-data-set;
  FORMAT variable(s) format;
  QUIT;
```

where

- *SAS-library* is the name of the SAS library that contains the data you want to modify.
- **NOLIST** suppresses the directory listing.
- *SAS-data-set* is the name of the SAS data set you want to modify.
- *variable* is the name of one or more variables whose format you want to assign, change, or remove.
- *format* is the name of a format to apply to the variable or variables listed before it. If you do not specify a format, any format associated with the variable is removed.



The DATASETS procedure is interactive and will run until you issue the QUIT statement.

#### Example

In the following code, two variables in the SAS data set **Flights** are changed. The format \$DEST. is associated with the variable Dest and the format is removed from the variable Baggage.

```
proc datasets lib=Mylib;
  modify flights;
  format dest $dest. ;
  format baggage;
  quit;
```



You cannot run this example because **Flights** is a fictitious data set.

#### Using a Permanent Storage Location for Formats

When you permanently associate a format with a variable in a data set, it is important to be sure that the format you are referencing is stored in a permanent location. Remember that the storage location for the format is determined when the format is created in the FORMAT procedure.

When you create formats that you want to use in subsequent SAS sessions, it is useful to

1. assign the **Library** libref to a SAS library in the SAS session in which you are running the PROC FORMAT step
2. specify LIB=LIBRARY in the PROC FORMAT step that creates the format
3. include a LIBNAME statement in the program that references the format to assign the **Library** libref to the library that contains the permanent format catalog.

You can store formats in any catalog you choose; however, you must identify the format catalogs to SAS before you can access them. You learn about this in the next topic.

**Q.6.** Which PROC DATASETS step assigns the format \$GRADE. to the variable Score in the SAS data set **Sasuser.Quarter1**?

```

proc datasets lib=sasuser nolist;
  format quarter1;
  modify score $grade.;
quit;
proc datasets data=sasuser.quarter1 nolist;
  format score $grade.;
quit;
proc datasets lib=sasuser nolist;
  modify quarter1;
  format score $grade.;
quit;

```

When a format is referenced, SAS automatically looks through the following libraries in this order:

- **Work.Formats**
- **Library.Formats**.

The **Library** libref is recommended for formats because it is automatically searched when a format is referenced. If you store formats in libraries or catalogs other than those in the default search path, you must use the FMTSEARCH= system option to tell SAS where to find your formats.

#### **General form, FMTSEARCH= system option:**

**OPTIONS FMTSEARCH= (catalog-1 catalog-2...catalog-n);**

where *catalog* is the name of one or more catalogs to search. The value of *catalog* can be either *libref* or *libref.catalog*. If only the libref is given, SAS assumes that **Formats** is the catalog name.

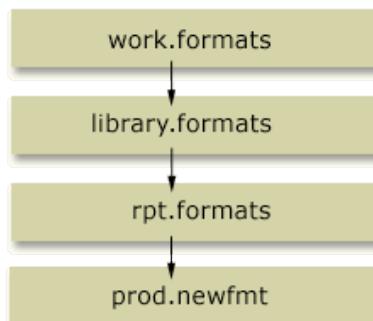
The **Work.Formats** catalog is always searched first, and the **Library.Formats** catalog is searched next, unless one or both catalogs appear in the FMTSEARCH= list.

#### **Example**

Suppose you have formats stored in the **Rpt** library and in the **Prod.Newfmt** catalog. The following OPTIONS statement tells SAS where to find your formats:

```
options fmtsearch=(rpt prod.newfmt);
```

Because no catalog is specified with the **Rpt** libref, the default catalog name **Formats** is assumed. This OPTIONS statement creates the following search order:



Because the **Work** and **Library** librefs were not specified in the FMTSEARCH= option, they are searched in default order.

Write an OPTIONS statement to specify the following search order for format catalogs:

**Q7.**

- formats stored in the **Abc** library
- formats stored in the **Newfmt** catalog of the **Def** library
- formats stored in the **Formats** catalog of the **Work** library.

### Substituting Formats to Avoid Errors

Let's look at what happens if you forget to specify a catalog in the FMTSEARCH= option, misspell a format name, or make some other mistake that causes SAS to fail to locate the format you have specified.

By default, the **FMTERR** system option is in effect. If you use a format that SAS cannot load, SAS issues an error message and stops processing the step. To prevent this, you must change the system option FMTERR to NOFMTERR. When **NOFMTERR** is in effect, SAS substitutes a format for the missing format and continues processing.

#### General form, FMTERR system option:

**OPTIONS FMTERR | NOFMTERR;**

where

- **FMTERR** specifies that when SAS cannot find a specified variable format, it generates an error message and stops processing. Substitution does not occur.
- **NOFMTERR** replaces missing formats with the *w.* or *\$w.* default format and continues processing.

### Example

Suppose the FMTERR system option is in effect. In a previous example, we created the \$ROUTES format to group airline routes into zones. In the following code, the \$ROUTES. format is misspelled:

```
proc print data=sasuser.cargorev(obs=10);
  format route $route.;
run;
```

Because FMTERR is in effect, the format cannot be located and SAS stops processing. An error message is written to the log.

#### SAS Log

```
30 proc print data=sasuser.cargorev(obs=10);
31   format route $route.;
ERROR: The format $ROUTE was not found or could not be loaded.
32 run;
```

NOTE: The SAS System stopped processing this step because of errors.

If the NOFMTERR system option is specified, substitution occurs and SAS continues to process the step.

```
options nofmterr;
```

```

proc print data=sasuser.cargorev(obs=10);
  format Route $route.;
run;

```

SAS substitutes the \$w format for the \$ROUTE. format that it could not locate. No message is written to the log and processing continues. You can see from the output that the format you intended to use has not been applied.

| Obs | Month | Date  | RevCargo | Route  |
|-----|-------|-------|----------|--------|
| 1   | 1     | 14610 | 2260     | Route2 |
| 2   | 1     | 14610 | 220293   | Route3 |
| 3   | 1     | 14610 | 4655     | Route1 |
| 4   | 1     | 14610 | 4004     | Route1 |
| 5   | 1     | 14611 | 8911     | Route1 |
| 6   | 1     | 14611 | 102900   | Route3 |
| 7   | 1     | 14612 | 1963     | Route3 |
| 8   | 1     | 14612 | 3321     | Route5 |
| 9   | 1     | 14612 | 2562     | Route3 |
| 10  | 1     | 14612 | 9447     | Route1 |

Given the following PROC FORMAT step, select the OPTIONS statement that must be submitted before SAS can use the REGION. format.

**Q.8.**

```

proc format lib=sasuser;
  value region
    1='North'
    2='South'
    3='East'
    4='West';
  run;
  proc print data=countrydata;
    format code region.;
  run;

  options fmtsearch=(sasuser);
  options nofmterr;
  options nofmterr fmtsearch=(work);

```

## Creating Formats from SAS Data Sets

You have seen that you can create a format by specifying values and labels in a PROC FORMAT step. You can also create a format from a SAS data set that contains value information (called a control data set). To do this, you use the **CNTLIN= option** to read the data and create the format.

**General form, CNTLIN= option in PROC FORMAT:**

```
PROC FORMAT LIBRARY=libref.catalog CNTLIN=SAS-data-set;
```

where

- *libref.catalog* is the name of the catalog in which you want to store the format
- *SAS-data-set* is the name of the SAS data set that you want to use to create the format.

### Example

Suppose you have a SAS data set named **Routes** that has the variables required to create a format. You specify the data set in the CNTLIN= option as follows:

```
proc format lib=library.formats cntlin=sasuser.routes;
run;
```

As you can see, the code for creating a format from a SAS data set is simple. However, the control data set must contain certain variables before it can be used to create a format, and most data sets must be restructured before they can be used.

### Rules for Control Data Sets

When you create a format using programming statements, you specify the name of the format, the range or value, and the label for each range or value as in the VALUE statement below:

```
value rainfall 0='none';
```

The control data set you use to create a format must contain variables that supply this same information. That is, the data set specified in the CNTLIN= option

- must contain the variables FmtName, Start, and Label, which contain the format name, value or beginning value in the range, and label.
- must contain the variable End if a range is specified. If there is no End variable, SAS assumes that the ending value of the format range is equal to the value of Start.
- must contain the variable Type for character formats, unless the value for FmtName begins with a \$.
- must be sorted by FmtName if multiple formats are specified.

Now let's see how you create a correctly structured data set.

### Example

Suppose you want to create a format that labels a three-letter airport code with the name of the city where the airport is located. You have a data set, **Sasuser.Acities**, that contains airport codes and airport cities. However, the data does not have the required variables for the CNTLIN= option.

**SAS Data Set Sasuser.Acities (Partial Listing)**

| City Where<br>Airport is Located | Start<br>Point | Airport Name                    | Country Where<br>Airport is<br>Located |
|----------------------------------|----------------|---------------------------------|----------------------------------------|
| Auckland                         | AKL            | International                   | New Zealand                            |
| Amsterdam                        | AMS            | Schiphol                        | Netherlands                            |
| Anchorage, AK                    | ANC            | Anchorage International Airport | USA                                    |

| <b>City Where Airport is Located</b> | <b>Start Point</b> | <b>Airport Name</b>              | <b>Country Where Airport is Located</b> |
|--------------------------------------|--------------------|----------------------------------|-----------------------------------------|
| Stockholm                            | ARN                | Arlanda                          | Sweden                                  |
| Athens (Athinai)                     | ATH                | Hellenikon International Airport | Greece                                  |
| Birmingham, AL                       | BHM                | Birmingham International Airport | USA                                     |
| Bangkok                              | BKK                | Don Muang International Airport  | Thailand                                |

To create a format from this data set, you need to

1. List data set variables
2. Restructure the data

### **Step 1: List Data Set Variables**

Remember that you need to have the variables FmtName, Start, and Label. You can submit a PROC CONTENTS step to get a listing of the variables in the **Sasuser.Acities** data set.

```
proc contents data=sasuser.acities;
run;
```

### Partial Output

| Alphabetic List of Variables and Attributes |          |      |     |                                  |
|---------------------------------------------|----------|------|-----|----------------------------------|
| #                                           | Variable | Type | Len | Label                            |
| 1                                           | City     | Char | 22  | City Where Airport is Located    |
| 2                                           | Code     | Char | 3   | Start Point                      |
| 4                                           | Country  | Char | 40  | Country Where Airport is Located |
| 3                                           | Name     | Char | 50  | Airport Name                     |



You can also get a list of variable names by using PROC DATASETS with a CONTENTS statement or by viewing the properties of the SAS data set in the SAS Explorer window.



You can also get a list of variable names by using PROC DATASETS with a CONTENTS statement or by viewing the properties of the SAS data set in the Properties window.

### Step 2: Restructure the Data

Once you have looked at the data and know the variable names, you are ready to write a DATA step to manipulate the data. The variable Code is the three-letter airport code and the variable City is the city where the airport is located. You can rename the variable Code to Start and the variable City to Label, but you also need to create the variable FmtName.

The code below is an efficient way to get your data ready to use. The DATA step uses

- the KEEP statement to write only the specified variables to the output data set
- the RETAIN statement to create the variable FmtName and set the value to '\$airport'
- the RENAME statement to rename the variable Code to Start (you do not need a variable named End because you are labeling discrete values rather than ranges) and to rename the variable City to Label.

```
data sasuser.aports;
keep Start Label FmtName;
retain FmtName '$airport';
set sasuser.acities (rename=(Code=Start
                           City= Label));
run;
proc print data=sasuser.aports(obs=10) noobs;
run;
```

Below is the listing of the first ten observations in the new data set **Sasuser.Aports**.

| FmtName   | Label            | Start |
|-----------|------------------|-------|
| \$airport | Auckland         | AKL   |
| \$airport | Amsterdam        | AMS   |
| \$airport | Anchorage, AK    | ANC   |
| \$airport | Stockholm        | ARN   |
| \$airport | Athens (Athinai) | ATH   |
| \$airport | Birmingham, AL   | BHM   |
| \$airport | Bangkok          | BKK   |

| FmtName   | Label                | Start |
|-----------|----------------------|-------|
| \$airport | Nashville, TN        | BNA   |
| \$airport | Boston, MA           | BOS   |
| \$airport | Brussels (Bruxelles) | BRU   |

This data set is now in the proper format to be used to create a format using the CNTLIN= option. Next, let's look again at the code that creates the format from this data set.

Once you have the data in the proper format, you can use the CNTLIN= option to create the format. The first PROC FORMAT step creates a format from the data set **Sasuser.Aports**. The second PROC FORMAT step documents the new format.

```
proc format library=sasuser cntlin=sasuser.aports;
run;
proc format library=sasuser fmtlib;
  select $airport;
run;
```

The first few lines of the output are shown below.

| Partial SAS Output                                      |     |                                       |
|---------------------------------------------------------|-----|---------------------------------------|
| FORMAT NAME : \$AIRPORT LENGTH: 22 NUMBER OF VALUES: 52 |     |                                       |
| MIN LENGTH: 1 MAX LENGTH: 40 DEFAULT LENGTH: 22 FUZZ: 0 |     |                                       |
| START                                                   | END | LABEL (VER. V7 V8 21OCT2002:14:13:14) |
| AKL                                                     | AKL | Auckland                              |
| AMS                                                     | AMS | Amsterdam                             |
| ANC                                                     | ANC | Anchorage, AK                         |
| ARN                                                     | ARN | Stockholm                             |
| ATH                                                     | ATH | Athens (Athinai)                      |
| BHM                                                     | BHM | Birmingham, AL                        |
| BKK                                                     | BKK | Bangkok                               |

### Apply the Format

Let's take a look at this format applied to the data set **Sasuser.Cargo99**. The following PROC PRINT code assigns the \$AIRPORT format to both the Dest and Origin variables:

```
proc print data=sasuser.cargo99 (obs=5);
  var origin dest cargorev;
  format origin dest $airport. ;
run;
```

| Obs | Origin             | Dest            | CargoRev     |
|-----|--------------------|-----------------|--------------|
| 1   | Raleigh-Durham, NC | London, England | \$111,720.00 |
| 2   | Raleigh-Durham, NC | London, England | \$109,270.00 |
| 3   | Raleigh-Durham, NC | London, England | \$109,270.00 |
| 4   | Raleigh-Durham, NC | London, England | \$116,130.00 |
| 5   | Raleigh-Durham, NC | London, England | \$108,290.00 |



For more information about using the CNTLIN= option, see the SAS documentation for the FORMAT procedure.

Which of the following PROC FORMAT steps creates a format from the temporary data set **Citycode** and saves the format to **Sasuser.Formats**?

**Q.9.**

```
proc format lib=work cntlin=sasuser.citycode;
run;
proc format lib=sasuser.formats cntlout=citycode;
run;
proc format lib=sasuser cntlin=citycode;
run;
```

What variables **must** be included in a data set when it is used in the CNTLIN= option?

**Q.10.**

- Format, Start, Label
- FmtName, Start, Label
- FmtName, Value, Label

## Creating SAS Data Sets from Custom Fomats

You know how to create a format from a SAS data set, but what if you want to create a SAS data set from a format? To do this, you use the CNTLOUT= option.

### General form, CNTLOUT= option in PROC FORMAT:

```
PROC FORMAT LIBRARY=libref.catalog CNTLOUT=SAS-data-set;
  SELECT format-name format-name...;
  EXCLUDE format-name format-name...;
RUN;
```

where

- *libref.catalog* is the name of the catalog in which the format is located
- *SAS-data-set* is the name of the SAS data set that you want to create
- *format-name* is the name of the format that you want to select or exclude.

The output control data set will contain variables that completely describe all aspects of each format, including optional settings. The output data set contains one observation per range per format in the specified catalog. You can use either the **SELECT** or **EXCLUDE** statement to include specific formats in the data set.

Creating a SAS data set from a format is very useful when you need to add information to a format but no longer have the the SAS data set you used to create the format. When you need to update a format, you can

1. create a SAS data set from the values in a format using CNTLOUT=
2. edit the data set using any number of methods
3. create a format from the updated SAS data set using CNTLIN=.

Next let's look at each of these steps individually.

**Q.11.**

Complete the PROC FORMAT step to create the SAS data set **Sasuser.Runs** from the format \$MARATHON. stored in the **Library.Formats** catalog.

```
-----  
select $marathon;
```

```
run;
```

## Example

In the last example, you created the \$AIRPORT. format. Suppose you want to add the following airport codes to the format:

| Value | Label        |
|-------|--------------|
| YYC   | Calgary, AB  |
| YYZ   | Toronto, ON  |
| YQB   | Quebec, QC   |
| YUL   | Montreal, QC |

### Step 1: Create a SAS Data Set from the Format

First, you write the \$AIRPORT. format out as a SAS data set. In the code below, the output data set is named **Sasuser.Fmtdata**. The SELECT statement is used so that the resulting data set has only the data for the \$AIRPORT. format. Without the SELECT statement, the data would have values for all the formats in the **Sasuser.Formats** catalog.

```
proc format lib=sasuser cntlout=sasuser.fmtdata;
  select $airport;
run;
```

When you use the CNTLOUT= option, SAS creates an output data set that has many variables for storing information about the format. The output data set **Sasuser.Fmtdata** has 50 rows and 21 columns. In the PRINT procedure below, the VAR statement specifies only a few of the variables to print:

```
proc print data=sasuser.fmtdata (obs=5) noobs;
  var fmtname start end label min max
    default length fuzz;
run;
```

| FMTNAME | START | END | LABEL            | MIN | MAX | DEFAULT | LENGTH | FUZZ |
|---------|-------|-----|------------------|-----|-----|---------|--------|------|
| AIRPORT | AKL   | AKL | Auckland         | 1   | 40  | 22      | 22     | 0    |
| AIRPORT | AMS   | AMS | Amsterdam        | 1   | 40  | 22      | 22     | 0    |
| AIRPORT | ANC   | ANC | Anchorage, AK    | 1   | 40  | 22      | 22     | 0    |
| AIRPORT | ARN   | ARN | Stockholm        | 1   | 40  | 22      | 22     | 0    |
| AIRPORT | ATH   | ATH | Athens (Athinai) | 1   | 40  | 22      | 22     | 0    |

As you can see, the data set contains End and other variables that were not in the original data. When you use the CNTLIN= option, if there is no End variable in the data set, SAS assumes that the Start and End variables have the same value. When you write the format out as a data set using the CNTLOUT= option, both variables are in the data set.

### Step 2: Edit the Data Set

The next step in updating the format is to edit the data set. You could use PROC SQL or a DATA step to add lines to the data set, or you could add rows using VIEWTABLE or data grid. Whatever method you choose, you **must** add values for the FmtName, Start, End, and Label variables. If the values for Start

and End are the same, you must enter values for both variables or SAS will return an error. You do not have to add values for the other variables in the data set.

### Step 3: Create a Format from the SAS Data Set

Once the data set is edited and saved, you can create a format from the data set using the CNTLIN= option. The following code creates the \$AIRPORT. format and then uses FMTLIB to document it:

```
proc format library=sasuser cntlin=sasuser.fmtdata;
run;
proc format lib=sasuser fmtlib;
  select $airport;
run;
```

The partial output shown below includes the new values in the format.

**Partial SAS Output**

| FORMAT NAME : \$AIRPORT LENGTH: 22 NUMBER OF VALUES: 56<br>MIN LENGTH: 1 MAX LENGTH: 40 DEFAULT LENGTH: 22 FUZZ: 0 |     |                         |
|--------------------------------------------------------------------------------------------------------------------|-----|-------------------------|
| START                                                                                                              | END | LABEL (CONT'D)          |
| SYD                                                                                                                | SYD | Sydney, New South Wales |
| VIE                                                                                                                | VIE | Wien (Vienna)           |
| WLG                                                                                                                | WLG | Wellington              |
| YQB                                                                                                                | YQB | Quebec, QC              |
| YUL                                                                                                                | YUL | Montreal, QC            |
| YYC                                                                                                                | YYC | Calgary, AB             |
| YYZ                                                                                                                | YYZ | Toronto, ON             |



For more information about using the CNTLOUT= option, see the SAS documentation for the FORMAT procedure.

## **Chapter Summary**

### **Creating Custom Formats Using the VALUE Statement**

Character and numeric formats are created by using VALUE statements in a FORMAT procedure. When you specify a libref on the LIBRARY= option, the format is stored in the specified library. If no catalog name is specified, the format is saved in the **Formats** catalog by default.

### **Creating Formats with Overlapping Ranges**

Use the MULTILABEL option to create a format that has overlapping ranges. When a format has overlapping ranges, the values in the format may have more than one label. This format can be used in procedures that support the MLF option.

### **Creating Custom Formats Using the PICTURE Statement**

The PICTURE statement is used to create a template for formatting values of numeric variables. Pictures are specified using digit selectors, message characters, and directives.

### **Documenting Formats**

Use the FMTLIB keyword in the PROC FORMAT statement to get documentation about the formats in the specified catalog. The output displays the format name, start and end values, and the label. You can also use the SELECT and EXCLUDE statements to process specific formats rather than an entire catalog.

### **Managing Formats**

Because formats are saved as catalog entries, you use PROC CATALOG to copy, rename, delete, or create a listing of the entries in a catalog.

### **Using Custom Formats**

Once you have created a format, you can reference it as you would reference a SAS format. If you have stored the format in a location other than **Work.Formats**, you must use the FMTSEARCH= system option to add the location to the search path so that SAS can locate the format. It can be useful to change the default FMTERR system option to NOFMTERR so that if SAS does not find a format you reference, it will substitute the *w.* or *\$w.* format and continue processing.

You can permanently associate a format with a variable by modifying the data set using PROC DATASETS.

### **Creating Formats from SAS Data Sets**

Use the CNTLIN= option to specify a SAS data set that you want to use to create a format. The SAS data set must contain the variables FmtName, Start, and Label. If the values have ranges, there must also be an End variable.

### **Creating SAS Data Sets from Formats**

Use the CNTLOUT= option to create a SAS data set from a format. This is useful for maintaining formats because you can easily update a SAS data set.

## Syntax

```
OPTIONS FMTSEARCH=(catalog catalog);
OPTIONS FMTERR | NOFMTERR;
LIBNAME libref 'SAS-data-library';

PROC FORMAT LIBRARY=libref;
  VALUE format-name (MULTILABEL)
    range1='label1'
    rangen='labeln';
RUN;

PROC FORMAT;
  PICTURE format-name
    range1='picture1'
    rangen='picturen';
RUN;

PROC FORMAT;
  PICTURE format-name
    value-or-range='picture'
    (DATATYPE=SAS-date-value-type);
RUN;

PROC FORMAT LIB=library FMTLIB;
  SELECT format-name1 format-name2;
  EXCLUDE format-name1 format-name2;
RUN;

PROC FORMAT LIBRARY=libref.catalog
  CNTLIN=SAS-data-set;
  SELECT format-name1 format-name2;
  EXCLUDE format-name1 format-name2;
RUN;

PROC FORMAT LIBRARY=libref.catalog
  CNTLOUT=SAS-data-set;
  SELECT format-name1 format-name2;
  EXCLUDE format-name1 format-name2;
RUN;

PROC CATALOG CATLOG=libref.catalog;
  CONTENTS <OUT=SAS-data-set>;
  COPY OUT=libref.catalog <options>;
  SELECT entry-name.entry-type(s);
  EXCLUDE entry-name.entry-type(s);
  DELETE entry-name.entry-type(s);
RUN;
QUIT;

PROC DATASETS LIB=SAS-library;
  MODIFY SAS-data-set;
  FORMAT variable(s) format;
QUIT;
```

## Sample Programs

### Creating a Multilabel Format

```
proc format;
  value dates (multilabel)
```

```

'01jan2000'd - '31mar2000'd = '1st Quarter'
'01apr2000'd - '30jun2000'd = '2nd Quarter'
'01jul2000'd - '30sep2000'd = '3rd Quarter'
'01oct2000'd - '31dec2000'd = '4th Quarter'
'01jan2000'd - '30jun2000'd = 'First Half of Year'
'01jul2000'd - '31dec2000'd = 'Second Half of Year';
run;

```

### **Creating a Picture Format**

```

proc format;
picture rainamt
  0-2='9.99 slight'
  2<-4='9.99 moderate'
  4<-<10='9.99 heavy'
  other='999 check value';
run;

```

### **Creating a Picture Format Using Directives**

```

proc format;
picture mydate
  low-high='%0d-%b%Y '(datatype=date);
run;

```

### **Restructuring a SAS Data Set and Creating a Format from the Data**

```

data sasuser.aports;
keep Start Label FmtName;
retain FmtName '$airport';
set sasuser.acities (rename=(Code=Start
  City= Label));
run;

proc format library=sasuser cntlin=sasuser.aports;
run;

```

### **Creating a Format from a SAS Data Set**

```

proc format lib=sasuser cntlout=sasuser.fmtdata;
  select $airport;
run;

```

### **Points to Remember**

- By default, SAS searches for formats in the **Work.Formats** and **Library.Formats** catalogs. If you store formats in other catalogs, you must use the FMTSEARCH= system option to tell SAS where to look for your formats.
- You can use the CNTLIN= option to create a format from a SAS data set, but the data
- must contain the variables FmtName, Start, and Label
- must contain the variable Type for character formats, unless the value for FmtName begins with a \$
- must contain the variable End if a range is specified.

## Chapter Quiz

Select the best answer for each question

1. Which SAS system option is used to identify format catalogs to SAS?
  - a. FMTERR
  - b. FMTLIB
  - c. NOFMTERR
  - d. FMTSEARCH=
2. Given the following PROC FORMAT step, how is the value 70 displayed when the AGEGRP. format is applied?

```
proc format;
  picture agegrp
    1-<13='00 Youth'
    13-<20='00 Teen'
    20-<70='00 Adult'
    70-high='000 Senior';
run;
```

  - a. 000 Senior
  - b. 70 Adult
  - c. 70 Senior
  - d. 070 Senior
3. When the NOFMTERR system option is in effect, what happens when SAS encounters a format it cannot locate?
  - a. Creates the format in the default **Work.Formats** directory and continues processing.
  - b. Substitutes the \$w. or w. format and continues processing.
  - c. Stops processing and writes an error message to the log.
  - d. Skips processing at that step and continues with the next step and writes a note to the log.
4. Which of the following variables **must** be in the data set that is specified on the CNTLIN= option?
  - a. End
  - b. FmtName
  - c. Value
  - d. Description
5. Given the following code, what option is missing?

```
proc format;
  value times (?)
    '00:00't-'04:59't = 'Red Eye'
    '05:00't-'11:59't = 'Morning'
    '12:00't-'17:59't = 'Afternoon'
    '18:00't-'23:59't = 'Evening'
    '00:00't-'11:59't = 'AM'
    '12:00't-'23:59't = 'PM';
run;
```

- a. MULTILABEL
  - b. MULTIRANGE
  - c. MLF
  - d. MULTIFORMAT
6. Which PROC FORMAT option is used to create a SAS data set from a format?
- a. CNTLIN=
  - b. LIB=
  - c. CNTLOUT=
  - d. FMTLIB
7. Given the following OPTIONS statement, in what order will SAS search to find a user-defined format?
- ```
options fmtsearch=(work abc.newfmt sasuser);
```
- a. **Work.Formats** ▶ **Abc.Newfmt** ▶ **Sasuser.Formats** ▶ **Library.Formats**
  - b. **Work.Formats** ▶ **Library.Formats** ▶ **Abc.Newfmt** ▶ **Sasuser.Formats**
  - c. **Work.Formats** ▶ **Abc.Newfmt** ▶ **Sasuser.Format**
  - d. the default search order
8. What option is used with PROC FORMAT to document the formats in a particular format catalog?
- a. FMTSEARCH
  - b. FMTERR
  - c. CATALOG
  - d. FMTLIB
9. Which set of statements would you add to the PROC CATALOG code to copy the LEVELS. and \$PICKS. formats from the **Sasuser.Formats** catalog to the **Work.Formats** catalog?
- ```
proc catalog cat=sasuser.formats;
?
?
run;
```
- a. copy out=sasuser.formats;
select levels.format \$picks.format;
  - b. copy out=work.formats;
select levels \$picks;
  - c. copy out=work.formats;
select levels.format picks.formatc;
  - d. copy out=work.formats;
select levels.format \$picks.format;

10. Given the following PROC FORMAT step, how is the value 6.1 displayed when the SKICOND format is applied?

```
proc format;
  value skicond
    0-3='Poor'
    3<-6='Fair'
    6<-9='Good'
    9<-high='Excellent';
run;
```

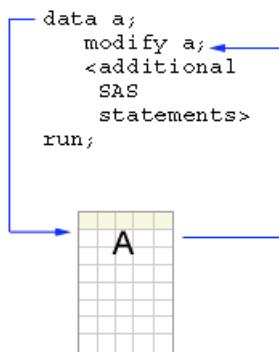
- a. 6.1
- b. Fair
- c. Good
- d. .

# Chapter 6

## Modifying SAS Data Sets and Tracking Changes

### Introduction

There are times when you want to modify the observations in a SAS data set without replacing the data set. You can do this in a DATA step with the MODIFY statement. Using the MODIFY statement, you can replace, delete, or append observations in an existing data set without creating an additional copy of the data. In this lesson, you learn to modify all the observations in a data set, matching observations using a BY statement, and observations located using an index.



When you modify data, it's often essential to safeguard your data and track the changes that are made. In this lesson you learn how to create integrity constraints to protect your data. You will also learn two different methods of tracking changes — audit trails and generation data sets. You use audit trails to track changes that are made to a data set in place, and you use generation data sets to track changes that are made when the data set is rebuilt.

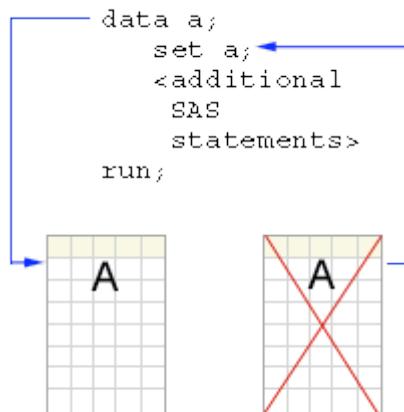
### Objectives

In this lesson, you learn to

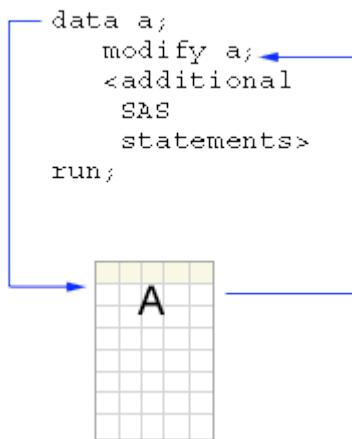
- use the MODIFY statement to update all observations in a SAS data set
- use a transaction data set to make modifications to a SAS data set
- use an index to locate observations to modify in a SAS data set
- place integrity constraints on variables in a SAS data set
- initiate and manage an audit trail file
- create and process generation data sets.

## Using the MODIFY Statement

When you submit a DATA step to create a SAS data set that is also named in a **MERGE**, **UPDATE**, or **SET** statement, SAS creates a second copy of the input data set. Once execution is complete, SAS deletes the original copy of the data set. As a result, the original data set is replaced by the new data set. The new data set can contain a different set of variables than the original data set and the attributes of the variables in the new data set can be different from those of the original data set.



In contrast, when you submit a DATA step to create a SAS data set that is also named in the **MODIFY statement**, SAS does not create a second copy of the data but instead updates the data set in place. Any variables can be added to the program data vector (PDV), but they are not written to the data set. Therefore, the set of variables in the data set does **not** change when the data is modified.



When you use the MODIFY statement, there is an implied REPLACE statement at the bottom of the DATA step instead of an OUTPUT statement. Using the MODIFY statement, you can update

- every observation in a data set
- observations using a transaction data set and a BY statement
- observations located using an index.



If the system terminates abnormally while a DATA step that is using the MODIFY statement is processing, you can lose data and possibly damage your master data set. You can recover from the failure by

- restoring the master file from a backup and restarting the step, or

- keeping an audit trail file and using it to determine which master observations have been updated.

You will learn about audit trails later in this lesson.

First let's take a look at using the MODIFY statement to modify all the observations in the data set.

## Modifying All Observations in a SAS Data Set

When every observation in a SAS data set requires the same modification, you can use the MODIFY statement and specify the modification using an assignment statement.

### General form, MODIFY statement with an assignment statement:

```
DATA SAS-data-set;
  MODIFY SAS-data-set;
    existing-variable = expression;
  RUN;
```

where

- *SAS-data-set* is the name of the SAS data set that you want to modify
- *existing-variable* is the name of the variable whose values you want to update
- *expression* is a function or other expression that you want to apply to the variable.

### Example

Suppose an airline has decided to give passengers more leg room. To do so the airline must decrease the number of seats in the business and economy classes. The SAS data set **Capacity** has the variables CapEcon and CapBusiness that hold values for the number of seats in the economy and business classes.

In the program below, the assignment statement for CapEcon reduces the number of seats in the economy class to 95% of the original number, and the assignment statement for CapBusiness reduces the number of seats in the business class to 90% of the original number. The INT function is used in both assignment statements to return the integer portion of the result.



If you choose to run this example, you must copy the data set **Capacity** from the **Sasuser** library to the **Work** library.

```
proc print data=capacity (obs=4);
run;
data capacity;
  modify capacity;
  CapEcon = int(CapEcon * .95);
  CapBusiness = int(CapBusiness * .90);
run;
proc print data=capacity (obs=4);
run;
```

The following output shows the data **before** the MODIFY statement.

| Obs | FlightID | RouteID | Origin | Dest | Cap1st | CapBusiness | CapEcon |
|-----|----------|---------|--------|------|--------|-------------|---------|
|-----|----------|---------|--------|------|--------|-------------|---------|

| Obs | FlightID | RouteID | Origin | Dest | Cap1st | CapBusiness | CapEcon |
|-----|----------|---------|--------|------|--------|-------------|---------|
| 1   | IA00100  | 0000001 | RDU    | LHR  | 14     | 30          | 163     |
| 2   | IA00201  | 0000002 | LHR    | RDU  | 14     | 30          | 163     |
| 3   | IA00300  | 0000003 | RDU    | FRA  | 14     | 30          | 163     |
| 4   | IA00400  | 0000004 | FRA    | RDU  | 14     | 30          | 163     |

The following output shows the data **after** the MODIFY statement. You can see that the values in CapBusiness and CapEcon have been reduced.

| Obs | FlightID | RouteID | Origin | Dest | Cap1st | CapBusiness | CapEcon |
|-----|----------|---------|--------|------|--------|-------------|---------|
| 1   | IA00100  | 0000001 | RDU    | LHR  | 14     | 27          | 154     |
| 2   | IA00201  | 0000002 | LHR    | RDU  | 14     | 27          | 154     |
| 3   | IA00300  | 0000003 | RDU    | FRA  | 14     | 27          | 154     |
| 4   | IA00400  | 0000004 | FRA    | RDU  | 14     | 27          | 154     |

### How SAS Updates Data Using the MODIFY Statement

The following steps show how the DATA step updates values in the SAS data set that is specified in the MODIFY statement.

1. The following program is submitted to SAS:

```
data capacity;
  modify capacity;
  CapEcon = int(CapEcon * .95);
  CapBusiness = int(CapBusiness * .90);
run;
```

2. The MODIFY statement reads the first observation from **Sasuser.Capacity** into the PDV.
3. The assignment statements update values for CapBusiness and CapEcon in the PDV.
4. An implied REPLACE statement writes the updated observation back over the same observation in **Sasuser.Capacity**. The process repeats for all observations in the data set.

What is true about using the MODIFY statement?

- Q.1.**
- When you use the MODIFY statement alone, you can select the observations that you want to modify.
  - When you use the MODIFY statement, there is an implied OUTPUT statement at the end of the DATA step.
  - When you use the MODIFY statement, you must specify the same data set that is specified on the DATA statement.

### Modifying Observations Using a Transaction Data Set

You have seen that you can use a MODIFY statement to update all observations in a data set, but there are times when you only want to update selected observations. You can modify a **master** SAS data set with values in a **transaction** data set by using the MODIFY statement with a BY statement to apply updates by matching observations.

### General form, MODIFY statement with a BY statement:

```
DATA SAS-data-set;
  MODIFY SAS-data-set transaction-data-set;
    BY key-variable;
  RUN;
```

where

- *SAS-data-set* is the name of the SAS data set that you want to modify (also called the master data set)
- *transaction-data-set* is the name of the SAS data set in which the updated values are stored
- *key-variable* is the name of the variable whose values will be matched in the master and transaction data sets.



In the MODIFY statement, you must list the master data set followed by the transaction data set.

The BY statement matches observations from the transaction data set with observations in the master data set. When the MODIFY statement reads an observation from the transaction data set, it uses dynamic WHERE processing (SAS internally generates a WHERE statement) to locate the matching observation in the master data set. The matching observation in the master data set can be replaced, deleted, or appended. By default, the observation is replaced.



Because the MODIFY statement uses WHERE processing to locate matching observations, neither data set requires sorting. However, having the master data set sorted or indexed and the transaction data set sorted reduces processing overhead, especially for large files.

### Example

Suppose you have a master data set, **Capacity**, which has route numbers for an airline. Some of the route numbers have changed, and the changes are stored in a transaction data set, **Newrtnum**. The master data set is updated by matching values of the variable FlightID.



If you choose to run this example, you must copy the data set **Capacity** from the **Sasuser** library to the **Work** library.

```
proc print data=capacity(obs=5);
run;
data capacity;
  modify capacity sasuser.newrtnum;
    by flightid;
  run;
proc print data=capacity(obs=5);
run;
```

The following PROC PRINT output displays the first five rows of the data set **Capacity** before updates were applied.

| Obs | FlightID | RouteID | Origin | Dest | Cap1st | CapBusiness | CapEcon |
|-----|----------|---------|--------|------|--------|-------------|---------|
| 1   | IA00100  | 0000001 | RDU    | LHR  | 14     | 30          | 163     |
| 2   | IA00201  | 0000002 | LHR    | RDU  | 14     | 30          | 163     |
| 3   | IA00300  | 0000003 | RDU    | FRA  | 14     | 30          | 163     |
| 4   | IA00400  | 0000004 | FRA    | RDU  | 14     | 30          | 163     |

| Obs | FlightID | RouteID | Origin | Dest | Cap1st | CapBusiness | CapEcon |
|-----|----------|---------|--------|------|--------|-------------|---------|
| 5   | IA00500  | 0000005 | RDU    | JFK  | 16     | .           | 251     |

As you can see in this PROC PRINT output, three values of FlightID in **Newrtnum** have matches in **Capacity**. For each matching observation, the values for RouteID are updated.

| Obs | FlightID | RouteID | Origin | Dest | Cap1st | CapBusiness | CapEcon |
|-----|----------|---------|--------|------|--------|-------------|---------|
| 1   | IA00100  | 0000101 | RDU    | LHR  | 14     | 30          | 163     |
| 2   | IA00201  | 0000002 | LHR    | RDU  | 14     | 30          | 163     |
| 3   | IA00300  | 0000003 | RDU    | FRA  | 14     | 30          | 163     |
| 4   | IA00400  | 0000400 | FRA    | RDU  | 14     | 30          | 163     |
| 5   | IA00500  | 0000035 | RDU    | JFK  | 16     | .           | 251     |

### How SAS Updates Data Using the MODIFY and BY Statements

The following steps show how SAS updates values in the master data set (**Capacity**) based on values in the transaction data set (**Sasuser.Newrtnum**).

1. The following program is submitted to SAS:

```
data capacity;
  modify capacity sasuser.newrtnum;
  by flightid;
run;
```

2. The MODIFY statement reads the BY variable FlightID from the first observation in the transaction data set (**Sasuser.Newrtnum**) into a memory buffer.
3. SAS builds a dynamic WHERE statement (where flightid = 'IA00100';).
4. SAS then applies the dynamic WHERE statement to the master data set (**Capacity**) and reads the selected observation into the PDV.
5. The MODIFY statement then reads the first observation from the transaction data set (**Sasuser.Newrtnum**) into the PDV, overlaying common variables to update the master data set. Only RouteID changes.
6. An implied REPLACE statement writes the updated observation in the PDV back over the same observation in the master data set. The process repeats until end of file in the transaction dataset.



For more information about the MODIFY and BY statements, see the SAS documentation.

Which code below correctly uses the transaction data set **Hr.NewSalary** to update the master data set **Hr.Salary** by EmployeeID?

- Q.2.**
- ```
data hr.salary;
  modify hr.salary hr.newsalary;
  by employeeid;
run;
data hr.salary;
  modify hr.newsalary hr.salary;
  by employeeid;
run;
```

## Handling Duplicate Values

When you use the MODIFY and BY statements to update a data set, WHERE processing starts at the top of the master data set and finds the first match and updates it. Let's take a look at what happens if there are duplicate values in the master or transaction data sets. Suppose you have the following code to make updates to the master data set **M** using the transaction data set **T**:

```
data m;
  modify m t;
  by a;
run;
```

If duplicate values of the BY variable exist in the **master data set**, only the first observation in the group of duplicate values is updated because WHERE processing begins at the top of the data set and updates the first match.

T		M		M updated	
A	B	A	B	A	B
1	30	1	20	1	30
2	32	1	21	1	21
		2	26	2	32

If duplicate values of the BY variable exist in the **transaction data set**, the duplicate values overwrite each other so that the last value in the group of duplicate transaction values is the result in the master data set.

T		M		M updated	
A	B	A	B	A	B
1	30	1	20	1	38
1	38	2	26	2	26
		3	22	3	22

You can avoid overwriting duplicate values by writing an accumulation statement so that all the observations in the transaction data set are added to the master observation.



If duplicate values exist in both the **master and transaction data sets**, you can use PROC SQL to apply the duplicate values in the transaction data set to the duplicate values in the master data set in a one-to-one correspondence.

## Handling Missing Values

If there are missing values in the transaction data set, SAS does not replace the data in the master data set with missing values unless they are special missing values.



A special missing value is a type of numeric missing value that enables you to represent different categories of missing data by using the letters A-Z or an underscore. You designate special missing values using the MISSING statement in the DATA step. For more information, see the SAS documentation.

You can specify how missing values in the transaction data set are handled by using the UPDATEMODE= option in the MODIFY statement.

### General form, MODIFY statement with the UPDATEREAD= option:

```
MODIFY master-data-set transaction-data-set  
UPDATEREAD=MISSINGCHECK | NOMISSINGCHECK;
```

where

- *master-data-set* is the name of the SAS data set that you want to modify.
- *transaction-data-set* is the name of the SAS data set in which the updated values are stored.
- **MISSINGCHECK** prevents missing values in the transaction data set from replacing values in the master data set unless they are special missing values. MISSINGCHECK is the default.
- **NOMISSINGCHECK** allows missing values in the transaction data set to replace the values in the master data set. Special missing values in the transaction data set still replace values in the master data set.

Given the following code and data set **Transaction**, what happens if the UPDATEREAD= option is set to its default setting?

**Q3.**

```
data master;  
modify master transaction;  
by a;  
run;
```

A	B	C
1	50	20
2	55	.
4	.	.

Missing values in B and C **will not** replace the corresponding values in the data set **Master**.

Missing values in B and C **will** replace the corresponding values in the data set **Master**.

### Modifying Observations Located by an Index

You have learned that you can use a BY statement to access values you want to update in a master data set by matching. When you have an indexed data set, you can use the index to directly access the values you want to update. To do this, you use

- a MODIFY statement with the KEY= option to name an indexed variable to locate the observations for updating
- another data source (typically a SAS data set named on a SET statement or an external file read by an INPUT statement) to provide a like-named variable whose values are supplied to the index.

### General form, MODIFY statement with the KEY= option:

```
MODIFY SAS-data-set KEY=index-name;
```

where

- *SAS-data-set* is the master data set, or the data set that you want to update
- *index-name* is the name of the simple or composite index that you are using to locate observations.

Updating with an index is different from updating using a BY statement. When you use the MODIFY statement with the KEY= option to name an index,

- you must **explicitly specify the update** that you want to occur. No automatic overlay of non-missing values in the transaction data set occurs as it does with the MODIFY/BY method.

- each observation in the transaction data set **must have a matching observation** in the master data set. If you have multiple observations in the transaction data set for one master observation, only the first observation in the transaction data set is applied. The other observations generate runtime errors and terminate the DATA step (unless you use the UNIQUE option, which is discussed later in this lesson).

### Example

Suppose that airline cargo weights for 1999 are stored in the master data set **Cargo99**, which has a composite index named **FlightDte** on the variables FlightID and Date. Some of the data is incorrect and the data set needs to be updated. The correct cargo data is stored in the transaction data set **Newcgnum**.

In the program below, the KEY= option specifies the **FlightDte** index. When a matching observation is found in **Cargo99**, three variables (CapCargo, CargoWgt, and CargoRev) are updated.



If you choose to run this example, you must copy the data set **Cargo99** from the **Sasuser** library to the **Work** library.

```
proc print data=cargo99(obs=5);
run;
data cargo99;
  set sasuser.newcgnum (rename =
    (capcargo = newCapCargo
     cargowgt = newCargoWgt
     cargorev = newCargoRev));
  modify cargo99 key=flightdte;
  capcargo = newcapcargo;
  cargowgt = newcargowgt;
  cargorev = newcargorev;
run;
proc print data=cargo99(obs=5);
run;
```

The output below shows the first five observations of the SAS data set **Cargo99** before it was modified by **Newcgnum**.

Obs	FlightID	RouteID	Origin	Dest	CapCargo	Date	CargoWgt	CargoRev
1	IA00100	0000001	RDU	LHR	82400	01JAN1999	45600	\$111,720.00
2	IA00100	0000001	RDU	LHR	82400	01AUG1999	44600	\$109,270.00
3	IA00100	0000001	RDU	LHR	82400	20AUG1999	44600	\$109,270.00
4	IA00100	0000001	RDU	LHR	82400	02SEP1999	47400	\$116,130.00
5	IA00100	0000001	RDU	LHR	82400	29DEC1999	44200	\$108,290.00

The output below shows the first five observations of the SAS data set **Cargo99** after it was modified by **Newcgnum**. Notice that the three variables in the first observation were updated by the values in **Newcgnum**.

Obs	FlightID	RouteID	Origin	Dest	CapCargo	Date	CargoWgt	CargoRev
1	IA00100	0000001	RDU	LHR	35055	01JAN1999	.	\$121,879.90

Obs	FlightID	RouteID	Origin	Dest	CapCargo	Date	CargoWgt	CargoRev
2	IA00100	0000001	RDU	LHR	82400	01AUG1999	44600	\$109,270.00
3	IA00100	0000001	RDU	LHR	82400	20AUG1999	44600	\$109,270.00
4	IA00100	0000001	RDU	LHR	82400	02SEP1999	47400	\$116,130.00
5	IA00100	0000001	RDU	LHR	82400	29DEC1999	44200	\$108,290.00

### How SAS Updates Data with the MODIFY Statement and KEY= Option

The flowing steps shows how SAS locates observations by using an index, then updates values in the master data set (**Cargo99**) based on values in the transaction data set (**Sasuser.Newcgnum**).

1. The following program is submitted to SAS:

```
data cargo99;
  set sasuser.newcgnum (rename =
    (capcargo = newCapCargo
     cargowgt = newCargoWgt
     cargorev = newCargoRev));
  modify cargo99 key=flghtdte;
  capcargo = newcapcargo;
  cargowgt = newcargowgt;
  cargorev = newcargorev;
run;
```

2. First, the SET statement reads an observation for the transaction data set (**Sasuser.Newcgnum**).
3. The KEY= option causes the MODIFY statement to use the values in FlightID and Date to access an observation in the master data set using the index **FlghtDte**. SAS reads the observation with matching index values into the master PDV.
4. The assignment statement updates CapCargo, CargoWgt, and CargoRev.
5. An implied REPLACE statement writes the updated observation back over the same observation in **Cargo99**. The process repeats until the end of file is reached in **Sasuser.Newcgnum**.



For more information about using the MODIFY statement with the KEY= option, see the SAS documentation.

Which statement about using the MODIFY statement and KEY= option is **false**?

**Q.4.**

Your program must specify the update to perform on values located by the index.

If the index specified on the KEY= option does not exist, SAS creates a simple index using the Obs variable.

The index specified on the KEY= option can be either a simple or composite index.

### Handling Duplicate Values

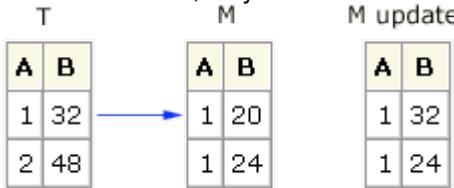
When you use an index to locate values to update, duplicate values of the indexed variable in the transaction data set may cause problems. Let's look at what happens with various scenarios when you use the following code to update the master data set **M** with values from the transaction data set **T**. The index on the **M** data set is built on the variable A:

```

data m;
set t (rename=(b=newb));
modify m key=index;
b=newb;
run;

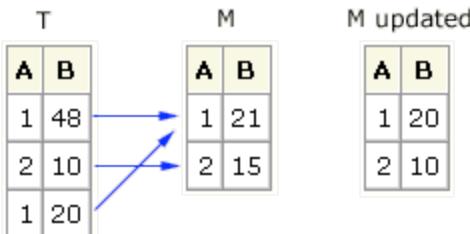
```

If there are **duplications in the master data set**, only the first occurrence is updated.

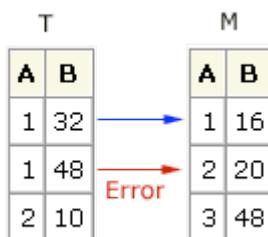


**i** If you want all duplicates in the master data set to be updated with the transaction value, use a DO loop to execute a SET statement with the KEY= option multiple times.

If there are **nonconsecutive duplications in the transaction data set**, SAS updates the first observation in the master data set. The last duplicate transaction value is the result in the master data set after the update.



If there are **consecutive duplications in the transaction data set**, some of which do not have a match in the master data set, then SAS performs a one-to-one update until it finds a non-match. At that time, the DATA step terminates with an error.



Adding the **UNIQUE** option to the MODIFY statement allows you to avoid the error in the DATA step. The UNIQUE option causes the DATA step to return to the top of the index each time it looks for a match for the value from the transaction data set. The UNIQUE option can only be used with the KEY= option.

#### General form, MODIFY statement with the UNIQUE option:

**MODIFY SAS-data-set KEY=index-name / UNIQUE;**

where

- *SAS-data-set* positions the input pointer on a specified column
- *index-name* is the name of the variable that is being created.

You can specify the **UNIQUE** option to

- apply multiple transactions to one master observation
- identify that each observation in the master data set contains a unique value of the index variable.

When you use the UNIQUE option and there are consecutive duplications in the transaction data set, SAS updates the first observation in the master data set. This is similar to what happens when you have nonconsecutive duplications in the transaction data set. If the values in the transaction data set should be added to the value in the master data set, you can write a statement to accumulate the values from all the duplicates.

T		M		M updated	
A	B	A	B	A	B
1	32	1	16	1	48
1	48	2	20	2	10
2	10	3	48	3	48

Which statement about using the UNIQUE option is **true**?

You can use the UNIQUE option with the KEY= option or the BY statement to change how SAS processes duplicate values.

## Q.5.

You can use the UNIQUE option when you want to apply multiple transactions to one master observation.

You can use the UNIQUE option when you have consecutive duplicate values in the master data set.

### Controlling the Update Process

The way SAS writes observations to a SAS data set when the DATA step contains a MODIFY statement depends on whether certain other statements are present. If no other statements are present, SAS writes the current observation to its original place in the SAS data set. This action occurs by default through an implied REPLACE statement at the bottom of the DATA step.

However, you can override this default behavior by explicitly adding the **OUTPUT**, **REPLACE**, or **REMOVE** statement.

#### General form for OUTPUT, REPLACE, and REMOVE statements:

```
OUTPUT;
REPLACE;
REMOVE;
```

where

- **OUTPUT** specifies that the current observation be written to the **end** of the data set
- **REPLACE** specifies that the current observation be rewritten to the **same location** in the data set
- **REMOVE** specifies that the current observation be **deleted** from the master data set.

Using OUTPUT, REPLACE, or REMOVE in a DATA step overrides the default replacement of observations. If you use any one of these statements in a DATA step, you must explicitly program each action that you want to take. You can use these three statements together as long as the sequence is logical.



If you use an OUTPUT statement in conjunction with a REPLACE or REMOVE statement, be sure the OUTPUT statement is executed **after** any REPLACE or REMOVE statements to ensure the integrity of the index position.

### Example

If the SAS data set **Transaction** has a variable named code having values of *yes*, *no*, and *new*, you can submit the following program to

```
delete rows where the value of code is no
update rows where the value of code is yes
append rows where the value of code is new.
data master;
  set transaction;
  modify master key = id;
  a = b;
  if code = 'no' then remove;
  else if code = 'yes' then replace;
  else if code = 'new' then output;
run;
```



You cannot run this example because **Transaction** and **Master** are fictitious data sets.

Which of these statements would you use with the MODIFY statement if you wanted to append observations from a transaction data set to the master data set?

**Q.6.**

APPEND

OUTPUT

REMOVE

## Monitoring I/O Error Conditions

When you use the MODIFY statement with a BY statement or KEY= option to update a data set, error checking is important for several reasons. The most important reason is that these tools use nonsequential access methods, so there is no guarantee that an observation will be located that satisfies the request. Error checking enables you to perform updates or not, depending on the outcome of the I/O condition.

The **automatic variable \_IORC\_** (Input Output Return Code) is created when you use the MODIFY statement with the BY statement or KEY= option. The value of \_IORC\_ is a numeric return code that indicates the status of the most recently executed I/O operation. Checking the value of this variable allows you to detect abnormal I/O conditions and direct execution in particular ways rather than having the application terminate abnormally.

### Using \_IORC\_ with %SYSRC

Because the values of the \_IORC\_ automatic variable are internal and subject to change, **%SYSRC**, an autocall macro, was created to enable you to test for **specific I/O conditions** while protecting your code from future changes in \_IORC\_ values.

**General form, \_IORC\_ with the %SYSRC autocall macro:**

**IF \_IORC\_= %SYSRC(*mnemonic*) THEN...**

where *mnemonic* is a code for a specific I/O condition.



%SYSRC is in the autocall library. You must have the MACRO system option in effect to use this macro. To learn more about using autocall macros, see the lesson **Storing Macro Programs**

When you use %SYSRC, you can check the value of \_IORC\_ by specifying one of the mnemonics listed in the table below.

Mnemonic	Meaning
_DSENMR	The observation in the transaction data set does not exist in the master data set (used only with the MODIFY and BY statements).
_DSEMTR	Multiple transaction data set observations do not exist on the master data set (used only with the MODIFY and BY statements).
_DSENOM	No matching observation (used with the KEY= option).
_SOK	The observation was located. _SOK has a value of 0.

**Example**

Suppose you are using the MODIFY statement with the KEY= option to update a SAS data set. In the program below, when \_IORC\_ has the value \_SOK, the observation is updated. When \_IORC\_ has the value \_DSENOM, no matching observation is found, so the observation is appended to the data set by the OUTPUT statement and \_ERROR\_ is reset to 0 in the do loop.

```
data master;
  set transaction;
  modify master key = id;
  if _IORC_= %sysrc(_sok) then
    do;
      a = b;
      replace;
    end;
  else
    if _IORC_= %sysrc(_drenom) then
      do;
        output;
        _ERROR_ = 0;
      end;
  run;
```



For more information about the \_IORC\_ automatic variable and %SYSRC, see information about error-checking tools in the SAS documentation.

**Q7.** Complete this IF statement by using \_IORC\_ and %SYSRC to execute statements **only** when a

matching observation is found in the master data set.

if \_\_\_\_\_ then do;

## Understanding Integrity Constraints

Now that you know how to modify data in place, you may be wondering how you can protect or insure the integrity of your data when it is modified. Integrity constraints are rules that you can specify in order to restrict the data values that can be stored for a variable in a data set. SAS enforces integrity constraints when values associated with a variable are added, updated, or deleted using techniques that modify data in place, such as

- a DATA step with the MODIFY statement
- an interactive data editing window
- PROC SQL with the INSERT INTO, SET, or UPDATE statements
- PROC APPEND.

When you place integrity constraints on a SAS data set, you specify the type of constraint that you want to create. Each constraint has a different action.

Type	Action
CHECK	ensures that a specific set or range of values are the only values in a column. It can also check the validity of a value in one column based on a value in another column within the same row.
NOT NULL	guarantees that a column has non-missing values in each row.
UNIQUE	enforces uniqueness for the values of a column.
PRIMARY KEY	uniquely defines a row within a table, which can be a single column or a set of columns. A table can have only one primary key. The PRIMARY KEY constraint includes the attributes of the NOT NULL and UNIQUE constraints.
FOREIGN KEY	links one or more rows in a table to a specific row in another table by matching a column or set of columns in one table with the PRIMARY KEY defined in another table. This parent/child relationship limits modifications made to both tables. The only acceptable values for a FOREIGN KEY are values of the PRIMARY KEY or missing values.



When you add an integrity constraint to a table that contains data, SAS checks all data values to determine whether they satisfy the constraint before the constraint is added.

You can use integrity constraints in two ways, **general** and **referential**. General constraints operate within a data set, and referential constraints operate between data sets.

### General Integrity Constraints

General integrity constraints enable you to restrict the values of variables within a single data set. The following four integrity constraints can be used as general integrity constraints:

- CHECK
- NOT NULL
- UNIQUE
- PRIMARY KEY.



A PRIMARY KEY constraint is a general integrity constraint as long as it does not have any FOREIGN KEY constraints referencing it. When PRIMARY KEY is used as a general constraint it is simply a shortcut for assigning the NOT NULL and UNIQUE constraints.

### Referential Integrity Constraints

Referential constraints enable you to link the data values of a column in one data set to the data values of columns in another data set. You create a referential integrity constraint when a FOREIGN KEY integrity constraint in one data set references a PRIMARY KEY integrity constraint in another data set. To create a referential integrity constraint, you must follow two steps:

- Define a PRIMARY KEY constraint on the first data set.
- Define a FOREIGN KEY constraint on other data sets.

Which integrity constraint would you place on the variable RainfallAmt to ensure that values are greater than or equal to zero?

**Q.8.**

CHECK  
UNIQUE  
PRIMARY KEY

### Placing Integrity Constraints on a Data Set

Integrity constraints can be created using

- the DATASETS procedure
- the SQL procedure.

Although you can use either procedure to create integrity constraints on existing data sets, you must use PROC SQL if you want to create integrity constraints at the same time that you create the data set. In this lesson you learn to use PROC DATASETS to place integrity contraints on an existing data set.

#### General form, DATASETS procedure with the IC CREATE statement:

```
PROC DATASETS LIB=libref <NOLIST>;
  MODIFY SAS-data-set;
  IC CREATE constraint-name=constraint
    <MESSAGE='Error Message'>;
  QUIT;
```

- where *libref* is the library in which the data set is stored. If you do not specify the LIB= option, the procedure uses the Work library.
- **NOLIST** suppresses the directory listing
- *SAS-data-set* is the name of the data set to which you want to apply the integrity constraint
- *constraint-name* is any name that you wish to give the integrity constraint
- *constraint* is the type of constraint that you are creating, specified in the following format:
- **NOT NULL** (*variable*)
- **UNIQUE** (*variables*)
- **CHECK** (*where-expression*)
- **PRIMARY KEY** (*variables*)
- **FOREIGN KEY** (*variables*) **REFERENCES** *table-name*
- *Error Message* is an optional message that you want the user to see if the constraint is violated.



You can use **IC** or **INTEGRITY CONSTRAINT** interchangeably.



To learn how to create integrity constraints using the SQL procedure, see the lesson [Creating and Managing Tables Using PROC SQL](#).

### Example

Suppose you have a data set that contains route information and passenger capacity for each class in an airline. You need to create integrity constraints to ensure that when the data set is updated

- the route ID number is both unique and required (PRIMARY KEY)
- the capacity for business class passengers must either be missing or be greater than the capacity for first class passengers (CHECK).

In the code below, the IC CREATE statement is used to create two general integrity constraints on variables in the data set **Capinfo**:

- The PRIMARY KEY constraint is placed on the **Routeid** variable. This constraint ensures that when values of **Routeid** are updated, they must be unique and nonmissing.



The same effect could be achieved by applying both the UNIQUE and NOT NULL constraints, but the PRIMARY KEY constraint is used as a shortcut.

- The CHECK constraint uses the WHERE expression to ensure that the only values of **CapBusiness** that are allowed are those greater than **Cap1st** or missing.



If you choose to run this example, you must copy the data set **Capinfo** from the **Sasuser** library to the **Work** library.

```
proc datasets nolist;
  modify capinfo;
  ic create PKIDInfo=primary key(routeid)
    message='You must supply a Route ID Number';
  ic create Class1=check(where=(cap1st<capbusiness or capbusiness=.))
    message='Cap1st must be less than CapBusiness';
quit;
```

Notice that the NOLIST option is used to prevent a listing of the Work directory that PROC DATASETS generally produces. When the constraint is created, a message is written to the SAS log.

SAS Log

```
45  modify capinfo;
46  ic create PKIDInfo = primary key (routeid)
47  message = 'You must supply a Route ID Number';
NOTE: Integrity constraint PKIDInfo defined.
48  ic create Class1 = check (where = (cap1st < capbusiness
49  or capbusiness = .)) message = 'Cap1st must be less
50  than CapBusiness';
NOTE: Integrity constraint Class1 defined.
51  run;
```



For the UNIQUE and PRIMARY KEY constraints, SAS builds indexes on the columns involved if an appropriate index does not already exist. Any index created by an integrity constraint can be used for other purposes, such as WHERE processing or the KEY= option in a SET statement.



For more information about creating integrity constraints, see the SAS documentation for the DATASETS procedure.

Write a statement to create an integrity constraint on the variable gender that limits the values of the variable to either *M* or *F*. Name the integrity constraint val\_gender.

**Q.9.**

```
proc datasets lib=mylib nolist;
  modify mydata;
  -----
  quit;
```

## How Constraints Are Enforced

Once integrity constraints are in place, SAS enforces them whenever you modify values in the data set in place. Techniques for modifying data in place include using

- a DATA step with the MODIFY statement
- interactive data editing windows
- PROC SQL with the INSERT INTO, SET, or UPDATE statements
- PROC APPEND.

### Example

The code in the previous example placed a check constraint on Cap1st and CapBusiness to ensure that values for the capacity in business class were either greater than first class or missing. Suppose that you ran the following program to triple the capacity in first class. This would probably violate the check constraint for some observations.

```
data capinfo;
  modify capinfo;
  cap1st=cap1st*3;
run;
```

The observations that failed to pass the integrity constraint are written to the SAS log. As you can see, all these observations would have had values of Cap1st greater than those of CapBusiness.

### SAS Log

```
FlightID=IA00100 RouteID=0000001 Origin=RDU Dest=LHR Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130_N_=1
FlightID=IA00201 RouteID=0000002 Origin=LHR Dest=RDU Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130_N_=2
FlightID=IA00300 RouteID=0000003 Origin=RDU Dest=FRA Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130_N_=3
FlightID=IA00400 RouteID=0000004 Origin=FRA Dest=RDU Cap1st=42
```

```
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130_N_=4
FlightID=IA02900 RouteID=0000029 Origin=SFO Dest=HNL Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130_N_=29
FlightID=IA03000 RouteID=0000030 Origin=HNL Dest=SFO Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130_N_=30
FlightID=IA03300 RouteID=0000033 Origin=RDU Dest=ANC Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130_N_=33
FlightID=IA03400 RouteID=0000034 Origin=ANC Dest=RDU Cap1st=42
CapBusiness=30 CapEcon=163
_ERROR_=1 _IORC_=660130_N_=34
NOTE: There were 50 observations read from the data set WORK.CAPINFO.
NOTE: The data set WORK.CAPINFO has been updated. There were 42
observations rewritten, 0 observations added and 0
observations deleted.
NOTE: There were 8 rejected updates, 0 rejected adds, and 0 rejected
deletes.
```

If you used the VIEWTABLE window or another window to make this update, SAS would have displayed the error message defined for the integrity constraint.



Rejected observations can be collected in a special file using the audit trail functionality that you will learn about later in this lesson.

### Copying a Data Set and Preserving Integrity Constraints

The APPEND, COPY, CPRT, CIMPORT, and SORT procedures preserve integrity constraints when their operation results in a copy of the original data file. Integrity constraints are also preserved if you copy a data set using the SAS Explorer window.



For more information about preserving integrity constraints, see the SAS documentation.



Rejected observations can be collected in a special file using the audit trail functionality that you will learn about later in this lesson.

### Copying a Data Set and Preserving Integrity Constraints

The APPEND, COPY, CPRT, CIMPORT, and SORT procedures preserve integrity constraints when their operation results in a copy of the original data file.



For more information about preserving integrity constraints, see the SAS documentation.

### Documenting Integrity Constraints

To view the descriptor portion of your data, including the integrity constraints that you have placed on a data set, you can use the CONTENTS statement in the DATASETS procedure.

### General form, DATASETS procedure with the CONTENTS statement:

```
PROC DATASETS LIB=libref <NOLIST>;
  CONTENTS DATA=SAS-data-set;
  QUIT;
```

where

- *libref* is the library in which the data set is stored
- **NOLIST** suppresses the directory listing
- **SAS-data-set** is the name of the data set that you want information about.



The CONTENTS statement in the DATASETS procedure results in the same information as the CONTENTS procedure.

### Example

The following code displays information about the **Capinfo** data set, including the integrity constraints that were added to this data set in the last example. Notice that the NOLIST option is used here to suppress the listing of all data sets in the Work library. With this option, only the information for the **Capinfo** data set is listed.

```
proc datasets nolist;
  contents data=capinfo;
  quit;
```

Only the integrity constraints portion of the output is shown below.

Alphabetic List of Integrity Constraints					
#	Integrity Constraint	Type	Variables	Where Clause	User Message
1	Class1	Check		(Cap1st<CapBusiness) or (CapBusiness=-.)	First Class Capacity must be less than Business Capacity
2	PKIDInfo	Primary Key	RouteID		You must supply a Route ID Number

### Removing Integrity Constraints

To remove an integrity constraint from a data set, use the DATASETS procedure with the IC DELETE statement.

### General form, DATASETS procedure with the IC DELETE statement:

```
PROC DATASETS LIB=libref <NOLIST>;
  MODIFY SAS-data-set;
  IC DELETE constraint-name;
  QUIT;
```

where

- *libref* is the name of the library in which the data set is stored. If you do not specify the LIB= option, the procedure uses the **Work** library.
- **NOLIST** suppresses the directory listing
- **SAS-data-set** is the name of the data set that has the integrity constraint

- *constraint-name* is the name of the integrity constraint that you want to delete.

### Example

The code below removes the integrity constraints on the **Capinfo** data set:

```
proc datasets;
  modify capinfo;
  ic delete pkidinfo;
  ic delete class1;
quit;
```

A message is written to the SAS log when the integrity constraint is deleted.

#### SAS Log

```
53  modify capinfo;
54  ic delete pkidinfo;
NOTE: Integrity constraint PKIDInfo deleted.
55  ic delete class1;
NOTE: All integrity constraints defined on WORK.CAPINFO.DATA
have been deleted.
56  run;
```

### Understanding Audit Trails

As you modify a data set, you may want to track the changes that you make by using an audit trail. An audit trail is an optional SAS file that logs modifications to a SAS table. Audit trails are used to track changes that are made to the data set in place. Specifically, audit trails track changes made with

- the VIEWTABLE window
- the data grid
- the MODIFY statement in the DATA step
- the UPDATE, INSERT, or DELETE statement in PROC SQL.

For each addition, deletion, and update to the data, the audit trail automatically stores a copy of the variables in the observation that was updated, and information such as who made the modification, what was modified, and when the modification was made. It can also store additional information in user-defined variables.

The following PROC CONTENTS output lists the variables in an audit trail file for a data set that has two variables, A and B. You will learn more about these variables later in this lesson.

#	Variable	Type	Len	Pos	Format	Label
1	A	Num	8	0		
2	B	Num	8	8		
3	Who	Char	20	16		Name
4	Why	Char	20	36		Reason
5	_ATDATETIME_	Num	8	56	DATETIME19.	

#	Variable	Type	Len	Pos	Format	Label
10	_ATMESSAGE_	Char	8	114		
6	_ATOBSNO_	Num	8	64		
9	_ATOPCODE_	Char	2	112		
7	_ATRETURNCODE_	Num	8	72		
8	_ATUSERID_	Char	32	80		



Any procedure or action that replaces the data set (such as the DATA step, CREATE TABLE in PROC SQL, or SORT without the OUT= option) will delete the audit trail. Audit trails should not be deleted with system tools such as Windows Explorer.

A SAS table can have only one audit trail file. The audit trail file

- is a read-only file
- is created by PROC DATASETS
- must be in the same library as the data file associated with it
- has the same name as the data set it is monitoring, but with a member type of AUDIT.

Next let's take a look at how you initiate an audit trail on a SAS data set.

## Initiating and Reading Audit Trails

You initiate an audit trail using the DATASETS procedure with the **AUDIT** and **INITIATE** statements.

**General form, DATASETS procedure to initiate an audit trail:**

```
PROC DATASETS LIB=libref <NOLIST>;
  AUDIT SAS-data-set <SAS-password>;
  INITIATE;
  RUN;
  QUIT;
```

where

- *libref* is the name of the library where the data set to be audited resides
- **NOLIST** suppresses the directory listing
- *SAS-data-set* is the name of the SAS data set that you want to audit
- *SAS-password* is the SAS data set password, if one exists
- **INITIATE** begins the audit trail on the data set specified in the AUDIT statement.

## Example

The following code initiates an audit trail on the data set **Capinfo**.



If you choose to run this example, you must copy the data set **Capinfo** from the **Sasuser** library to the **Work** library.

```
proc datasets nolist;
  audit capinfo;
  initiate;
  quit;
```

SAS Log

```
60 audit capinfo;
61 initiate;
WARNING: The audited data file WORK.CAPINFO.DATA is not
    password protected.
    Apply an ALTER password to prevent accidental
    deletion or replacement of it and any associated
    audit files.
62 run;
NOTE: The data set WORK.CAPINFO.AUDIT has 0 observations
    and 13 variables.
```



The audit trail file uses the SAS password that is assigned to the parent data set; therefore, it is recommended that you alter the password for the parent data set. Use the ALTER= data set option to assign an alter-password to a SAS data set or to access a read-, write-, or alter-protected SAS data set. If another password is used or no password is used, then the audit file is still created, but is not protected.



For more information about audit trails, see the SAS documentation for the DATASETS procedure.

### Reading Audit Trail Files

When the audit trail is initiated, it has no observations until the first modification is made to the audited data set. When the audit trail file contains data, you can read it with any component of SAS that reads a data set. To refer to the audit trail file, use the TYPE= data set option.

**General form, TYPE= data set option to specify an audit file:**  
**(TYPE=AUDIT)**

### Examples

The following PROC CONTENTS code displays the contents of the audit trial file:

```
proc contents data=mylib.sales (type=audit);
run;
```

The following PROC PRINT code lists the data in the audit trail file for the data set **Capinfo**:

```
proc print data=capinfo (type=audit);
run;
```

Which code begins an audit trail on the SAS data set **Sasuser.Financial**?

**Q.10.**

```
proc datasets nolist;
  audit financial;
  initiate;
  quit;
  proc datasets lib=sasuser nolist;
    initiate audit financial;
    quit;
  proc datasets lib=sasuser nolist;
    audit financial;
```

```
initiate;  
quit;
```

## Controlling Data in the Audit Trail

Now that you have seen how to initiate audit trails and read an audit trail file, let's look more closely at the information the audit trail file contains. The audit trail file can contain three types of variables:

- **data set variables** that store copies of the columns in the audited SAS data set
- **audit trail variables** that automatically store information about data modifications
- **user variables** that store user-entered information.

	#	Variable	Type	Len	Pos	Format	Label
Data Set Variables	1	A	Num	8	0		
	2	B	Num	8	8		
User Variables	3	Who	Char	20	16		Name
	4	Why	Char	20	36		Reason
Audit Trail Variables	5	_ATDATETIME_	Num	8	56	DATETIME19.	
	10	_ATMESSAGE_	Char	8	114		
	6	_ATOBSNO_	Num	8	64		
	9	_ATOPCODE_	Char	2	112		
	7	_ATRETURNCODE_	Num	8	72		
	8	_ATUSERID_	Char	32	80		

You can use additional statements in the PROC DATASETS step to control which variables appear in the audit trail. Let's take a closer look at each of the three types of variables that can be found in an audit trail.

### Data Set Variables

As you might expect, the audit trail file has the same set of variables that are in the audited data set. If the data set contains the variables A and B, the variables A and B are also in the audit trail file.

Next let's take a look at the audit trail variables that automatically store information about changes that you make to the data.

### Audit Trail Variables

Audit trail variables automatically store information about data modifications. Audit trail variable names begin with AT followed by a specific string, such as DATETIME.

Audit trail variable	Information stored
_ATDATETIME_	date and time of a modification
_ATUSERID_	login user ID associated with a modification
_ATOBSNO_	observation number affected by the modification unless REUSE=YES
_ATRETURNCODE_	event return code
_ATMESSAGE_	SAS log message at the time of the modification
_ATOPCODE_	code describing the type of operation

## Values of the \_ATOPCODE\_ Variable

The \_ATOPCODE\_ variable contains a code that describes the type of operation that wrote the observation to the audit file. For example, if you modified all observations in an audited data set, the audit file would contain twice as many observations as the original data set. The audit file would contain one observation that matched the original observation with an \_ATOPCODE\_ value of *DR*, and one updated observation with an \_ATOPCODE\_ value of *DW*.

The table below shows the possible values of the \_ATOPCODE\_ variable.

_ATOPCODE_ value	Event
DA	added data record image
DD	deleted data record image
DR	before-update record image
DW	after-update record image
EA	observation add failed
ED	observation delete failed
EU	observation update failed

You can define what information is stored in the audit file by using the LOG statement when you initiate the audit trail.

## Using the LOG Statement to Control the Data in the Audit Trail

When you initiate an audit trail, options in the LOG statement determine the type of entries stored in the audit trail, along with their corresponding \_ATOPCODE\_ values. The ERROR\_IMAGE option controls E operation codes. The BEFORE\_IMAGE option controls the DR operation code, and the DATA\_IMAGE option controls all other D operation codes. If you omit the LOG statement when you initiate the audit trail, the default behavior is to log all images.

### General form, LOG statement:

**LOG <audit-settings>;**

- where *audit-settings* are any of the following:
  - BEFORE\_IMAGE=YES|NO controls storage of before-update record images (the 'DR' operation)
  - DATA\_IMAGE=YES|NO controls storage of after-update record images (for example, other operations starting with 'D')
  - ERROR\_IMAGE=YES|NO controls storage of unsuccessful update record images (for example, operations starting with 'E').

### Example

The following code initiates an audit trail on the data set **Capinfo** but stores only error record images. This means that the audit file will contain only records where an error occurred. The \_ATOPCODE\_ values can be only *EA*, *ED*, and *EU*.



If you choose to run this example, you must copy the data set **Capinfo** from the **Sasuser** library to the **Work** library.

```
proc datasets nolist;
  audit capinfo;
  initiate;
  log data_image=NO before_image=NO;
quit;
```

What happens if you omit the LOG statement when the audit trail is initiated?

- Q11.** All \_ATOPCODES\_ codes are logged.  
No \_ATOPCODES\_ codes are logged.

If an audit trail is initiated with the following code, what is true about the audit trail file?

- Q12.**
- ```
proc datasets nolist;
  audit mydata;
  initiate;
  log error_image=no data_image=no;
quit;
```
- For each modification, the audit trail file contains only a record of each observation before the data set was updated.  
For each modification, the audit trail file contains a record of the observation before and after the data set was updated.  
For each modification, the audit trail file contains only a record of any errors that occurred when the data set was updated.

## User Variables

User variables allow the person editing the file to enter information about changes they are making to the data. Although the data values are stored in the audit file, you can update them in the data set like any other variable.

User variables are created by using the USER\_VAR statement in the audit trail specification.

### General form, USER\_VAR statement:

```
USER_VAR variable-name <$><length><LABEL='variable-label'>;
```

where

- *variable-name* is the name of the user variable you are creating
- \$ indicates the variable is a character variable
- *length* specifies the length of the variable (the default is 8)
- *variable-label* specifies a label for the variable enclosed in quotation marks.



You can create more than one user variable in a single USER\_VAR statement.

User variables are unique in SAS in that they are stored in one file (the audit file) and opened for update in another file (the data set). When the data set is opened for update, the user variables display, and you can edit them as though they are part of the data set.

## Example

Suppose you must monitor the updates for the data set **Capinfo**. The following code initiates an audit trail for the data set **Capinfo** and creates two user variables, who and why, to store who made changes to the data set and why the changes were made.



If you choose to run this example, you must copy the data set **Capinfo** from the **Sasuser** library to the **Work** library.

```
proc datasets nolist;
  audit capinfo;
  initiate;
  user_var who $20 label = 'Who made the change'
            why $20 label = 'Why the change was made';
quit;
```

Once these user variables are set up, they are retrieved from the audit trail and displayed when the data set is opened for update. You can enter data values for the user variables as you would for any data variable. The data values are saved to the audit trail as each observation is saved. The user variables are not available when the data is opened for browsing or printing. To rename a user variable or modify its attributes, you modify the data set, not the audit file.

- Q.13.** Write a statement to create a user variable named admin that records information entered about why a modification was made to the file. Give the variable a length of 20 and label the variable *Reason for update*.
- 

## Controlling the Audit Trail

Once you activate an audit trail, you can suspend and resume logging, and terminate (delete) the audit trail by resubmitting a PROC DATASETS step with additional statements. You use the DATASETS procedure to suspend and then resume the audit trail. You also use this procedure to delete or terminate an audit trail.

**General form, DATASETS procedure to suspend, resume, or terminate an audit trail:**

```
PROC DATASETS LIB=libref <NOLIST>;
  AUDIT SAS-data-set <SAS-password>;
  SUSPEND | RESUME | TERMINATE;
  QUIT;
```

where

- *libref* is the name of the library where the table to be audited resides
- **NOLIST** suppresses the directory listing
- *SAS-data-set* is the name of the SAS data set that you want to audit
- *SAS-password* is the SAS data file password, if one exists
- **SUSPEND** suspends event logging to the audit file, but does not delete the audit file
- **RESUME** resumes event logging to the audit file, if it was suspended
- **TERMINATE** terminates event logging and deletes the audit file.



Because each update to the data file is also written to the audit file, the audit trail can negatively impact system performance. You may want to consider suspending the audit trail for large, regularly scheduled batch updates.

## Example

The following code terminates the audit trail on the data set **Capinfo**.



If you choose to run this example, you must copy the data set **Capinfo** from the **Sasuser** library to the **Work** library.

```
proc datasets nolist;
  audit capinfo;
  terminate;
quit;
```

A message is written to the log when the audit trail is terminated.

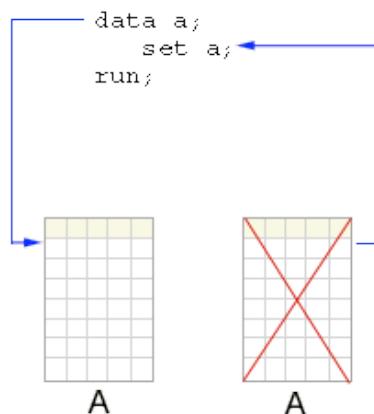
### SAS Log

```
65 audit capinfo;
66 terminate;
NOTE: Deleting WORK.CAPINFO (memtype=AUDIT).
67 run;
68 quit;
```

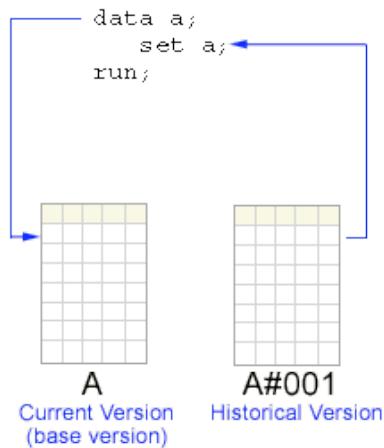
## Understanding Generation Data Sets

You have learned that you can keep an audit trail to track observation updates made to an individual data set in place. However, if you replace the data set, the audit trail is lost. Generation data sets allow you to maintain multiple versions or generations of a SAS data set. A new generation is created each time the file is replaced.

By default, **generation data sets are not in effect**. As the SAS data set **A** is replaced, there are two copies of **A** in the SAS library. When the DATA step completes execution, SAS removes the original copy of the data set **A** from the library.



When **generation data sets are in effect** and the SAS data set **A** is replaced, there are two copies of **A** in the SAS library. When the DATA step completes execution, SAS keeps the original copy of the SAS data set **A** in the library and renames it.



Each generation of a generation data set is stored as part of a generation group. Each generation data set in a generation group has the same root member name, but each has a different version number. The most recent version is called the base version. When generations are in effect, SAS filenames are limited to 28 characters. The last four characters are reserved for the version numbers.



Generation data sets are not supported on VMS.

## Initiating Generation Data Sets

To initiate generation data sets and to specify the maximum number of versions to maintain, you use the output data set option **GENMAX=** when creating or replacing a data set. If the data set already exists, you can use the **GENMAX=** option with the DATASETS procedure and the MODIFY statement.

### General form, DATASETS procedure and MODIFY statement with the GENMAX= option:

```

PROC DATASETS LIB=libref <NOLIST>;
  MODIFY SAS-data-set (GENMAX=n);
  QUIT;

```

where

- *libref* is the library that contains the data you want to modify
- **NOLIST** suppresses the directory listing
- *SAS-data-set* is the name of the SAS data set that you want to modify
- *n* is the number of historical versions you want to keep, including the base version
- *n* = 0, no historical versions are kept (this is the default).
- *n* > 0, the specified number of versions of the file will be kept. The number includes the base version.

## Example

The following DATASETS procedure modifies the data set **Cargorev** and requests that up to four versions be kept (one base version and three historical versions).



If you choose to run this example, you must copy the data set **Cargorev** from the **Sasuser** library to the **Work** library.

```

proc datasets nolist;
  modify cargorev (genmax=4);
quit;

```

No message is written to the log when you specify the GENMAX= option.

### Creating Generation Data Sets

Remember, new versions of a generation data set are created only when a data set is replaced, not when it is modified in place. To create new generations, use

- a DATA step with a SET statement
- a DATA step with a MERGE statement
- PROC SORT without the OUT= option
- PROC SQL with a CREATE TABLE statement.

Write a statement to modify the data set Snowfall and specify that one base version and seven historical versions of the data set are saved when the data set is replaced.

**Q14.**

```
proc datasets nolist;  
-----  
quit;
```

### Processing Generation Data Sets

Once you have a generation group that contains more than one generation data set, you may want to select a particular data set to process. To select a particular generation, you use the **GENNUM=** data set option.

#### General form, GENNUM= data set option:

**GENNUM=n**

- where *n* specifies a particular historical version of a data set:
- *n>0* is an absolute reference to a historical version by its generation number
- *n<0* is a relative reference to a historical version
- *n=0* is the current version.

#### Examples

To print the current version of the data, you do not need to use the GENNUM= option. Simply use

```
proc print data=year;  
run;
```

To print the youngest historical version, you can either specify the relative or absolute reference on the GENNUM= option, as shown:

```
proc print data=year(gennum=4); /*absolute reference*/  
run;
```

or

```
proc print data=year(gennum=-1); /*relative reference*/  
run;
```

You can also view information on a specific generation using the GENNUM= option with PROC CONTENTS, as shown:

```
proc contents data=year(gennum=-1); /*relative reference*/
run;
```

Now that you have seen a few examples of using the GENNUM= option, lets take a look at how generation numbers change.

### How Generation Numbers Change

When you use the GENNUM= option, you can refer to either the absolute or relative generation number. It's helpful to understand how generation numbers change so that you can identify the generation you want to process.

First, let's take a look at how SAS names generation data sets. The first time a data set with generations in effect is replaced, SAS keeps the replaced data set and appends a four-character version number to its member name, which includes the pound symbol (#) and a three-digit number. That is, for a data set named **A**, the replaced data set becomes **A#001**. When the data set is replaced for the second time, the replaced data set becomes **A#002**; that is, **A#002** is the version that is chronologically closest to the base version. The table below shows the result after three replacements.

| Data Set Name | Explanation                               |
|---------------|-------------------------------------------|
| <b>A</b>      | base (current) version                    |
| <b>A#003</b>  | most recent (youngest) historical version |
| <b>A#002</b>  | second most recent historical version     |
| <b>A#001</b>  | oldest historical version                 |

The limit for version numbers that SAS can append is #999. After 1000 replacements, SAS rolls over the youngest version number to #000.

Now let's take a look at how absolute and relative generation numbers (specified on the GENNUM= option) change. Each time SAS creates a new generation, the absolute generation number increases sequentially. As older generations are deleted, their absolute generation numbers are retired.

In contrast, the relative generation number always refers to generations in relation to the base generation. The base or current generation is always 0 and -1 is the youngest historical version.

The following table shows data set names and their absolute and relative GENNUM= numbers.

| Iteration | SAS Code                 | Data Set Names  | GENNUM= Absolute Reference | GENNUM= Relative Reference | Explanation                                                                        |
|-----------|--------------------------|-----------------|----------------------------|----------------------------|------------------------------------------------------------------------------------|
| 1         | data Year<br>(genmax=3); | <b>Year</b>     | 1                          | 0                          | The data set <b>Year</b> is created, and three generations are requested.          |
| 2         | data Year;               | <b>Year</b>     | 2                          | 0                          | <b>Year</b> is replaced. <b>Year</b> from iteration 1 is renamed <b>Year#001</b> . |
|           |                          | <b>Year#001</b> | 1                          | -1                         |                                                                                    |
| 3         | data Year;               | <b>Year</b>     | 3                          | 0                          | <b>Year</b> is replaced. <b>Year</b> from iteration 2 is renamed <b>Year#002</b> . |
|           |                          | <b>Year#002</b> | 2                          | -1                         |                                                                                    |

|   |                          |                 |   |    |                                                                                                                                                                        |
|---|--------------------------|-----------------|---|----|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|   |                          | <b>Year#001</b> | 1 | -2 |                                                                                                                                                                        |
| 4 | data Year;               | <b>Year</b>     | 4 | 0  | <b>Year</b> is replaced. <b>Year</b> from iteration 3 is renamed <b>Year#003</b> . <b>Year#001</b> from iteration 1, which is the oldest, is deleted.                  |
|   |                          | <b>Year#003</b> | 3 | -1 |                                                                                                                                                                        |
|   |                          | <b>Year#002</b> | 2 | -2 |                                                                                                                                                                        |
| 5 | data Year<br>(genmax=2); | <b>Year</b>     | 5 | 0  | <b>Year</b> is replaced, and the number of generations is changed to 2. <b>Year</b> from iteration 4 is renamed <b>Year#004</b> . The two oldest versions are deleted. |
|   |                          | <b>Year#004</b> | 4 | -1 |                                                                                                                                                                        |

Suppose a data set has GENMAX=4. After 6 updates, what relative generation number represents the **oldest** historical version of the data set?

**Q15.**

GENNUM=0

GENNUM=-3

GENNUM=6

You have learned that you use PROC DATASETS to initiate generation data sets on an existing SAS data set. Once you have created generation data sets, you can use PROC DATASETS to perform management tasks such as

- deleting all or some of the generations
- renaming an entire generation group or any member of the group to a new base name.

#### General form, PROC DATASETS with the CHANGE and DELETE statements:

```
PROC DATASETS LIB=libref <NOLIST>;
  CHANGE SAS-data-set<(GENNUM=n)>=new-data-set-name;
  DELETE SAS-data-set<(GENNUM=n | HIST | ALL)>;
  QUIT;
```

where

- *libref* is the library that contains the data you want to modify
- **NOLIST** suppresses the directory listing
- **SAS-data-set** is the name of the SAS data set you want to change or delete
- **new-data-set-name** is the new name for the SAS data set in the **CHANGE** statement
- *n* is the absolute or relative reference to a generation number
- **HIST** refers to all generations except the base version
- **ALL** refers to the base version and all generations.

#### Examples

The following code uses the CHANGE statement to rename the data set **SalesData** to **Sales**. If generations have been created, the base name of all generations will be renamed.

```
proc datasets library=quarter1 nolist;
  change salesData=sales;
quit;
```

The following code uses the GENNUM= option to rename only the second historical data set:

```
proc datasets library=quarter1 nolist;
```

```
change sales(gennum=2)=newsales;  
quit;
```

The following code deletes one historical version. This may leave a hole in the generation group.

```
proc datasets library=quarter1 nolist;  
  delete newsales(gennum=-1);  
quit;
```

When you use the GENNUM= option with the DELETE statement, you can use the HIST and ALL keywords. The following code uses the **HIST** keyword to delete all of the historical versions:

```
proc datasets library=quarter1 nolist;  
  delete newsales(gennum=HIST);  
quit;
```

The following code uses the **ALL** keyword in the GENNUM= option to delete all of the SAS data sets in a generation group:

```
proc datasets library=quarter1 nolist;  
  delete newsales(gennum=ALL);  
quit;
```



For more information about using the DATASETS procedure to process data, see the SAS documentation.

Write a statement to rename all the generations of the data set **QuarterSales** to **Sales**.

**Q.16.**

```
proc datasets lib=mylib nolist;  
  _____  
quit;
```

## Chapter Summary

### Using the MODIFY Statement

When you use the MODIFY statement to modify a SAS data set, SAS does not create a second copy of the data as it does when you use the SET, MERGE, or UPDATE statements. The descriptor portion of the SAS data set stays the same and the updated observation is written to the data set in the location of the original observation.

### Modifying All Observations in a SAS Data Set

You can use the MODIFY statement with an assignment statement to modify all the observations for a variable in a SAS data set.

### Modifying Observations Using a Transaction Data Set

To modify a master data set using a transaction data set, you use the MODIFY statement with a BY statement to specify the variable you want to use to match. When you use the MODIFY/BY statements, SAS uses a dynamic WHERE clause to locate observations in the master data set. You can specify how missing values in the transaction data set are handled by using the UPDATEMODE= option in the MODIFY statement.

### Modifying Observations Located by an Index

You can use the MODIFY statement with the KEY= option to name a simple or composite index for the SAS data set that is being modified. The KEY= argument retrieves observations from the SAS data file based on index values that are supplied by like-named variables in a transaction data set. If you have contiguous duplications in the transaction data set such that there is no match in the master data set, you can use the UNIQUE option to cause a KEY= search to always begin at the top of the index file for each duplicate transaction.

### Controlling the Update Process

The way SAS writes observations to a SAS data set when the DATA step contains a MODIFY statement depends on whether certain other statements are present. If no statement is present, SAS writes the current observation to its original place in the SAS data set. This occurs as the last action in the step as though a REPLACE statement were the last statement in the step. However, you can override this default behavior by explicitly adding the OUTPUT, REPLACE, or REMOVE statement.

You can use the automatic variable \_IORC\_ with the %SYSRC autocall macro to test for specific I/O error conditions that are created when you use the BY statement or the KEY= option in the MODIFY statement. The automatic variable \_IORC\_ contains a return code for each I/O operation that the MODIFY statement attempts to perform. The best way to test for values of \_IORC\_ is with the mnemonic codes that are provided by the **SYSRC** autocall macro.

### Placing Integrity Constraints on a Data Set

Integrity constraints are rules that you can specify in order to restrict the data values that can be stored for a variable in a SAS data file. SAS enforces integrity constraints when values associated with a variable are added, updated, or deleted. You can place integrity constraints on an existing data set using the IC CREATE statement in the DATASETS procedure.

## **Documenting and Removing Integrity Constraints**

You can view information about the integrity constraints on a data set using the CONTENTS statement in the DATASETS procedure. If you want to remove integrity constraints from a file, you use the IC DELETE statement.

## **Initiating and Terminating Audit Trails**

An audit trail is an optional SAS file that logs modifications to a SAS table. You initiate an audit trail using the DATASETS procedure with the AUDIT and INITIATE statements. You also suspend, resume, and terminate audit trails using the DATASETS procedure. Once there is data in the audit trail file, you can read it with the TYPE= data set option.

## **Controlling Data in the Audit Trail**

The audit trail file can contain three types of variables:

- data set variables that store copies of the columns in the audited SAS data file
- audit trail variables that automatically store information about data modifications
- user variables that store user-entered information.

You can use the LOG statement to control which types of records are written to an audit trail file.

## **Initiating Generation Data Sets**

Each generation of a generation data set is stored as part of a generation group. A new generation is created each time the file is replaced. Each generation in a generation group has the same root member name, but each has a different version number. You initiate generation data sets by using the GENMAX= option to specify the number of generation data sets to keep.

## **Processing Generation Data Sets**

To select a particular generation to process, you use the GENNUM= option. GENNUM= is an input/update data set option that identifies which generation to open. The GENNUM can be a relative or absolute reference to a generation within a generation group. You can rename or delete generations using the CHANGE and DELETE statements in a PROC DATASETS step.

## **Syntax**

### **Modifying All Observations in a SAS Data Set**

```
DATA SAS-data-set;
  MODIFY SAS-data-set;
    existing-variable = expression;
  RUN;
```

### **Modifying a Master Data Set Using the BY Statement**

```
MODIFY master-data-set transaction-data-set
  <UPDATEMODE=MISSINGCHECK | NOMISSINGCHECK>;
  BY key-variable;
```

## Modifying a Master Data Set Using a Transaction Data Set and an Index

```
MODIFY master-data-set KEY=index </UNIQUE>;
```

### Controlling the Update Process

```
OUTPUT;  
REPLACE;  
REMOVE;
```

### Using PROC DATASETS to create integrity constraints, generation data sets, and audit trails

```
PROC DATASETS <LIB=libref> <NOLIST> ;  
  IC CREATE <constraint-name=><constraint-type  
    <MESSAGE='Error Message'>;  
  IC DELETE constraint-name;  
  MODIFY SAS-data-set (GENMAX=n);  
  AUDIT SAS-data-file <password>;  
  INITIATE;  
  <LOG <audit-settings>>;  
  <USER_VARvariable-name <$><length><LABEL='variable-label'>>;  
  SUSPEND | RESUME | TERMINATE;  
  CONTENTS data=SAS-data-set;  
  QUIT;
```

### Using \_IORC\_ with %SYSRC

```
IF _IORC_= %SYSRC(mnemonic) THEN...
```

### Specifying an Audit Trail File

```
(TYPE=AUDIT)
```

### Using PROC DATASETS to Rename or Delete Generation Data Sets

```
PROC DATASETS LIB=SAS-library <NOLIST> ;  
  CHANGE SAS-data-set<(GENNUM=n)>=new-data-set-name;  
  DELETE SAS-data-set<(GENNUM=n | HIST | ALL)>;  
  QUIT;
```

## Sample Programs

### Modifying a Data Set Using the MODIFY Statement with a BY Statement and with the KEY= Option

```
data capacity;  
  modify capacity sasuser.newrtnum;  
  by flightid;  
run;  
data cargo99;  
  set sasuser.newcgnnum (rename =  
    (capcargo = newCapCargo  
     cargowgt = newCargoWgt  
     cargorev = newCargoRev));  
  modify cargo99 key=flghtdte;
```

```
capcargo = newcapcargo;
cargowgt = newcargowgt;
cargorev = newcargorev;
run;
```

## Placing Integrity Constraints on Data

```
proc datasets nolist;
  modify capinfo;
    ic create PKIDInfo=primary key(routeid)
      message='You must supply a Route ID Number';
    ic create Class1=check(where=(cap1st<capbusiness
      or capbusiness=.))
      message='Cap1st must be less than CapBusiness';
  quit;
```

## Initiating an Audit Trail

```
proc datasets nolist;
  audit capinfo;
  initiate;
quit;
```

## Initiating Generation Data Sets

```
proc datasets nolist;
  modify cargorev (genmax=4);
quit;
```

## Points to Remember

- The MODIFY statement in a DATA step is used to make updates to a SAS data set in place. The descriptor portion of the SAS data set cannot be changed.
- Integrity constraints are only enforced when modifications are made to the data. If the data set is replaced, integrity constraints are lost.
- Audit trail files track changes made to data sets **in place** with
  - the MODIFY statement in the DATA step
  - the UPDATE, INSERT, or DELETE statement in PROC SQL.
- Generation data sets are used to track changes that are made when a data set is replaced by
  - using the SET, MERGE, or UPDATE statements in the DATA step
  - sorting data in place with PROC SORT
  - using the CREATE TABLE statement in PROC SQL.

## Chapter Quiz

Select the best answer for each question.

1. Which type of integrity constraint would you place on the variable StoreID to ensure that there are no missing values and that there are no duplicate values?
  - a. UNIQUE
  - b. CHECK
  - c. PRIMARY KEY
  - d. NOT NULL
2. Which code creates an audit trail on the SAS data set **Reports.Quarter1**?
  - a. 

```
proc datasets nolist;
  audit quarter1;
  initiate;
  quit;
```
  - b. 

```
proc datasets lib=reports nolist;
  audit initiate reports.quarter1;
  quit;
```
  - c. 

```
proc datasets lib=reports nolist;
  initiate audit quarter1;
  quit;
```
  - d. 

```
proc datasets lib=reports nolist;
  audit quarter1;
  initiate;
  quit;
```
3. Which DATA step uses the transaction data set **Records.OVERNIGHT** to update the master data set **Records.Snowfall** by accumAmt?
  - a. 

```
data records.snowfall;
  modify records.snowfall records.overnight
  key=accumAmt;
  run;
```
  - b. 

```
data records.snowfall;
  modify records.overnight records.snowfall;
  by accumAmt;
  run;
```
  - c. 

```
data records.snowfall;
  modify records.snowfall records.overnight;
  by accumAmt;
  run;
```
  - d. 

```
data records.snowfall;
  modify records.snowfall records.overnight;
  update accumAmt;
  run;
```
4. The automatic variable **\_IORC\_** is created when you use the MODIFY statement with a BY statement or the KEY= option. How can you use the value of **\_IORC\_**?
  - a. to determine whether the index specified on the KEY= option is a valid index
  - b. to determine the number of observations that were updated in the master data set

- c. to determine the status of the I/O operation
  - d. to determine the number of observations that could **not** be updated in the master data set
- 5. Which PROC DATASETS step creates an integrity constraint named val\_age on the data set **Survey** to ensure that values of the variable age are greater than or equal to 18?
  - a. 

```
proc datasets nolist;
  modify age;
  ic create val_age=check(where=(age>=18));
quit;
```
  - b. 

```
proc datasets nolist;
  modify Survey;
  ic create val_age=check(age>=18);
quit;
```
  - c. 

```
proc datasets nolist;
  modify survey;
  integrity constraint
    val_age=check(where=(age>=18));
quit;
```
  - d. 

```
proc datasets nolist;
  modify survey;
  ic create val_age=check(where=(age>=18));
quit;
```
- 6. Which statement about using the MODIFY statement in a DATA step is **true**?
  - a. MODIFY creates a second copy of the data while variables in the data are being matched with a WHERE clause and then deletes the second copy.
  - b. You cannot modify the descriptor portion of the data set using the MODIFY statement.
  - c. You can use the MODIFY statement to change the name of a variable.
  - d. If the system terminates abnormally while a DATA step that is using the WHERE statement is processing, SAS automatically saves a copy of the unaltered data set.
- 7. Which of the following statements about audit trails is **true**?
  - a. They create historical versions of data so that a copy of the data set is saved each time the data is replaced.
  - b. They record information about changes to observations in a data set each time the data set is replaced.
  - c. They record information about changes to observations in a data set each time the data is modified in place.
  - d. The audit trail file has the same name as the SAS data file it is monitoring, but has #AUDIT at the end of the data set name.
- 8. Which code initiates generation data sets on the existing SAS data set **Sasuser.Amounts** and specifies that five historical versions are saved in addition to the base version?
  - a. 

```
proc datasets lib=sasuser nolist;
  modify Amounts (genmax=6);
quit;
```
  - b. 

```
proc datasets lib=sasuser nolist;
  modify Amounts (genmax=5);
quit;
```

- c. proc datasets lib=sasuser nolist;  
    modify Amounts (gennum=6);  
    quit;
- d. proc datasets lib=sasuser nolist;  
    modify Amounts (gennum=5);  
    quit;
9. Which statement about using the KEY= option in the MODIFY statement is **true**?
- SAS locates the variables to update using the index specified in the KEY= option and then automatically overlays nonmissing transaction values as it does when you use the MODIFY/BY statements.
  - When you use the KEY= option, you must explicitly state the update that you want to make. SAS does not automatically overlay nonmissing transaction values.
  - The KEY= option is used to specify a variable to match for updating observations.
  - The index named in the KEY= option must be a simple index.
10. Which code deletes all generations of the data set **Sasuser.Amounts** including the base data set?
- proc datasets lib=sasuser nolist;  
    delete amounts (gennum=ALL);  
    quit;
  - proc datasets lib=sasuser nolist;  
    delete amounts (gennum=HIST);  
    quit;
  - proc datasets lib=sasuser nolist;  
    delete amounts (gennum=0);  
    quit;
  - proc datasets lib=sasuser nolist;  
    delete amounts;  
    quit;

# Chapter 7

## Introduction to Efficient SAS Programming

### Introduction

As an experienced programmer, you want your SAS programs to obtain the desired results while minimizing the use of resources such as CPU time, real time, memory, and I/O.



It is particularly important to optimize your SAS programs if you write or maintain production programs and work with large data sets. However, before you can select the most efficient programming technique to perform a particular task, you must carefully consider the technical environment and the resource constraints at your site. There is no single set of programming techniques that is most efficient in all situations. Instead, trade-offs in resource usage are associated with each technique.

In this lesson you will learn about analyzing the requirements for efficiency at your site and about running benchmarks to select the most efficient SAS programming techniques.



This lesson has no quiz.

### Objectives

In this lesson, you learn to

- identify the resources that are used by a SAS program
- identify the main factors to consider when you assess the efficiency needs at your site
- identify common efficiency trade-offs
- select the appropriate SAS system option(s) to track and report the resource usage statistics that you want in your operating environment
- interpret resource usage statistics that are displayed in the SAS log in the z/OS, UNIX, and Windows operating environments
- identify the guidelines that you should follow in order to benchmark effectively.

## Overview of Computing Resources

Running a SAS program requires the programmer and the computer to perform a variety of tasks. The **programmer** must write, submit, and maintain the program. The **computer** must load the required SAS software components and the program into memory, compile the program, locate the data that the program requires, and execute the program.

The following resources are used to run a SAS program:

| Resource                                                                                                  | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-----------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| programmer time<br>      | the amount of time required for the programmer to write and maintain the program. Programmer time is difficult to quantify, but it can be decreased through well-documented, logical programming practices and the use of reusable SAS code modules.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| CPU time<br>             | the amount of time the central processing unit (CPU) uses to perform requested tasks such as calculations, reading and writing data, conditional logic, and iterative logic.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| real time<br>            | the clock time (elapsed time) it takes to execute a job or step. Real time is heavily dependent on the capacity of the system and on the load (the number of users that are sharing the system's resources).<br> Because you cannot always control the capacity and the load demands on your system, real time is sometimes a less useful measure of program efficiency than CPU time. However, excessive use of real time often motivates programmers to improve a program's efficiency. Some procedures in SAS 9.1 give you the option of using <a href="#">threaded processing</a> to reduce real time. Threaded processing can cause an increase in CPU time, so it is recommended that you track both CPU time and real time. |
| memory<br>             | the size of the work area in volatile memory that is required for holding executable program modules, data, and buffers.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| data storage space<br> | the amount of space on a disk or tape that is required for storing data. Data storage space is measured in a variety of units, some of which are used only in certain operating environments, as described below: <ul style="list-style-type: none"><li>All operating environments use bytes, kilobytes, megabytes, gigabytes, and terabytes.</li><li>z/OS also uses blocks, tracks, and cylinders.</li></ul>                                                                                                                                                                                                                                                                                                                                                                                                       |
| I/O<br>                | a measurement of the read and write operations that are performed as data and programs are copied from a storage device to memory (input) or from memory to a storage or display device (output).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |

## Threaded Processing

If your site has installed SAS 9.1 or later, then you should consider whether you can optimize the performance of your SAS programs by taking advantage of the new SAS threaded technology. This technology enables you to work with virtually any volume of data efficiently and to exploit your hardware's capabilities to the maximum. **Threading** improves performance by enabling a single SAS session to use **multiple I/O channels** and to take advantage of hardware that has **multiple CPUs**, called symmetric multiprocessor (SMP) machines. SMP machines can spawn and manage multiple pieces of executable code, called threads.

Each **thread** is a single, independent flow of control through a program or within a process. **Threaded processing** takes advantage of multiple CPUs by executing multiple threads in parallel (parallel processing). Threaded jobs are completed in substantially less real time than if each task were handled sequentially. (CPU time for threaded jobs is generally increased because of the overhead that is required for managing the threads, separating the data into streams for processing in separate threads, and combining the data from the threads into a single file.)

Threaded processing is an important part of the solution to two common constraints on application performance: constraints on I/O and constraints on the CPU. SAS 9.1 threaded technology addresses these constraints on SAS performance as described below:

| Constraint                                                                                                                                                     | Solution                                                                                                                                                                     | Type of SAS Processing Supported       |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------|
| <b>I/O-bound</b> - the application can process data faster than the data can be delivered to the application for processing or for output from the application | multiple CPUs<br>partitioned I/O<br>threaded technology: <b>SAS Scalable Performance Data Engine (SPD Engine)</b><br>RAID (redundant arrays of inexpensive disks) technology | <b>threaded I/O</b>                    |
| <b>CPU-bound</b> - the application can receive data faster than it can perform the necessary processing on that data.                                          | multiple CPUs<br>threaded technology: <b>thread-enabled SAS procedures</b>                                                                                                   | <b>threaded application processing</b> |

Let's take a closer look at how SAS threaded technology improves performance.

### SAS Scalable Performance Data Engine

The SAS Scalable Performance Data Engine (SPD Engine) supports threaded I/O, which can boost the performance of SAS applications that are I/O bound. The SPD Engine supports threaded I/O by physically partitioning each data set into smaller pieces that can each be distributed across a number of different disks or disk arrays. Partitioned data sets can span disk drives but can still be referenced as a single data set. When an SPD Engine data set is read, the data from these partitions can be delivered simultaneously (in parallel). In this way, the SPD Engine can read many millions of observations into SAS applications very rapidly by spawning a thread for each data partition and evaluating WHERE expressions on multiple CPUs. For large data sets, the performance gain from threaded I/O is significant.



The SPD Engine has additional capabilities that are not covered here. For more information about the SPD Engine, see the documentation for the SAS Scalable Performance Data Engine.

### Thread-enabled SAS Procedures

For CPU-bound applications, the solution is to divide the job of computation across multiple simultaneous threads. These threads can run on one CPU or on multiple CPUs. Even if your application is not currently CPU-bound, if you increase the amount of data that can be delivered to an application, you will naturally increase the need for faster processing of that data. Thread-enabled SAS procedures can solve this problem.

In Base SAS 9.1, the following procedures have been thread-enabled (modified so that multiple threads can be performed in parallel if the hardware supports threaded processing): PROC SORT, PROC SQL, PROC MEANS, PROC SUMMARY, PROC REPORT, and PROC TABULATE. In addition, threaded processing is being integrated into a variety of other SAS features in order to improve performance. For example, when you create an index, if sorting is required, SAS attempts to use the thread-enabled sort to sort the data.



To learn about thread-enabled procedures in products other than Base SAS 9.1, see the SAS documentation.

## Assessing Efficiency Needs at Your Site

The first step in making an effective decision about how to optimize your SAS programs is to assess your site's technical environment, your program(s), and your data.

### Assessing Your Technical Environment

To determine which resources are scarce or costly at your site, work with your IT department to analyze the following characteristics of your technical environment:

| Category              | Characteristics                                                                                                                                                                                   |
|-----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| hardware              | amount of available memory<br>number of CPUs<br>number and type of peripheral devices<br>communications hardware<br>network bandwidth<br>storage capacity<br>I/O bandwidth<br>capacity to upgrade |
| operating environment | resource allocation<br>scheduling algorithms<br>I/O methods                                                                                                                                       |
| system load           | number of users or jobs sharing system resources<br>network traffic (expected)<br>predicted increase in load in the future                                                                        |
| SAS environment       | which SAS software products are installed<br>number of CPUs and amount of memory allocated for SAS programming<br>which methods are available for running SAS programs at your site               |

In most cases, one or two resources are the most limited or most expensive for your programs. You can usually decrease the amount of critical resources that are used if you are willing to sacrifice some efficiency of the resources that are less critical at your site.

### Assessing Your Programs

Developing an efficient program requires time and thought. To determine whether the additional amount of resources saved is worth the time and effort spent to achieve the savings, consider the following characteristics of each of your programs:

| Characteristic | Guidelines for Optimizing |
|----------------|---------------------------|
|----------------|---------------------------|

|                                      |                                                                                                                                                                                                                                                                                                          |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| size of the program                  | As the program increases in size, the potential for savings increases. Focus on improving the efficiency of <b>large programs</b> .                                                                                                                                                                      |
| number of times the program will run | The difference in resources used by an inefficient program and an efficient program that is run once or a few times is relatively small, whereas the cumulative difference for a program that is run frequently is large. Focus on improving the efficiency of programs that are <b>run many times</b> . |

## Assessing Your Data

The effectiveness of any efficiency technique depends greatly on the data with which you use it. When you know the characteristics of your data, you can select the techniques that take advantage of those characteristics. Consider the following characteristics of your data:

| Characteristic | Guidelines for Optimizing                                                                                                                                                    |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| volume of data | As the volume of data increases, the potential for savings also increases. Focus on improving the efficiency of programs that use <b>large data sets or many data sets</b> . |
| type of data   | Specific efficiency techniques might work better with some <b>types of data</b> (for example, data that has missing values) than with others.                                |

## Understanding Efficiency Trade-offs

When you are trying to optimize SAS programs, it is important to understand that there are trade-offs. Decreasing the use of one resource frequently increases the use of another. The following table shows examples of some common efficiency trade-offs.

| Decreasing usage of this resource ...                                                                                                                  | Might increase usage of this resource ...                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|
| <b>disk space</b><br>                                               | <b>CPU time</b><br>        |
| <b>I/O</b> (by reading or writing more data at one time)<br>        | <b>memory</b><br>          |
| <b>real time</b> (by enabling <u>threading</u> in SAS 9.1)<br>      | <b>CPU time</b><br>        |
| <b>any computer resource</b> (by increasing program complexity)<br> | <b>programmer time</b><br> |

As these trade-offs illustrate, **there is no single best way to optimize a SAS program; it depends on your situation**. However, this lesson and the following lessons provide information that can help you determine which programming techniques are most efficient in your environment:

- **Controlling Memory Usage**

- Controlling Data Storage Space
- Utilizing Best Practices
- Selecting Efficient Sorting Strategies
- Querying Data Efficiently.

Decreasing disk space usage might **increase** usage of which of the following resources?

- CPU time  
**Q1.** I/O  
 both  
 neither

## Using SAS System Options to Track Resources

You can specify one or more of the SAS system options **STIMER**, **MEMRPT**, **FULLSTIMER**, and **STATS** to track and report on resource utilization. The availability, usage, and functionality of these options vary by operating environment, as described below:

| Option            | <b>z/OS</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          | <b>UNIX and Windows</b>                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>STIMER</b>     | Specifies that the <b>CPU time</b> is to be <b>tracked</b> throughout the SAS session. Can be set at <b>invocation only</b> . Is the <b>default</b> setting.                                                                                                                                                                                                                                                                                                                                                                                                         | Specifies that <b>CPU time and real-time statistics</b> are to be <b>tracked and written</b> to the SAS log throughout the SAS session. Can be set <b>either</b> at invocation or by using an <b>OPTIONS</b> statement. Is the <b>default</b> setting.                                                                                                                                                                                                      |
| <b>MEMRPT</b>     | Specifies that <b>memory usage statistics</b> are to be <b>tracked</b> throughout the SAS session. Can be set <b>either</b> at invocation or by using an <b>OPTIONS</b> statement. Is the <b>default</b> setting.                                                                                                                                                                                                                                                                                                                                                    | <b>Not available</b> as a separate option; this functionality is part of the <b>FULLSTIMER</b> option.                                                                                                                                                                                                                                                                                                                                                      |
| <b>FULLSTIMER</b> | Specifies that <b>all available resource usage statistics</b> are to be <b>tracked and written</b> to the SAS log throughout the SAS session. Can be set <b>either</b> at invocation or by using an <b>OPTIONS</b> statement.<br><br> In the z/OS operating environment, FULLSTIMER is an alias for the FULLSTATS option.<br><br> This option is ignored unless STIMER or MEMRPT is in effect. | Specifies that <b>all available resource usage statistics</b> are to be <b>tracked and written</b> to the SAS log throughout the SAS session. Can be set <b>either</b> at invocation or by using an <b>OPTIONS</b> statement.<br><br> In Windows operating environments, some statistics will not be calculated accurately unless FULLSTIMER is specified at invocation. |
| <b>STATS</b>      | Tells SAS to <b>write</b> statistics that are tracked by any combination of the preceding options to the SAS log. Can be set <b>either</b> at invocation or by using an <b>OPTIONS</b> statement. Is the <b>default</b> setting.                                                                                                                                                                                                                                                                                                                                     | <b>Not available</b> as a separate option.                                                                                                                                                                                                                                                                                                                                                                                                                  |

SAS system options are initialized with default settings when SAS is invoked. However, the default settings for some SAS system options vary both by operating environment and by site. For details, see the SAS documentation for your operating environment.

You can turn off any of these system options by using the options below:

- **NOSTIMER**
- **NOMEMRPT**
- **NOFULLTIMER**
- **NOSTATS**

 In the z/OS operating environment, NOFULLTIMER is an alias for the NOFULLSTATS option.

### **Viewing Resource Statistics**

To see examples of the resource usage statistics that are written to the SAS log in each operating environment, select the links below.

- z/OS
- UNIX
- Windows

 For more information about using these system options to track resource usage in your operating environment, see the SAS documentation for your operating environment. Guidelines for interpreting the statistics that are generated by the FULLTIMER SAS option are also available at support.sas.com.

 You can also use SAS Application Response Measurement (ARM) macros to monitor the performance of your applications. ARM macros are not covered in this course. To learn more about ARM macros, see the SAS documentation and detailed information about ARM macros at support.sas.com.

Which of the following system options enables you to track all available resource usage statistics throughout a SAS session?

- Q.2.**
- STATS
  - STIMER
  - NOMEMRPT
  - FULLTIMER

### **Using Benchmarks to Compare Techniques**

To decide which SAS programming technique is most efficient for a particular task, you can **benchmark** (measure and compare) the resource usage for each technique that you are comparing.

#### **Guidelines for Benchmarking**

Your benchmarking is most likely to yield useful results if you follow these guidelines:

- **Before you test the programming techniques, turn on the SAS system options that report resource usage.**

As explained earlier, to track and report on resource utilization, you can use some or all of the system options STIMER, MEMRPT, FULLSTIMER, and STATS. The availability, usage, and functionality of these options vary by operating environment. You can also specify MSGLEVEL=1 to display additional notes in the SAS log.



For more information about the SAS system option MSGLEVEL=, see the lesson [Creating Samples and Indexes](#) or the lesson [Creating and Managing Indexes Using PROC SQL](#).

- **Execute the code for each programming technique in a separate SAS session.**

If the program that you are benchmarking is not large, you can optimize it by changing individual programming techniques, one at a time, and running the entire program after each change to measure the effect on resource usage. However, a more complex program might be easier to optimize by identifying the steps that use the most resources and extracting those steps into separate programs. You can measure the effects of different programming techniques by repeatedly changing, running, and measuring the separate programs. When isolating parts of your program, be careful to measure their resource usage under the conditions in which they are used in the complete program.



The first time that program code (including the DATA step, functions and formats, and SAS procedures) is referenced, the operating system might have to load the code into memory or assign virtual address space to it. The first time data is read, it is often loaded into a cache from which it can be retrieved more quickly the next time it is read. The resource usage that is required for performing these actions is called overhead.

- **In each programming technique that you are testing, include only the SAS code that is essential for performing the task.**

If you include too many elements in the code for each technique, you will not know what caused the results.

- **If your system is doing other work at the same time you are running your benchmarking tests, be sure to run the code for each programming technique several times.**

Running the code several times reduces any variability in resource consumption that is associated with other work that the system is doing. How you handle multiple measurements depends on the resource, as indicated below:

- Use the minimum **real time** and **CPU time** measurements, because these represent most closely the amount of time your programming technique actually requires. The larger time values (especially in the case of real time) are the result of interference from other work that the computer was doing while your program ran.
- The amount of **memory** shouldn't vary from trial to trial. If memory does vary, it's possible that your program sometimes shares a resource with another program. In this situation, you must determine whether the higher or lower memory consumption is more likely to be the case when your program is used in production.
- **I/O** can be an especially elusive resource to measure. With modern file systems and storage systems, the effect of your program on the I/O activity of the computer sometimes has to be observed by operating system tools, file system tools, or storage

system tools because it cannot be captured by your SAS session. Data is often aggressively cached by modern file systems and storage systems, and file caches are greatly affected by other activity in the file system. Be as realistic as you can about measuring I/O—it is possible to achieve good performance on a system that is not doing other work, but performance is likely to worsen when the application is deployed in a more realistic environment.

- **Run your benchmarking tests under the conditions in which your final program will run.**

Results might vary under different conditions, so it is important to control the conditions under which your benchmarks are tested. For example, if batch execution and large data sets are used in your environment, you should incorporate these conditions into your benchmarking environment.

- **After testing is finished, consider turning off the options that report resource usage.**

The options that report resource usage consume resources. If it is a higher priority in your environment to minimize resource usage than to periodically check an application's resource usage, then it is most efficient to turn off these options.



To turn off the FULLSTIMER option, use the following statement:

```
options nofulltimer;
```

Which of the following should you **not** do when you are benchmarking?

**Q.3.**

When you run your tests, make sure that the system is doing no other work.

To avoid including any initial overhead in your benchmark measurements, execute the code for each technique in a separate SAS session.

If you have multiple measurements for real time and CPU time, use the minimum values.

Include only the SAS code that is essential for performing the task that you are measuring.

## Chapter Summary

### **Overview of Computing Resources**

Resources that are required for running a SAS program include the following: programmer time, CPU time, real time, memory, data storage space, and I/O.

### **Assessing Efficiency Needs at Your Site**

To make an effective decision about how to optimize your SAS programs, work with your IT department to assess the following factors at your site: the technical environment, your individual SAS programs, and the data.

### **Understanding Efficiency Trade-offs**

It is important to understand the trade-offs that are involved in optimizing your SAS programs. Decreasing the use of one resource frequently increases the use of another. There is no single best way to optimize a SAS program; it depends on your situation.

### **Using SAS System Options to Track Resources**

You can specify one or more of the SAS system options STIMER, MEMRPT, FULLSTIMER, and STATS to track and report on resource utilization. (In the z/OS environment, FULLSTIMER is an alias for FULLSTATS.) The availability, usage, and functionality of these options varies by operating environment.

### **Using Benchmarks to Compare Techniques**

To determine which SAS programming technique is most efficient for a particular task, you can benchmark (measure and compare) the resource usage of each technique.

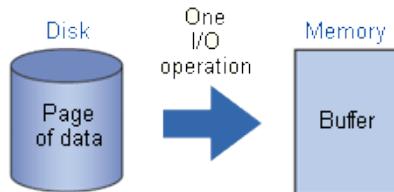
# Chapter 8

## Controlling Memory Usage

### Introduction

As you have learned, there is no single set of programming techniques that is most efficient or appropriate in all situations. However, if reducing execution time is an important consideration in your computing environment, one way of achieving that goal is to reduce the number of times SAS has to read from or write to the storage medium.

In this lesson you learn to use options and a statement to control the size and number of data buffers, which in turn can affect your programs' execution times by reducing the number of I/O operations that SAS must perform.



This lesson does not cover the Scalable Performance Data Engine (SPD Engine), which is a SAS 9.1 technology for threaded processing. For details about using the SPD Engine to improve performance, see the SAS documentation.

### Objectives

In this lesson, you learn to

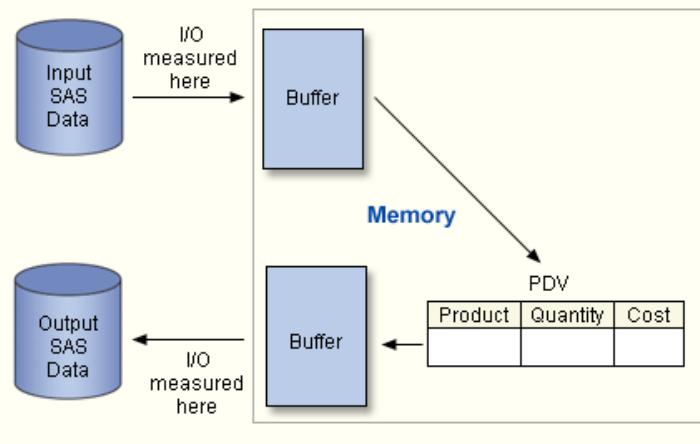
- control the amount of data that is loaded into memory with each I/O transfer
- reduce I/O by holding a SAS data file in memory through multiple steps of a program.

## Controlling Page Size and the Number of Buffers

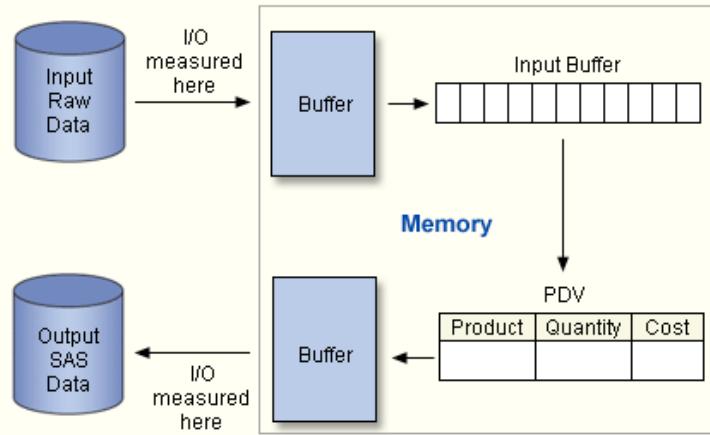
### Measuring I/O

Improvement in I/O can come at the cost of increased memory consumption. In order to understand the relationship between I/O and memory, it is helpful to know when data is copied to a buffer and where I/O is measured. When you create a SAS data set using a DATA step,

1. SAS copies the data from the input data set to a buffer in memory
2. one observation at a time is loaded into the program data vector
3. each observation is written to an output buffer when processing is complete
4. the contents of the output buffer are written to the disk when the buffer is full.



The process for reading external files is similar. However, each record is first read into the input buffer before the data is parsed and read into the program data vector.



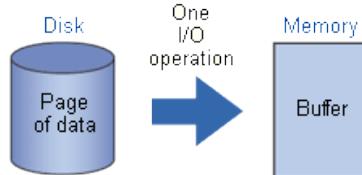
In both cases, I/O is measured when the input data is copied to the buffer in memory and when it is read from the output buffer to the output data set.

### Page Size

Think of a buffer as a container in memory that is big enough for only one page of data. A **page**

- is the unit of data transfer between the storage device and memory
- includes the number of bytes used by the descriptor portion, the data values, and any overhead
- is fixed in size when the data set is created, either to a default value or to a user-specified value.

The amount of data that can be transferred to one buffer in a single I/O operation is referred to as **page size**. Page size is analogous to buffer size for SAS data sets.



A larger page size can reduce execution time by reducing the number of times SAS has to read from or write to the storage medium. However, the improvement in execution time comes at the cost of increased memory consumption.

### Reporting Page Size

You can use the CONTENTS procedure or the CONTENTS statement in the DATASETS procedure to report the page size and the number of pages.

```
proc contents data=company.order_fact;  
run;
```

Partial PROC CONTENTS Output

| Engine/Host Dependent Information |                        |
|-----------------------------------|------------------------|
| <b>Data Set Page Size</b>         | 8192                   |
| <b>Number of Data Set Pages</b>   | 9423                   |
| <b>First Data Page</b>            | 1                      |
| <b>Max Obs per Page</b>           | 101                    |
| <b>Obs in First Data Page</b>     | 70                     |
| <b>Number of Data Set Repairs</b> | 0                      |
| <b>File Name</b>                  | c:\order_fact.sas7bdat |
| <b>Release Created</b>            | 9.0101B0               |
| <b>Host Created</b>               | XP_PRO                 |

The total number of bytes that a data file occupies equals the page size multiplied by the number of pages. For example, the page size for **Company.Order\_fact** is 8192 and the number of pages is 9423. Therefore, the data file occupies 77,193,216 bytes.



Note that the information that is available from PROC CONTENTS depends on the operating environment.



Page size is analogous to buffer size for SAS data sets.



In uncompressed data files, there is a 16-byte overhead at the beginning of each page and a 1-bit per observation overhead (rounded up to the nearest byte), used to denote an observation's status as deleted or not deleted, at the end of each page.

You can learn about the structure of uncompressed and compressed data files in the lesson **Controlling Data Storage Space**.

Which of the following statements is true?

**Q.1.**

- A page is the unit of data transfer between the storage device and memory.
- The size of a page can be increased or decreased after a data set is created.
- A page includes only the number of bytes that are used by the data values.

Where is I/O measured when a SAS data set is created?

**Q.2.**

- A. when the input data is copied from the source to the input buffer
  - B. when the input data is copied from the input buffer to the program data vector
  - C. when the output data is copied from the output buffer to the output data set.
- A and B
  - A and C
  - A, B, and C.

## Using the BUFSIZE= Option

To select a default page size, SAS uses an algorithm that is based on observation length, engine, and operating environment. The default page size is optimal for most SAS activities, especially on computers that are supporting multiple SAS jobs concurrently. However, in some cases, choosing a page/buffer size that is larger than the default can speed up execution time by reducing the number of times that SAS must read from or write to the storage medium.

You can use the **BUFSIZE=** system option or data set option to control the page size of an output SAS data set. **BUFSIZE=** specifies not only the page size (in bytes), but also the size of each buffer that is used for reading or writing the SAS data set. The new buffer size is a permanent attribute of the data set; after it is specified, it is used whenever the data set is processed.

### General form, **BUFSIZE=** option:

**BUFSIZE= MIN | MAX | *n*;**

where

MIN sets the page size to the smallest possible number in your operating environment.

MAX sets the page size to the maximum possible number in your operating environment.

*n* specifies the page size in bytes. For example, a value of 8 specifies a page size of 8 bytes, and a value of 4K specifies a page size of 4096 bytes. The default is 0, which causes SAS to use the optimal page size for the operating environment.



MIN might cause unexpected results and should be avoided. Use BUFSIZE=0 to reset the buffer page size to the default value in your operating environment.



The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

Only certain page/buffer size values are valid for each operating environment. If you request an invalid value for your operating environment, SAS automatically rounds up to the next valid page/buffer size. BUFSIZE=0 is interpreted as a request for the default page/buffer size.

In the following program, the BUFSIZE= system option specifies a page size of 30720 bytes.

```
options bufsize=30720;
filename orders 'c:\orders.dat';
data company.orders_fact;
  infile orders;
  <more SAS code>
run;
```

Before you change the default page size, it is important to consider the access pattern for the data as well as the I/O transfer rate of the underlying hardware. In some cases, increasing the page size might degrade performance, particularly when the data is processed using direct (random) access.



The default value for BUFSIZE= is determined by your operating environment and is set to optimize sequential access. To improve performance for direct access, you should change the value for BUFSIZE=. For the default setting and possible settings for direct access, see the BUFSIZE= system option in the SAS documentation for your operating environment.



You can override the BUFSIZE= system option by using the BUFSIZE= data set option.



If you use the COPY procedure to copy a data set to a library that is accessed via a different engine, the original page/buffer size is **not** necessarily retained.

## Using the BUFNO= Option

You can use the BUFNO= system or data set option to control the number of buffers that are available for reading or writing a SAS data set. By increasing the number of buffers, you can control how many pages of data are loaded into memory with each I/O transfer.



Increasing the number of buffers might not affect performance under the Windows and UNIX operating environments, especially when you work with large data sets. By default, the Windows and UNIX operating environments read one buffer at a time. Under the SAS 9 Windows environment, you can override this default by turning on the SGIO system option when you invoke SAS. For details on the SGIO system option, see the SAS documentation for the Windows operating environment.

The following techniques might help to minimize I/O consumption:

- When you work with a small data set, allocate as many buffers as there are pages in the data set so that the entire data set can be loaded into memory. This technique is most effective if you read the same observations several times during processing.
- Under the z/OS operating environment, increase the number of buffers allocated, rather than the size of each buffer, as the size of the data set increases.

**General form, BUFNO= option:**

**BUFNO= MIN | MAX | n;**

where

- MIN sets the minimum number of buffers to 0, which causes SAS to use the minimum optimal value for the operating environment. This is the default.
- MAX sets the number of buffers to the maximum possible number in your operating environment, up to the largest four-byte, signed integer, which is  $2^{31}-1$ , or approximately 2 billion.
- *n* specifies the number of buffers to be allocated.



The recommended maximum for this option is 10.

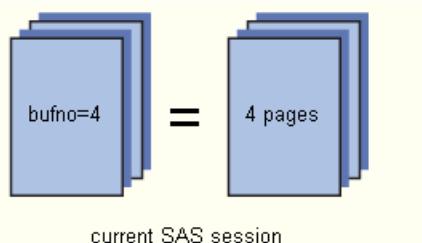


The syntax that is shown here applies to the OPTIONS statement. On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment.

In the following program, the BUFNO= system option specifies that 4 buffers are available.

```
options bufno=4;
filename orders 'c:\orders.dat';
data company.orders_fact;
  infile orders;
  <more SAS code>
run;
proc print data=company.orders_fact;
run;
```

The buffer number is not a permanent attribute of the data set and is valid only for the current step or SAS session.



You can override the BUFNO= system option by using the BUFNO= data set option.



In SAS 9 and later, the BUFNO= option has no effect on thread-enabled procedures under the z/OS operating environment.

The product of BUFNO= and BUFSIZE=, rather than the specific value of either option, determines how much data can be transferred in one I/O operation. Increasing the value of either option increases the amount of data that can be transferred in one I/O operation.

| <b>BUFSIZE</b> | <b>BUFNO</b> | <b>Bytes Transferred<br/>In One I/O Operation</b> |
|----------------|--------------|---------------------------------------------------|
| 6144           | 2            | 12,288                                            |
| 6144           | 10           | 61,440                                            |
| 30,720         | 2            | 61,440                                            |
| 30,720         | 10           | 307,200                                           |

The number of buffers and the buffer-size have a minimal effect on CPU usage.



### Comparative Example: Using the BUFSIZE= Option and the BUFNO= Option

Suppose you want to compare the resource usage when a data set is read using different buffer sizes and a varying number of buffers. The following sample programs use the following settings for the BUFNO= option and the BUFSIZE= option.

1. BUFSIZE=6144, BUFNO=2
2. BUFSIZE=6144, BUFNO=5
3. BUFSIZE=6144, BUFNO=10
4. BUFSIZE=12288, BUFNO=2
5. BUFSIZE=12288, BUFNO=5
6. BUFSIZE=12288, BUFNO=10

You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for controlling page size and the number of buffers.



6144 bytes is the default page size under the z/OS operating environment.

## Programming Techniques

### 1. BUFSIZE=6144, BUFNO=2

This program reads the data set **Retail.Order\_fact** and creates the data set **Work.Orders**. The BUFSIZE= option specifies that **Work.Orders** is created with a buffer size of 6144 bytes. The BUFNO= option specifies that 2 pages of data are loaded into memory with each I/O transfer.

```
data work.orders (bufsize=6144 bufno=2);
  set retail.order_fact;
run;
```

### 2. BUFSIZE=6144, BUFNO=5

This program reads the data set **Retail.Order\_fact** and creates the data set **Work.Orders**. The BUFSIZE= option specifies that **Work.Orders** is created with a buffer size of 6144 bytes. The BUFNO= option specifies that 5 pages of data are loaded into memory with each I/O transfer.

```
data work.orders (bufsize=6144 bufno=5);
  set retail.order_fact;
run;
```

### 3. BUFSIZE=6144, BUFNO=10

This program reads the data set **Retail.Order\_fact** and creates the data set **Work.Orders**. The BUFSIZE= option specifies that **Work.Orders** is created with a buffer size of 6144 bytes. The BUFNO= option specifies that 10 pages of data are loaded into memory with each I/O transfer.

```
data work.orders (bufsize=6144 bufno=10);
  set retail.order_fact;
run;
```

### 4. BUFSIZE=12288, BUFNO=2

This program reads the data set **Retail.Order\_fact** and creates the data set **Work.Orders**. The BUFSIZE= option specifies that **Work.Orders** is created with a buffer size of 12288 bytes. The BUFNO= option specifies that 2 pages of data are loaded into memory with each I/O transfer.

```
data work.orders (bufsize=12288 bufno=2);
  set retail.order_fact;
run;
```

### 5. BUFSIZE=12288, BUFNO=5

This program reads the data set **Retail.Order\_fact** and creates the data set **Work.Orders**. The BUFSIZE= option specifies that **Work.Orders** is created with a buffer size of 12288 bytes. The BUFNO= option specifies that 5 pages of data are loaded into memory with each I/O transfer.

```
data work.orders (bufsize=12288 bufno=5);
  set retail.order_fact;
run;
```

### 6. BUFSIZE=12288, BUFNO=10

This program reads the data set **Retail.Order\_fact** and creates the data set **Work.Orders**. The BUFSIZE= option specifies that **Work.Orders** is created with a buffer size of 12288 bytes. The BUFNO= option specifies that 10 pages of data are loaded into memory with each I/O transfer.

```
data work.orders (bufsize=12288 bufno=10);
  set retail.order_fact;
run;
```

## General Recommendations

- To reduce I/O operations on a small data set, allocate as many buffers as there are pages in the data set so that the entire data set can be loaded into memory. This technique is most effective if you read the same observations several times during processing.
- Under the z/OS operating environment, as the size of the data set increases, increase the number of buffers allocated, rather than the size of each buffer, to minimize I/O consumption.

Which system option can be used to control the size of a page?

**Q.3.**

BUFNO=

BUFSIZE=

PAGESIZE=

Which of the following is generally the most important factor in determining how much data can be transferred in one I/O operation?

**Q.4.** The value of the BUFNO= option.

The value of the BUFSIZE= option.

The product of the BUFNO= option and the BUFSIZE= option.

## Using the SASFILE Statement

Another way of improving performance is to use the SASFILE statement to hold a SAS data file in memory so that the data is available to multiple program steps. Keeping the data file open reduces open/close operations, including the allocation and freeing of memory for buffers.

### General form, SASFILE statement

**SASFILE SAS-data-file <(password-option(s))> OPEN | LOAD | CLOSE;**

where

- **SAS-data-file** is a valid SAS data file (a SAS data set with the member type DATA)
- **password-option(s)** specifies one or more password options
- **OPEN** opens the file and allocates the buffers, but defers reading the data into memory until a procedure or statement is executed
- **LOAD** opens the file, allocates the buffers, and reads the data into memory
- **CLOSE** closes the file and frees the buffers.

The SASFILE statement opens a SAS data file and allocates enough buffers to hold the entire file in memory. Once the data file is read, the data is held in memory, and it is available to subsequent DATA and PROC steps or applications until either

- a SASFILE CLOSE statement frees the buffers and closes the file
- the program ends, which automatically frees the buffers and closes the file.

In the following program, the SASFILE statement opens the SAS data file **Company.Sales**, allocates the buffers, and reads the data into memory.

```
sasfile company.sales load;
proc print data=company.sales;
  var Customer_Age_Group;
run;
proc tabulate data=company.sales;
  class Customer_Age_Group;
  var Customer_BirthDate;
  table Customer_Age_Group,Customer_BirthDate*(mean median);
run;
sasfile company.sales close;
```



The SASFILE statement can also be used to reduce CPU time and I/O in SAS programs that repeatedly read one or more SAS data views. Use a DATA step to create a SAS data file in the **Work** library that contains the view's result set. Then use the SASFILE statement to load that data file into memory.



Though a file that is opened with the SASFILE statement can be used for subsequent input or update processing, it cannot be used for subsequent utility or output processing. For example, you cannot replace the file or rename its variables.

### Guidelines for Using the SASFILE Statement

When the SASFILE statement executes, SAS allocates the number of buffers based on the number of pages for the data file and index file. If the file in memory increases in size during processing because of changes or additions to the data, the number of buffers also increases.

It is important to note that I/O processing is reduced only if there is sufficient real memory. If there is not sufficient real memory, the operating environment might

- use virtual memory
- use the default number of buffers.

If SAS uses virtual memory, there might be a degradation in performance.

If you need to repeatedly process part of a SAS data file and the entire file won't fit into memory, use a DATA step with the SASFILE statement to create a subset of the file that does fit into memory, and then process that subset repeatedly. This saves CPU time in the processing steps because those steps will read a smaller file, in addition to the benefit of the file being resident in memory.



When using a SASFILE statement, monitor the paging activity (the I/O activity that is done by the virtual memory management subsystem of your operating environment) while your program runs. If the paging activity increases substantially, consider keeping less data in memory and using techniques described elsewhere in this course to reduce memory requirements.



### Comparative Example: Using the SASFILE Statement

Suppose you want to create multiple reports from SAS data files that vary in size. Using small, medium, and large data files, you can compare the resource usage when the PRINT, TABULATE, MEANS, and FREQ procedures are used with and without the SASFILE statement to create reports.

| Name of Data File | Number of Rows | Page Size | Number of Pages | Number of Bytes |
|-------------------|----------------|-----------|-----------------|-----------------|
| Retail.Small      | 45,876         | 24,576    | 540             | 13,279,232      |
| Retail.Medium     | 458,765        | 24,576    | 5,398           | 132,669,440     |
| Retail.Large      | 4,587,654      | 24,576    | 53,973          | 1,326,448,640   |

1. Small Data File without the SASFILE Statement
2. Medium Data File without the SASFILE Statement
3. Large Data File without the SASFILE Statement
4. Small Data File with the SASFILE Statement
5. Medium Data File with the SASFILE Statement
6. Large Data File with the SASFILE Statement.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for using the SASFILE statement.

## Programming Techniques

### 1. Small Data File without the SASFILE Statement

This program creates reports using the PRINT, TABULATE, MEANS, and FREQ procedures. The SAS data file **Retail.Small** is opened and closed with each procedure.

```
proc print data=retail.small;
  where cs=100;
  var Customer_Age_Group;
run;
proc tabulate data=retail.small;
  class Customer_Age_Group;
  var Customer_BirthDate;
  table Customer_Age_Group,Customer_BirthDate*(mean median);
run;
proc means data=retail.small;
  var Customer_Age;
  class Customer_Group;
  output out=summary sum=;
run;
proc freq data=retail.small;
  tables Customer_Country;
run;
```

### 2. Medium Data File without the SASFILE Statement

This program creates reports using the PRINT, TABULATE, MEANS, and FREQ procedures. The SAS data file **Retail.Medium** is opened and closed with each procedure.

```
proc print data=retail.medium;
  where cm=100;
  var Customer_Age_Group;
run;
proc tabulate data=retail.medium;
  class Customer_Age_Group;
  var Customer_BirthDate;
  table Customer_Age_Group,Customer_BirthDate*(mean median);
run;
proc means data=retail.medium;
  var Customer_Age;
  class Customer_Group;
  output out=summary sum=;
run;
proc freq data=retail.medium;
  tables Customer_Country;
run;
```

### 3. Large Data File without the SASFILE Statement

This program creates reports using the PRINT, TABULATE, MEANS, and FREQ procedures. The SAS data file **Retail.Large** is opened and closed with each procedure.

```
proc print data=retail.large;
  where cl=100;
  var Customer_Age_Group;
run;
```

```

proc tabulate data=retail.large;
  class Customer_Age_Group;
  var Customer_BirthDate;
  table Customer_Age_Group,Customer_BirthDate*(mean median);
run;
proc means data=retail.large;
  var Customer_Age;
  class Customer_Group;
  output out=summary sum=;
run;
proc freq data=retail.large;
  tables Customer_Country;
run;

```

#### **4. Small Data File with the SASFILE Statement**

In this program, the SASFILE LOAD statement opens the SAS data file **Retail.Small** and loads the entire file into memory. The data is then available to the PRINT, TABULATE, MEANS, and FREQ procedures. The SASFILE CLOSE statement closes **Retail.Small** and frees the buffers.

```

sasfile retail.small load;
proc print data=retail.small;
  where cs=100;
  var Customer_Age_Group;
run;
proc tabulate data=retail.small;
  class Customer_Age_Group;
  var Customer_BirthDate;
  table Customer_Age_Group,Customer_BirthDate*(mean median);
run;
proc means data=retail.small;
  var Customer_Age;
  class Customer_Group;
  output out=summary sum=;
run;
proc freq data=retail.small;
  tables Customer_Country;
run;
sasfile retail.small close;

```

#### **5. Medium Data File with the SASFILE Statement**

In this program, the SASFILE LOAD statement opens the SAS data file **Retail.Medium** and loads the entire file into memory. The data is then available to the PRINT, TABULATE, MEANS, and FREQ procedures. The SASFILE CLOSE statement closes **Retail.Medium** and frees the buffers.

```

sasfile retail.medium load;
proc print data=retail.medium;
  where cm=100;
  var Customer_Age_Group;
run;
proc tabulate data=retail.medium;
  class Customer_Age_Group;
  var Customer_BirthDate;
  table Customer_Age_Group,Customer_BirthDate*(mean median);
run;
proc means data=retail.medium;

```

```

var Customer_Age;
class Customer_Group;
output out=summary sum=;
run;
proc freq data=retail.medium;
tables Customer_Country;
run;
sasfile retail.medium close;

```

## 6. Large Data File with the SASFILE Statement

In this program, the SASFILE LOAD statement opens the SAS data file **Retail.Large** and loads the entire file into memory. The data is then available to the PRINT, TABULATE, MEANS, and FREQ procedures. The SASFILE CLOSE statement closes **Retail.Large** and frees the buffers.

```

sasfile retail.large load;
proc print data=retail.large;
where cl=100;
var Customer_Age_Group;
run;
proc tabulate data=retail.large;
class Customer_Age_Group;
var Customer_BirthDate;
table Customer_Age_Group,Customer_BirthDate*(mean median);
run;
proc means data=retail.large;
var Customer_Age;
class Customer_Group;
output out=summary sum=;
run;
proc freq data=retail.large;
tables Customer_Country;
run;
sasfile retail.large close;

```

## General Recommendations

- If you need to repeatedly process a SAS data file that will fit entirely in memory, use the SASFILE statement to reduce I/O and some CPU usage.
- If you use the SASFILE statement and the SAS data file will not fit entirely in memory, the code will execute, but there might be a degradation in performance.
- If you need to repeatedly process part of a SAS data file and the entire file won't fit into memory, use a DATA step with the SASFILE statement to create a subset of the file that does fit into memory, and then process that subset repeatedly. This saves CPU time in the processing steps because those steps will read a smaller file, in addition to the benefit of the file being resident in memory.

### The SASFILE OPEN statement

**Q.5.**

- opens a SAS data file and allocates the buffers, but defers reading the data into memory until a procedure or statement is executed.
- opens a SAS data file, allocates the buffers, and reads the data into memory.
- opens a SAS data file, allocates the buffers, reads the data into memory, and closes the file.

Which of the following is true about the SASFILE statement?

- Q.6.**
- It always increases I/O processing.
  - It always reduces I/O processing.
  - It reduces I/O processing only if there is sufficient real memory.

## Additional Features

### Using the IBUFSIZE= System Option

Beginning with SAS 9, you can use the IBUFSIZE= system option to specify the page size for an index file. Typically, you do not need to specify an index page size. However, you might need to use the IBUFSIZE= option if

- there are many levels in the index
- the length of an index value is very large.

The main resource that is saved when reducing levels in the index is I/O. If your application is experiencing a lot of I/O in the index file, increasing the page size might help. However, you must re-create the index file after increasing the page size. The number of pages that are required for the index varies with the page size, the length of the index value, and the values themselves.

#### General form, IBUFSIZE= system option:

**IBUFSIZE= MIN | MAX | n;**

where

- MIN sets the page size for an index file to -32767. The IBUFSIZE= option is defined as a signed integer so that negative values can be supplied for internal testing purposes. This might cause unexpected results.
- MAX sets the page size for an index file to the maximum possible number. For IBUFSIZE=, the maximum value is 32,767 bytes.
- *n* specifies the page size in bytes.



The MIN setting should be avoided.

When an index is used to process a request, such as for WHERE processing, SAS searches the index file in order to rapidly locate the requested record(s). The page size affects the number of levels in the index. The more pages there are, the more levels in the index. The more levels, the longer the index search takes. Increasing the page size allows more index values to be stored on each page, thus reducing the number of pages (and the number of levels).

Use IBUFSIZE=0 to reset the index page size to the default value in your operating environment.



For details on using the IBUFSIZE= system option, see the SAS documentation.

## **Chapter Summary**

### **Controlling Page Size and the Number of Buffers**

When you read a SAS data set or an external file, I/O is measured when the input data is copied to the buffer in memory and when it is read from the output buffer to the output data set.

A page is the unit of data transfer between the storage device and memory. When you create a SAS data set, SAS takes the data and copies it to a buffer. Each buffer can hold one page of data.

The amount of data that can be transferred to one buffer in a single I/O operation is referred to as the page size. Increasing the page size can speed up execution time by reducing the number of times SAS has to read from or write to the storage medium. You can use the CONTENTS procedure to report the page size and the number of pages.

You can use the BUFSIZE= system option or data set option to control the page size of an output SAS data set. The new buffer size is permanent; after it is specified, it is used whenever the data set is processed.

You can use the BUFNO= system or data set option to control how many buffers are available for reading or writing a SAS data set. By increasing the number of buffers, you can control how many pages of data are loaded into memory with each I/O transfer.

The product of BUFNO= and BUFSIZE=, rather than the specific value of either option, determines how much data can be transferred in one I/O operation. Increasing either option increases the amount of data that can be transferred in one I/O operation. However, the improvement in I/O comes at the cost of increased memory consumption.

Review the related comparative example:

- Using the BUFSIZE= Option and the BUFNO= Option.

### **Using the SASFILE Statement**

Another way of improving performance is to use the SASFILE statement to hold a SAS data file in memory so that the data is available to multiple program steps. Keeping the data set open reduces open/close operations, including the allocation and freeing of memory for buffers.

When the SASFILE statement executes, SAS allocates the number of buffers based on the number of pages for the data file and index file. If the file in memory increases in size during processing because of changes or additions to the data, the number of buffers also increases.

It is important to note that I/O processing is reduced only if there is sufficient real memory. If SAS uses virtual memory, there can be a degradation in performance.

Review the related comparative example:

- Using the SASFILE Statement.

### **Additional Features**

The IBUFSIZE= system option specifies the page size for an index file. Typically, you do not need to specify an index page size. However, you might need to use the IBUFSIZE= option if

- there are many levels in the index
- the length of an index value is very large.

The main resource that is saved when reducing levels in the index is I/O. If your application is experiencing a lot of I/O in the index file, increasing the page size might help. However, you must re-create the index file after increasing the page size. The number of pages that are required for the index varies with the page size, the length of the index value, and the values themselves.

## Chapter Quiz

Select the best answer for each question.

1. Which of the following statements is true regarding the BUFNO= option?
  - a. The BUFNO= option specifies the size of each buffer that is used for reading or writing a SAS data set.
  - b. The BUFNO= option can improve execution time by limiting the number of input/output operations that are required.
  - c. Using the BUFNO= option results in permanent changes to the data set.
  - d. Using the BUFNO= option to increase the number of buffers results in decreased memory consumption.
2. Which of the following statements is **not** true regarding a page?
  - a. A page is the unit of data transfer between the engine and the operating environment.
  - b. A page includes the number of bytes used by the descriptor portion, the data values, and the overhead.
  - c. The size of a page is analogous to buffer size.
  - d. The size of a page can be changed at any time.
3. The total number of bytes occupied by a data set equals...?
  - a. the page size multiplied by the number of pages.
  - b. the page size multiplied by the number of observations.
  - c. the sum of the page size and the number of pages.
  - d. the number of pages multiplied by the number of variables.
4. Which statement opens the file **Work.Qarter1**, allocates enough buffers to hold the entire file in memory, and reads the data into memory?
  - a. sasfile work.qarter1 open;
  - b. sasfile work.qarter1 load;
  - c. sasfile work.qarter1 bufno=max;
  - d. sasfile work.qarter1 bufsize=max;
5. Which of the following statements is true regarding a file that is opened with the SASFILE statement?
  - a. The file is available to subsequent DATA and PROC steps or applications until a SASFILE CLOSE statement is executed or until the program ends.
  - b. The file is available to subsequent DATA and PROC steps or applications until a SASFILE END statement is executed.
  - c. The file is available for subsequent utility or output processing until the program ends.
  - d. If the file increases in size during processing, the number of buffers remains the same.

# Chapter 9

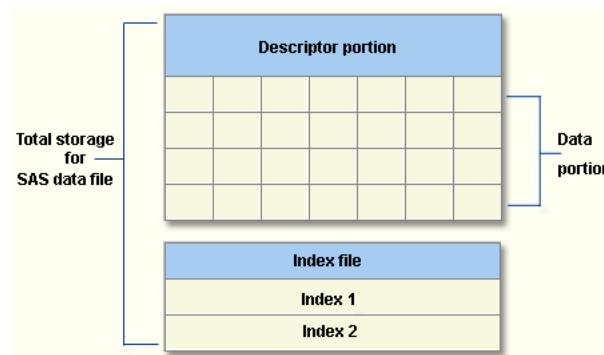
## Controlling Data Storage Space

### Introduction

In many computing environments, data storage space is a limited resource. Therefore, it might be more important for you to conserve data storage space than to conserve other resources.

When you store your data in a SAS data file, you use the sum of the data storage space that is required for the following:

- the descriptor portion
- the observations
- any storage overhead
- any associated indexes.



In this lesson you learn to use a variety of techniques for minimizing the amount of space that your SAS data files occupy.



This lesson does not cover the Scalable Performance Data Engine (SPD Engine), which is a SAS 9.1 technology for [threaded processing](#). For details about using the SPD Engine to improve performance, see the SAS documentation.

### Objectives

In this lesson, you learn to

- describe how SAS stores character variables
- determine how to reduce the length of character variables and how to expand the values
- describe how SAS stores numeric variables
- determine how to safely reduce the space that is required for storing numeric variables in SAS data sets
- define the structure of a compressed SAS data file
- create a compressed SAS data file
- examine the advantages and disadvantages of compression
- describe the difference between a SAS data file and a SAS data view
- examine the advantages and disadvantages of DATA step views.

## Reducing Data Storage Space for Character Variables

One way to reduce the amount of data storage space that you need is to reduce the length of character data, thereby eliminating wasted space. Before discussing how to reduce the length of a character variable, let's look at how SAS assigns lengths to character variables.

### How SAS Assigns Lengths to Character Variables

SAS character variables store data as 1 character per byte. A SAS character variable can be from 1 to 32,767 bytes in length.

The first reference to a variable in the DATA step defines it in the program data vector and in the descriptor portion of the data set. Unless otherwise defined, the first value that is specified for a SAS character variable determines the variable's length. For example, if the length of a character variable called *Name* has not been defined and if the first value that is specified for it is *Smith*, the length of *Name* is set to 5. Then, if the next value specified for *Name* is *Johnson*, the value is stored as *Johns* in the data set. Similarly, if the first value specified for *Town* is *Williamsburg*, the length of *Town* is set to 12. If the next value for *Town* is specified as *Cary*, the length is still 12, and the value is padded with blanks to fill the extra space.

Keep in mind that the first reference to a variable might not be an assignment statement. If SAS cannot determine a length for a character variable when the variable is created in the program data vector, SAS assigns a default length of 8 bytes to the variable.

### Reducing the Length of Character Data with the LENGTH Statement

You can use a LENGTH statement to reduce the length of character variables. It is useful to reduce the length of a character variable with a LENGTH statement when you have a large data set that contains many character variables.

#### General form, LENGTH statement for character variables:

**LENGTH** *variable(s)* \$ *length*;

where

- *variable(s)* specifies the name of one or more SAS variables, separated by spaces.
- *length* is an integer from 1 to 32,767 that specifies the length of the variable(s).



Make sure the LENGTH statement appears before any other reference to the variable in the DATA step. If the variable has been created by another statement, then a later use of the LENGTH statement will not change its size.

### Other Techniques

There are other techniques that you can use to reduce the length of your character data—especially if your data set has repeated values.

For example, suppose you have a data set that records employee names along with the employees' departments. Instead of recording the complete department name in the data set, you could assign a code to each department and record the code in your data set instead. You could record the complete department name along with the corresponding code in a separate data set, where you would have to

record each full-length value only once. Then you could use a table lookup operation to convert the code to the department name for reporting purposes. This is called normalizing the data.

The tables below represent two data sets. **Employees** records the employee names and department codes for all employees. **DeptCodes** records the department name and department code, and is shown in its entirety. If the **Employees** data set contains several hundred observations, then using department codes instead of the complete department names can save a substantial amount of data storage space.

**Employees** (partial listing)

| LastName   | FirstName   | Dept |
|------------|-------------|------|
| Mills      | Dorothy E.  | FOP  |
| Bower      | Eileen A.   | FIT  |
| Reading    | Tony R.     | HRS  |
| Judd       | Carol A.    | HRS  |
| Wonsild    | Hanna       | AOP  |
| Anderson   | Christopher | SMK  |
| Massengill | Annette W.  | FOP  |
| Badine     | David       | COP  |

**DeptCodes**

| Dept | Department           |
|------|----------------------|
| FOP  | Airport Operations   |
| FIT  | Corporate Operations |
| HRS  | Finance & IT         |
| AOP  | Flight Operations    |
| SMK  | Human Resources      |
| COP  | Sales & Marketing    |



You can learn about table lookup operations in the lesson [Combining Data Horizontally](#).

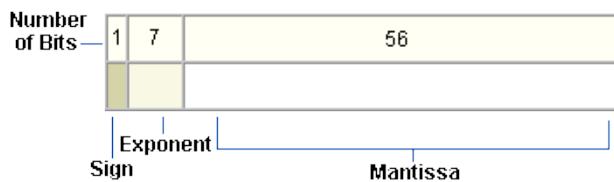
## Reducing Data Storage Space for Numeric Variables

Another way to eliminate wasted space and thereby to reduce the amount of data storage space that you need is to reduce the length of **numeric** variables. In addition to conserving data storage space, reduced-length numeric variables use **less I/O**, both when data is written and when it is read. For a file that is read frequently, this savings can be significant. However, in order to safely reduce the length of numeric variables, you need to understand how SAS stores numeric data.

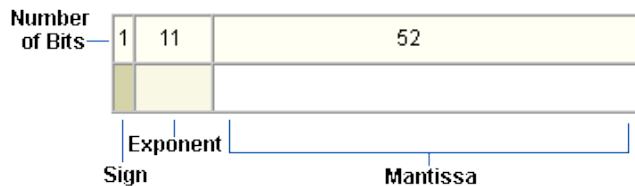
### How SAS Stores Numeric Variables

To store numbers of large magnitude and to perform computations that require many digits of precision to the right of the decimal point, SAS stores all numeric values using double-precision [floating-point representation](#). SAS stores the value of a numeric variable as multiple digits per byte. A SAS numeric variable can be from 2 to 8 bytes or 3 to 8 bytes in length, depending on your operating environment. The default length for a numeric variable is 8 bytes.

The figures below show how SAS stores a numeric value in 8 bytes. For mainframe environments, the first bit stores the sign, the next seven bits store the exponent of the value, and the remaining 56 bits store the mantissa.



For non-mainframe environments, the first bit stores the sign, the next eleven bits store the exponent of the value, and the remaining 52 bits store the mantissa.



The minimum length for a numeric variable is 2 bytes in mainframe environments and 3 bytes in non-mainframe environments.

Now that you have seen how SAS stores numeric variables, let's look at how you can assign a length to your numeric variables that is less than the default length of 8 bytes.

### Floating-Point Representation

Floating-point representation is an implementation of what is generally known as scientific notation, in which values are represented as numbers between 0 and 1 times a power of 10. The following is an example of a number in scientific notation:

$$0.1234 \times 10^4$$

Numbers in scientific notation are comprised of the following parts:

- The **mantissa** is the number that is multiplied by the base; in this example, the mantissa is .1234.
- The **base** is the number that is raised to a power; in this example, the base is 10.
- The **exponent** is the power to which the base is raised; in this example, the exponent is 4.

Floating-point representation is a form of scientific notation, except that on most operating systems the base is not 10, but is either 2 (non-mainframe) or 16 (mainframe). The following table summarizes various representations of floating-point numbers that are stored in 8 bytes.

| Representation       | Base | Exponent Bits | Maximum Mantissa Bits |
|----------------------|------|---------------|-----------------------|
| IBM Mainframe        | 16   | 7             | 56                    |
| IEEE (non-mainframe) | 2    | 11            | 52                    |

For example, the number 35298 can also be written in floating-point representation, as shown in the image below.

+0.35298 \* (10\*\*5)

In a SAS program, a number can be written in the following ways:

- 35298
- 3529.8E+01
- 352.98E+02
- 35.298E+03

- and so on.

The mantissa defines the significant digits of a value. Values that have 16 or 17 significant digits (depending on the operating environment) can be represented accurately with 8 bytes.

## Assigning Lengths to Numeric Variables

You can use a **LENGTH statement** to assign a length from 2 to 8 bytes to numeric variables. Remember, the minimum length of numeric variables depends on the operating environment. Also, keep in mind that the LENGTH statement affects the length of a numeric variable only in the output data set. Numeric variables **always** have a length of 8 bytes in the program data vector and during processing.

### General form, LENGTH statement for numeric variables:

**LENGTH** *variable(s)* *length* <**DEFAULT=n**>;

where

- *variable(s)* specifies the name of one or more numeric SAS variables, separated by spaces.
- *length* is an integer that specifies the length of the variable(s).
- the optional **DEFAULT=n** argument changes the default number of bytes that SAS uses to store the values of any newly created numeric variables. If you use the **DEFAULT=** argument, you do not need to list any *variable(s)*.



Values between 2 and 8 or between 3 and 8 depending on your operating environment are valid for *n* or *length*.

**DEFAULT=** applies only to numeric variables that are added to the program data vector after the LENGTH statement is compiled. You would list specific variables in the LENGTH statement along with the **DEFAULT=** argument only if you wanted those variables to have a length other than the value for **DEFAULT=**. If you list individual variables in the LENGTH statement, you must list an integer length for each of them.



You should assign reduced lengths to numeric variables only if those variables have integer values. Fractional numbers lose precision if truncated. You will learn more about the loss of precision with reduced-length numeric variables on the next page of this lesson.

## Example

The following program assigns a length of 4 to the new variable **Sale\_Percent** in the data set **ReducedSales**. The LENGTH statement in this DATA step does not apply to the variables that are read in from the **Sales** data set; those variables will maintain whatever length they had in **Sales** when they are read into **ReducedSales**.

```
data reducedsales;
length default=4;
set sales;
Sale_Percent=15;
run;
```

## Maintaining Precision in Reduced-Length Numeric Variables

There is a limit to the values that you can precisely store in a reduced-length numeric variable. You have learned that reducing the number of bytes that are used for storing a numeric variable does **not** affect how the numbers are stored in the program data vector. Instead, specifying a value of less than 8 in the LENGTH statement causes the number to be truncated to the specified length when the value is written to the SAS data set.

You should never use the LENGTH statement to reduce the length of your numeric variables if the values are not integers. Fractional numbers lose precision if truncated. Even if the values are integers, you should keep in mind that reducing the length of a numeric variable limits the integer values that can accurately be stored as a value.

The following table lists the possible storage length for integer values on UNIX or Windows operating environments.

| UNIX/Windows   |                                     |
|----------------|-------------------------------------|
| Length (bytes) | Largest Integer Represented Exactly |
| 3              | 8,192                               |
| 4              | 2,097,152                           |
| 5              | 536,870,912                         |
| 6              | 137,438,953,472                     |
| 7              | 35,184,372,088,832                  |
| 8              | 9,007,199,254,740,992               |

The following table lists the possible storage length for integer values on the z/OS operating environment.

| z/OS           |                                     |
|----------------|-------------------------------------|
| Length (bytes) | Largest Integer Represented Exactly |
| 2              | 256                                 |
| 3              | 65,536                              |
| 4              | 16,777,216                          |
| 5              | 4,294,967,296                       |
| 6              | 1,099,511,627,776                   |
| 7              | 281,474,946,710,656                 |
| 8              | 72,057,594,037,927,936              |

When you store an integer that is equal to or less than the number listed above as the largest integer that can be represented exactly in a reduced-length variable, SAS truncates bytes that contain only zeros. If the integer that is stored in a reduced-length variable is larger than the recommended limit, SAS truncates bytes that contain numbers other than zero, and the integer value is changed. Similarly, you should not reduce the stored size of non-integer data because it can result in a loss of precision due to the truncation of nonzero bytes.

If you decide to reduce the length of your numeric variables, you might want to verify that you have not lost any precision in your values. Let's look at one way to do this.

### Using PROC COMPARE

You can use PROC COMPARE to gauge the precision of the values that are stored in a shortened numeric variable by comparing the original variable with the shortened variable. The COMPARE

procedure compares the contents of two SAS data sets, selected variables in different data sets, or variables within the same data set.

**General form, PROC COMPARE step to compare two data sets:**

```
PROC COMPARE BASE=SAS-data-set-one COMPARE=SAS-data-set-two;  
RUN;
```

- where *SAS-data-set-one* and *SAS-data-set-two* specify the two SAS data sets that you want to compare.

PROC COMPARE is a good technique to use for gauging the loss of precision in shortened numeric variables because it shows you whether there are differences in the stored numeric values even if these differences do not show up once the numeric variables have been formatted. PROC COMPARE looks at the two data sets and compares their

- data set attributes
- variables
- variable attributes for matching variables
- observations
- values in matching variables.

Output from the COMPARE procedure includes

- a data set summary
- a variables summary
- a listing of common variables that have different attributes
- an observation summary
- a values comparison summary
- a listing of variables that have unequal values
- a detailed list of value comparison results for variables.

**Example**

The data set **Company.Discount** contains data about sale dates and discounts for certain retail products. There are 35 observations in **Company.Discount**, which is described below.

| Variable         | Type | Length | Description                               |
|------------------|------|--------|-------------------------------------------|
| Product_ID       | num  | 8      | product ID number                         |
| Start_Date       | num  | 4      | start date of sale                        |
| End_Date         | num  | 5      | end date of sale                          |
| Unit_Sales_Price | num  | 8      | discounted sales price per unit           |
| Discount         | num  | 8      | discount as percent of normal sales price |

Suppose you shorten the length of the numeric variable *Discount*. The DATA step below creates a new data set named **Company.Discount\_Short**, whose only difference from **Company.Discount** is that the length of the variable *Discount* is 4 instead of 8.

```
data company.discount_short;  
length Discount 4;  
set Company.Discount;
```

```
run
```

You can use PROC COMPARE to evaluate whether shortening the length of Discount affects the precision of its values by comparing **Company.Discount** to **Company.Discount\_Short**.

```
proc compare base=company.discount  
            compare=company.discount_short;  
run;
```

If you were to print these two data sets (**Company.Discount** and **Company.Discount\_Short**), the values might appear to be identical. However, there are differences in the values as they are stored that are not apparent in the formatted output.

In the partial output below, you can see that shortening the length of Discount results in a loss of precision in its values; the values for Discount in **Company.Discount\_Short** differ by a maximum of **1.9073E-07**. The value comparison results show that although the values for Discount in the first five observations appear as 70% in both data sets, the precise (unformatted) values differ by **-1.907E-7**.

#### Partial PROC COMPARE Output

##### Variables Summary

Number of Variables in Common: 5.  
Number of Variables with Differing Attributes: 1.

##### Listing of Common Variables with Differing Attributes

| Variable | Dataset | Type | Length | Format |
|----------|---------|------|--------|--------|
|----------|---------|------|--------|--------|

|          |                        |     |   |          |
|----------|------------------------|-----|---|----------|
| Discount | COMPANY.DISCOUNT       | Num | 8 | PERCENT. |
|          | COMPANY.DISCOUNT_SHORT | Num | 4 | PERCENT. |

##### Values Comparison Summary

Number of Variables Compared with All Observations Equal: 4.  
Number of Variables Compared with Some Observations Unequal: 1.  
Total Number of Values which Compare Unequal: 35.  
**Maximum Difference: 1.9073E-07.**

##### Value Comparison Results for Variables

| Discount as Percent of Normal Retail Sal |  |          |          |                  |           |
|------------------------------------------|--|----------|----------|------------------|-----------|
| .. es Price                              |  |          |          |                  |           |
| Base Compare                             |  |          |          |                  |           |
| Obs                                      |  | Discount | Discount | Diff.            | % Diff    |
|                                          |  |          |          |                  |           |
| 1                                        |  | 70%      | 70%      | <b>-1.907E-7</b> | -0.000027 |
| 2                                        |  | 70%      | 70%      | <b>-1.907E-7</b> | -0.000027 |
| 3                                        |  | 70%      | 70%      | <b>-1.907E-7</b> | -0.000027 |
| 4                                        |  | 70%      | 70%      | <b>-1.907E-7</b> | -0.000027 |
| 5                                        |  | 70%      | 70%      | <b>-1.907E-7</b> | -0.000027 |

Which of the following statements regarding reduced-length numeric variables is false?

- Q.1. Reduced-length numeric variables have a length of 8 bytes in the program data vector.
- Reduced-length numeric variables are truncated when they are read.
- Reduced-length numeric variables can be stored in 2 or 3 to 8 bytes, depending on the operating environment.
- Reduced-length numeric variables should be integer values.



### Comparative Example: Creating a SAS Data Set That Contains Reduced-Length Numeric Variables

Suppose you want to create a SAS data set in which to store retail data about a group of orders. Suppose that the data you want to include in your data set is all numeric data and that it is currently stored in a raw data file. You can create the data set using

1. Default-Length Numeric Variables
2. Reduced-Length Numeric Variables

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for creating reduced-length numeric variables.



Throughout this course, the keyword `_NULL_` is often used as the data set name in sample programs. Using `_NULL_` causes SAS to execute the DATA step as if it were creating a new data set. However, no observations or variables are written to an output data set. Using `_NULL_` when benchmarking enables you to determine what resources are used to read a SAS data set.

## Programming Techniques

### 1. Default-Length Numeric Variables

This program reads the external data file that is referenced by the fileref `flat1` and creates a new data set called `Retail.Longnums` that contains 12 numeric variables. Each of the variables in `Retail.Longnums` has the default storage length of 8 bytes. The second DATA step in this program reads the numeric variables from `Retail.Longnums`.

```
data retail.longnums;
  infile flat1;
  input Customer_ID      12.
        Employee_ID     12.
        Street_ID       12.
        Order_Date      date9.
        Delivery_Date   date9.
        Order_ID        12.
        Order_Type      commal6.
        Product_ID      12.
        Quantity        4.
        Total_Retail_Price dollar13.2
        CostPrice_Per_Unit dollar13.2
        Discount        5.2    ;
run;
```

```
data _null_;
  set retail.longnums;
run;
```

## 2. Reduced-Length Numeric Variables

This program reads the external data file that is referenced by the fileref **flat1** and creates a new SAS data set called **Retail.Shortnums** that contains 12 numeric variables. A LENGTH statement is used to reduce the storage length of most of the numeric variables in **Retail.Shortnums**, as follows:

- Total\_Retail\_Price and CostPrice\_Per\_Unit have a storage length of 8 bytes.
- Product\_ID has a storage length of 7 bytes.
- Street\_ID and Order\_ID have a storage length of 6 bytes.
- Employee\_ID has a storage length of 5 bytes.
- Customer\_ID, Order\_Date, Delivery\_Date and Discount have a storage length of 4 bytes.
- Order\_Type and Quantity have a storage length of 3 bytes.

The second DATA step reads the reduced-length numeric variables from **Retail.Shortnums**.

```
data retail.shortnums;
  infile flat1;
  length Quantity      3
        Customer_ID Order_Date
        Delivery_Date Discount 4
        Employee_ID      5
        Street_ID Order_ID   6
        Product_ID       7
        Total_Retail_Price
        CostPrice_Per_Unit 8;
  input Customer_ID    12.
        Employee_ID     12.
        Street_ID       12.
        Order_Date      date9.
        Delivery_Date   date9.
        Order_ID        12.
        Order_Type      comma16.
        Product_ID      12.
        Quantity        4.
        Total_Retail_Price dollar13.2
        CostPrice_Per_Unit dollar13.2
        Discount        5.2  ;
  run;
  data _null_;
    set retail.shortnums;
  run;
```

 Remember that when you reduce the storage length of numeric variables, you risk losing precision in their values. You can use PROC COMPARE to verify the precision of shortened numeric variables.

```
proc compare data=retail.longnums;
  compare=retail.shortnums;
run;
```

## General Recommendations

- Create reduced-length numeric variables for integer values when you need to conserve data storage space.

What is the advantage of reducing the length of numeric variables in a data set?

Reducing the length of numeric variables reduces the amount of space that is required for storing them in the program data vector.

**Q.2.** Reducing the length of the numeric variables in a data set reduces the overall size of the data set.

Reducing the length of the numeric variables in a data set increases the precision with which the variable's values are stored.

Reducing the length of the numeric variables increases the amount of I/O that is required for processing the data set.

## Compressing Data Files

Compressing your data files is another method that you can use to conserve data storage space. Compressing a data file is a process that reduces the number of bytes that are required in order to represent each observation in a data file.

Reading from or writing to a compressed file during data processing requires fewer I/O operations because there are fewer data set pages in a compressed data file. However, in order to read a compressed file, each observation must be uncompressed. This requires more CPU resources than reading an uncompressed file. Also, in some cases, compressing a file might actually increase its size rather than decreasing it.

Before examining how to compress a data file, let's review the structure of an uncompressed data file as compared to the structure of a compressed data file.

### Review of Uncompressed Data File Structure

By default, a SAS data file is not compressed. In uncompressed data files,

- each data value of a particular variable occupies the same number of bytes as any other data value of that variable.
- each observation occupies the same number of bytes as any other observation.
- character values are padded with blanks.
- numeric values are padded with binary zeros.
- there is a 16-byte overhead at the beginning of each page.
- there is a 1-bit per observation overhead (rounded up to the nearest byte) at the end of each page; this bit denotes an observation's status as deleted or not deleted.
- new observations are added at the end of the file. If a new observation won't fit on the current last page of the file, a whole new data set page is added.
- the descriptor portion of the data file is stored at the end of the first page of the file.

The figure below depicts the structure of an uncompressed data file.

| Page<br>1 | 16 byte OH | Obs 1 | Obs 2 | Obs 3        | Un-used Space | 1 bit per Obs OH | Descriptor       |
|-----------|------------|-------|-------|--------------|---------------|------------------|------------------|
| Page 2    | 16 byte OH | Obs 4 | Obs 5 | Obs 6        | Obs 7         | Unused Space     | 1 bit per Obs OH |
| :         | :          | :     | :     | :            | :             |                  | :                |
| Page n    | 16 byte OH | Obs y | Obs z | Unused Space |               |                  | 1 bit per Obs OH |

In comparison, let's look at the characteristics of a compressed data file.

### Compressed Data File Structure

Compressed data files

- treat an observation as a single string of bytes by ignoring variable types and boundaries.
- collapse consecutive repeating characters and numbers into fewer bytes.
- contain a 28-byte overhead at the beginning of each page.
- contain a 12-byte- or 24-byte-per-observation overhead following the page overhead. This space is used for deletion status, compressed length, pointers, and flags.

Each observation in a compressed data file can have a different length, which means that some pages in the data file can store more observations than others can. When an updated observation is larger than its original size, it is stored on the same data file page and uses available space. If not enough space is available on the original page, the observation is stored on the next page that has enough space, and a pointer is stored on the original page.

The image below depicts the structure of a compressed data file.

| Page<br>1 | 28 byte OH | 12 or 24 bytes per Obs OH | Unused Space | Obs 3 | Obs 2 | Obs 1 | Descriptor  |
|-----------|------------|---------------------------|--------------|-------|-------|-------|-------------|
| Page 2    | 28 byte OH | 12 or 24 bytes per Obs OH | Unused Space | Obs 8 | Obs 7 | Obs 6 | Obs 5 Obs 4 |
| :         | :          | :                         | :            | :     | :     |       | :           |
| Page n    | 28 byte OH | 12 or 24 bytes per Obs OH | Unused Space |       | Obs z | Obs y | Obs x       |



The overhead for each observation is 12 bytes per observation on 32-bit operating environments and 24 bytes per observation on 64-bit operating environments.

### Deciding Whether to Compress a Data File

Not all data files are good candidates for compression. Remember that in order for SAS to read a compressed file, each observation must be uncompressed. This requires more CPU resources than

reading an uncompressed file. However, compression can be beneficial when the data file has one or more of the following properties:

- It is large.
- It contains many long character values.
- It contains many values that have repeated characters or binary zeros.
- It contains many missing values.
- It contains repeated values in variables that are physically stored next to one another.

In character data, the most frequently encountered repeated value is the blank. Long text fields, such as comments and addresses, often contain repeated blanks. Likewise, binary zeros are used to pad numeric values that can be stored in fewer bytes than are available in a particular numeric variable. This happens most often when you assign a small or medium-sized integer to an 8-byte numeric variable.

 If saving disk space is crucial, consider storing missing data as a small integer, such as 0 or 9, rather than as a SAS missing value. Small integers can be compressed more than SAS missing values can.

A data file is **not** a good candidate for compression if it has

- few repeated characters
- small physical size
- few missing values
- short text strings.

Next, let's look at how to compress a data file.

### The COMPRESS= System Option and the COMPRESS= Data Set Option

To compress a data file, you use either the COMPRESS= data set option or the COMPRESS= system option. You use the **COMPRESS= system option** to compress all data files that you create during a SAS session. Similarly, you use the **COMPRESS= data set option** to compress an individual data file.

#### General form, COMPRESS= system option:

**OPTIONS COMPRESS= NO | YES | CHAR | BINARY ;**

where

- **NO** is the default setting, which does not compress the data set.
- **CHAR** or **YES** uses the Run Length Encoding (RLE) compression algorithm, which compresses repeating consecutive bytes such as trailing blanks or repeated zeros.
- **BINARY** uses Ross Data Compression (RDC), which combines run-length encoding and sliding-window compression.



If you set the COMPRESS= system option to a value other than **NO**, SAS compresses every data set that is created during the current SAS session, including temporary data sets in the **Work** library. Although this might conserve data storage space, it will also use greater amounts of other resources.

#### General form, COMPRESS= data set option:

```
DATA SAS-data-set (COMPRESS= NO | YES | CHAR | BINARY);
```

where

- *SAS-data-set* specifies the data set that you want to compress.
- **NO** is the default setting, which does not compress the data set.
- **CHAR** or **YES** uses the Run Length Encoding (RLE) compression algorithm, which compresses repeating consecutive bytes such as trailing blanks or repeated zeros.
- **BINARY** uses Ross Data Compression (RDC), which combines run-length encoding and sliding-window compression.



The COMPRESS= data set option overrides the COMPRESS= system option.

The **YES** or **CHAR** setting for the COMPRESS= option uses the RLE compression algorithm. RLE compresses observations by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations. Therefore, RLE is most often useful for character data that contains repeated blanks. The **YES** or **CHAR** setting is also good for compressing numeric data in which most of the values are zero.

The **BINARY** setting for the COMPRESS= option uses RDC, which combines run-length encoding and sliding-window compression. This method is highly effective for compressing medium to large blocks of binary data (numeric variables).

A file that has been compressed using the **BINARY** setting of the COMPRESS= option takes significantly more CPU time to uncompress than a file that was compressed with the **YES** or **CHAR** setting. **BINARY** is more efficient with observations that are several hundred bytes or more in length. **BINARY** can also be very effective with character data that contains patterns rather than simple repetitions.

When you create a compressed data file, SAS compares the size of the compressed file to the size of the uncompressed file of the same page size and record count. Then SAS writes a note to the log indicating the percentage of reduction that is obtained by compressing the file.

When you use either of the COMPRESS= options, SAS calculates the size of the overhead that is introduced by compression as well as the maximum size of an observation in the data set that you are attempting to compress. If the maximum size of the observation is smaller than the overhead that is introduced by compression, SAS disables compression, creates an uncompressed data set, and issues a warning message stating that the file was not compressed. This feature is available beginning in SAS 8.2.

Once a file is compressed, the setting is a permanent attribute of the file. In order to change the setting to uncompressed, you must re-create the file.



Compression of observations is not supported by all SAS engines. See the SAS documentation for the COMPRESS= data set option for more information.

## Example

The data set **Company.Customer** contains demographic information about a retail company's customers. The data set includes character variables such as *Customer\_Name*, *Customer\_FirstName*, *Customer\_LastName*, and *Customer\_Address*. These character variables have the potential to contain many repeated blanks in their values. The following program will create a compressed data set named **Company.Customers\_Compressed** from **Company.Customer** even if the COMPRESS= system option is set to **NO**.

```
data company.customer_compressed (compress=char);
  set company.customer;
run;
```

SAS writes a note to the SAS log about the compression of the new data set, as shown below.

#### SAS Log

```
NOTE: There were 89954 observations read from the data
      set COMPANY.CUSTOMER.
NOTE: The data set COMPANY.CUSTOMER_COMPRESSED has 89954
      observations and 11 variables.
NOTE: Compressing data set COMPANY.CUSTOMER_COMPRESSED
decreased size by 32.81 percent.
Compressed is 991 pages; un-compressed would require
1475 pages.
NOTE: DATA statement used (Total process time):
      real time      3.90 seconds
      cpu time      0.96 seconds
```

Which of the following data sets would be a good candidate for compression?

**Q3.**

- A data set that has 10 long character variables, 2 numeric variables, and 100 observations.
- A data set that has 3 short character variables, 2 integer numeric variables, and 5,000,000 observations.
- A data set that has 3 character variables, 57 numeric variables, and 1,000,000 observations.
- All of the above.

Now that you have seen how to create a compressed data set, let's look at working with compressed data sets. In general, you use a compressed data set in your programs in the same way that you would use an uncompressed data set. However, there are two options that relate specifically to compressed data sets.

#### Accessing Observations Directly in a Compressed Data Set

By default, the DATA step processes observations in a SAS data set sequentially. However, sometimes you might want to access observations directly rather than sequentially because doing so can conserve resources such as CPU time, I/O, and real time. You can use the POINT= option in the MODIFY or SET statement to access observations directly rather than sequentially. You can review information about the POINT= option in the lesson **Creating Samples and Indexes**. You can also use the FSEDIT procedure to access observations directly.

Allowing direct access to observations in a compressed data set increases the CPU time that is required for creating or updating the data set. You can set an option that does not allow direct access for compressed data sets. If it is not important for you to be able to point directly to an observation by number within a compressed data set, it is a good idea to disallow direct access in order to improve the efficiency of creating and updating the data set. Let's look at how to disallow direct access to observations in a compressed data set.

#### The POINTOBS= Data Set Option

When you are working with compressed data sets, you use the **POINTOBS=** data set option to control whether observations can be processed with direct access (by observation number) rather than with sequential access only. This option is available beginning in SAS 8.

#### General form, **POINTOBS=** data set option:

**DATA SAS-data-set (POINTOBS= YES | NO);**

where

- **SAS-data-set** specifies the data set that you want to compress.
- **YES** is the default setting, which allows random access to the data set.
- **NO** does not allow random access to the data set.



In order to use the **POINTOBS=** data set option, the **COMPRESS=** option must have a value of **YES**, **CHAR**, or **BINARY** for the **SAS-data set** that is specified.

Allowing random access to a data set does not affect the efficiency of retrieving information from a data set, but it does increase the CPU usage by approximately 10% when you create or update a compressed data set. That is, allowing random access reduces the efficiency of writing to a compressed data set but does not affect the efficiency of reading from a compressed data set. Therefore, if you do not need to access data by observation number, then by specifying **POINTOBS=NO**, you can improve performance by approximately 10% when creating a compressed data set and when updating or adding observations to it.

#### Example

The following program creates a data set named **Company.Customer\_Compressed** from the **Company.Customer** data set and ensures that random access to the compressed data set is not allowed.

```
data company.customer_compressed (compress=yes pointobs=no);
  set company.customer;
run;
```

Now let's look at an option that enables you to further reduce the data storage space that is required for your compressed data sets.

#### The **REUSE=** System Option and the **REUSE=** Data Set Option

In a compressed data set, SAS appends new observations to the end of the data set by default. If you delete an observation within the data set, empty disk space remains in its place. However, it is possible to track and reuse free space within the data set when you delete or update observations. By reusing space within a data set, you can conserve data storage space.

The **REUSE= system option** and the **REUSE= data set option** specify whether or not SAS reuses space when observations are added to a compressed data set. If you set the **REUSE=** data set option to **YES** in a **DATA** statement, SAS tracks and reuses space in the compressed data set that is created in that **DATA** step. If you set the **REUSE= system option** to **YES**, SAS tracks and reuses free space in all compressed data sets that are created for the remainder of the current SAS session.

#### General form, **REUSE=** system option:

**OPTIONS REUSE= NO | YES;**

where

- **NO** is the default setting, which specifies that SAS does not track unused space in the compressed data set.
- **YES** specifies that SAS tracks free space and reuses it whenever observations are added to an existing compressed data set.

#### General form, REUSE= data set option:

```
DATA SAS-data-set (COMPRESS=YES REUSE= NO | YES);
```

where

- **SAS-data-set** specifies the data set that you want to compress.
- **NO** is the default setting, which specifies that SAS does not track unused space in the compressed data set.
- **YES** specifies that SAS tracks free space and reuses it whenever observations are added to an existing compressed data set.



The REUSE= data set option overrides the REUSE= system option.

If the REUSE= option is set to **YES**, observations that are added to the SAS data set are inserted wherever enough free space exists, instead of at the end of the SAS data set.

Specifying **NO** for the REUSE= option results in less efficient usage of space if you delete or update many observations in a SAS data set because there will be unused space within the data set. With the REUSE= option set to **NO**, the APPEND procedure, the FSEDIT procedure, and other procedures that add observations to the SAS data set add observations to the end of the data set, as they do for uncompressed data sets.

You cannot change the REUSE= attribute of a compressed data set after it is created. This means that space is tracked and reused in the compressed SAS data set according to the value of the REUSE= option that was specified when the SAS data set was created, not when you add and delete observations. Also, you should be aware that even with the REUSE= option set to **YES**, the APPEND procedure will add observations to the end of the data set.



Specifying **YES** as the value for the REUSE= option causes the POINTOBS= option to have a value of **NO** even if you specify **YES** as the value for POINTOBS=. The insertion of a new observation into unused space (rather than at the end of the data set) and the use of direct access are not compatible.

#### Example

The following program creates a compressed data set named **Company.Customer\_Compressed** from the **Company.Customer** data set. Because the REUSE= option is set to **YES**, SAS will track and reuse any empty space within the compressed data set.

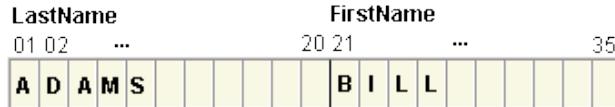
```
data company.customer_compressed (compress=yes reuse=yes);
  set company.customer;
run;
```

#### How SAS Compresses Data

Let's take a closer look at how SAS compresses data. A fictional data set named **Roster** is described in the table below.

| Variable  | Type      | Length |
|-----------|-----------|--------|
| LastName  | Character | 20     |
| FirstName | Character | 15     |

In uncompressed form, each observation in **Roster** uses a total of 35 bytes to store these two variables: 20 bytes for the first variable, LastName, and 15 bytes for the second variable, FirstName. The image below illustrates the storage of the first observation in the uncompressed version of **Roster**.



Suppose that you use the **CHAR** setting for the **COMPRESS=** option to compress **Roster**. In compressed form, the repeated blanks are removed from each value. The first observation from **Roster** uses a total of only 13 bytes: 7 for the first variable, LastName, and 6 for the second variable, FirstName. The image below illustrates the storage of the first observation in the compressed version of **Roster**.



The @ indicates the number of uncompressed characters that follow. The # indicates the number of blanks repeated at this point in the observation. Only a SAS engine can access these bytes. You cannot print or manipulate them.



Remember that in a compressed data set, observations might not all have the same length, because the length of an observation depends on the length of each value in the observation.



## Comparative Example: Creating and Reading Compressed Data Files

Suppose you want to create two SAS data sets from data that is stored in two raw data files. The raw data file that is referenced by the fileref **flat1** contains numeric data about customer orders for a retail company; you want to create a SAS data set named **Retail.Orders** from this raw data file. The raw data file that is referenced by the fileref **flat2** contains character data about customers for a retail company; you want to create a SAS data set named **Retail.Customers** from this raw data file.

In both cases, you can use the DATA step to create either an uncompressed data file or a compressed data file. Furthermore, you can use either binary or character compression in either case. That is, you can use the following techniques:

1. Numeric Data, No Compression
2. Numeric Data, **BINARY** Compression
3. Numeric Data, **CHAR** Compression
4. Character Data, No Compression
5. Character Data, **BINARY** Compression
6. Character Data, **CHAR** Compression.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the

structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for creating compressed data files.

## Programming Techniques

### 1. Numeric Data, No Compression

The following program creates the SAS data set **Retail.Orders**, which contains numeric data and is uncompressed. The second DATA step reads the uncompressed data file.

```
data retail.orders(compress=no);
infile flat1;
input Customer_ID      12.
      Employee_ID     12.
      Street_ID       12.
      Order_Date      date9.
      Delivery_Date   date9.
      Order_ID        12.
      Order_Type      commal6.
      Product_ID      12.
      Quantity        4.
      Total_Retail_Price dollar13.2
      CostPrice_Per_Unit dollar13.2
      Discount        5.2      ;
run;
data _null_;
  set retail.orders;
run;
```

### 2. Numeric Data, *BIN*ARY Compression

The following program creates the SAS data set **Retail.Orders\_binary**, which contains numeric data and uses *BIN*ARY compression. The second DATA step reads the compressed data file.

```
data retail.orders_binary(compress=binary);
infile flat1;
input Customer_ID      12.
      Employee_ID     12.
      Street_ID       12.
      Order_Date      date9.
      Delivery_Date   date9.
      Order_ID        12.
      Order_Type      commal6.
      Product_ID      12.
      Quantity        4.
      Total_Retail_Price dollar13.2
      CostPrice_Per_Unit dollar13.2
      Discount        5.2      ;
run;
data _null_;
  set retail.orders_binary;
run;
```

### 3. Numeric Data, *CHAR* Compression

The following program creates the SAS data set **Retail.Orders\_char**, which contains numeric data and uses *CHAR* compression. The second DATA step reads the compressed data file.

```
data retail.orders_char(compress=char);
infile flat1;
input Customer_ID      12.
```

```

Employee_ID      12.
Street_ID        12.
Order_Date       date9.
Delivery_Date    date9.
Order_ID         12.
Order_Type       commal6.
Product_ID       12.
Quantity         4.
Total_Retail_Price dollar13.2
CostPrice_Per_Unit dollar13.2
Discount         5.2      ;
run;
data _null_;
  set retail.orders_char;
run;

```

#### 4. Character Data, No Compression

The following program creates the SAS data set **Retail.Customers**, which contains character data and is uncompressed. The second DATA step reads the uncompressed data file.

```

data retail.customers(compress=no);
infile flat2;
input Customer_Country   $40.
  Customer_Gender   $1.
  Customer_Name     $40.
  Customer_FirstName $20.
  Customer_LastName $30.
  Customer_Age_Group $12.
  Customer_Type     $40.
  Customer_Group    $40.
  Customer_Address   $45.
  Street_Number     $8.      ;
run;
data _null_;
  set retail.cutomers;
run;

```

#### 5. Character Data, *BINARY* Compression

The following program creates the SAS data set **Retail.Customers\_binary**, which contains character data and uses *BINARY* compression. The second DATA step reads the compressed data file.

```

data retail.customers_binary(compress=binary);
infile flat2;
input Customer_Country   $40.
  Customer_Gender   $1.
  Customer_Name     $40.
  Customer_FirstName $20.
  Customer_LastName $30.
  Customer_Age_Group $12.
  Customer_Type     $40.
  Customer_Group    $40.
  Customer_Address   $45.
  Street_Number     $8.      ;
run;
data _null_;
  set retail.customers_binary;
run;

```

## 6. Character Data, CHAR Compression

The following program creates the SAS data set **Retail.Customers\_char**, which contains character data and uses *CHAR* compression. The second DATA step reads the compressed data file.

```
data retail.customers_char(compress=char);
  infile flat2;
  input Customer_Country  $40.
    Customer_Gender   $1.
    Customer_Name    $40.
    Customer_FirstName $20.
    Customer_LastName $30.
    Customer_Age_Group $12.
    Customer_Type    $40.
    Customer_Group   $40.
    Customer_Address $45.
    Street_Number    $8.      ;
run;
data _null_;
  set retail.customers_char;
run;
```

### General Recommendations

- Save data storage space by compressing data, but remember that compressed data causes an increase in CPU usage because the data must be uncompressed for processing. Compressing data always uses more CPU resources than not compressing data.
- Use binary compression only if the observation length is several hundred bytes or more.

In which of the following cases should you use COMPRESS=BINARY to compress your data file?

**Q.4.**

The variables in the data file are mostly numeric.

The observation length is 25.

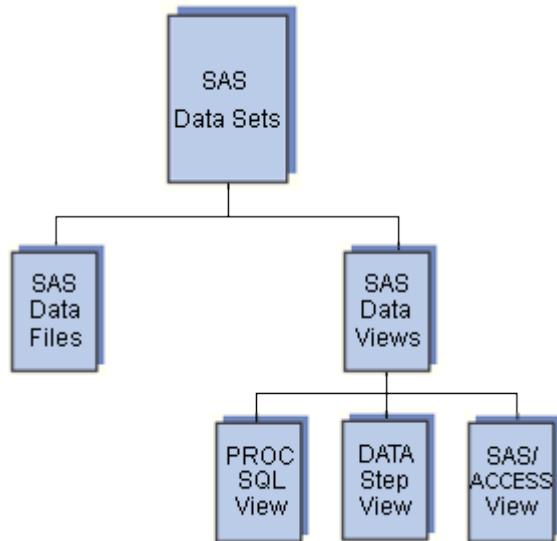
There are not many observations in the data file.

The variables in the data file are mostly character, with a length of 1.

## Using SAS DATA Step Views to Conserve Data Storage Space

Another way to save disk space is to leave your data in its original location and use a SAS data view to access it. Before looking at working with data views, let's look at what a SAS data view is and how it compares to a SAS data file.

A SAS data file and a SAS data view are both types of SAS data sets. The first type, a SAS data file, contains both descriptor information about the data and the data values. The second type, a SAS data view, contains only descriptor information about the data and instructions on how to retrieve data values that are stored elsewhere.



The main difference between SAS data files and SAS data views is where the data values are stored. A SAS data file contains the data values, and a SAS data view does not contain the values. Therefore, data views can be particularly useful if you are working with data values that change often.

Suppose you have a flat file that you read into a SAS data file. If the values in the flat file change, you need to update the data file in order to reflect those changes so that you access the correct values when you reference the data file. However, suppose you use a data view to access the values in your flat file instead of reading those values into a data file. You do not need to update the data view when the values in your flat file change, because each time you reference the view it will execute and access the most recent values in your flat file.

In most cases, you can use a SAS data view as if it were a SAS data file, although there are a few things to keep in mind when you are working with data views.



There are multiple types of SAS data views. This lesson discusses only DATA step views. To learn more about PROC SQL views, see the lesson [Creating and Managing Views Using PROC SQL](#). For more information about SAS data views and SAS data files, see the SAS documentation.

Now let's look at DATA step views.

## DATA Step Views

A DATA step view contains a partially compiled DATA step program that can read data from a variety of sources, including

- raw data files
- SAS data files
- PROC SQL views
- SAS/ACCESS views
- DB2, ORACLE, or other DBMS data.

A DATA step view can be created only in a DATA step. A DATA step view cannot contain global statements, host-specific data set options, or most host-specific FILE and INFILE statements. Also, a DATA step view cannot be indexed or compressed.

You can use DATA step views to

- always access the most current data in changing files
- avoid storing a copy of a large data file
- combine data from multiple sources.

The compiled code does not take up much room for storage, so you can create DATA step views to conserve disk space. On the other hand, use of DATA step views can increase CPU usage because SAS must execute the stored DATA step program each time you use the view.

To create a DATA step view, specify the VIEW= option after the final data set name in the DATA statement.

**General form, DATA step to create a DATA step view:**

```
DATA SAS-data-view <SAS-data-file-1 ... SAS data-file-n> /  
  VIEW=SAS-data-view;  
  <SAS statements>
```

**RUN;**

where

- SAS-data-view names the data view to be created
- SAS-data-file-1 ... SAS-data-file-n is an optional list that names any data files to be created
- SAS statements includes other DATA step syntax to create the data view and any data files that are listed in the DATA statement.

The VIEW= option tells SAS to compile, but not to execute, the source program and to store the compiled code in the input DATA step view that is named in the option.



If you specify additional data files in the DATA statement, SAS creates these data files when the view is processed in a subsequent DATA or PROC step. Therefore, you need to reference the data view before you attempt to reference the data file in later steps.

### Example

The following program creates a DATA step view named **Company.Newdata** that reads from the file referenced by the fileref in the INFILE statement.

```
data company.newdata / view=company.newdata;  
  infile <fileref>;  
  <DATA step statements>  
run;
```

### The DESCRIBE Statement

Beginning in SAS 8, DATA step views retain source statements. You can retrieve these statements by using the DESCRIBE statement. The following example uses the DESCRIBE statement in a DATA step to write a copy of the source code for the data view **Company.Newdata** to the SAS log:

```
data view=company.newdata;  
  describe;  
run;
```

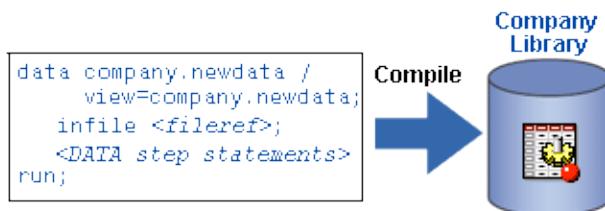
Now let's look at using DATA step views.

### Creating and Referencing a SAS DATA Step View

In order to use DATA step views successfully, you need to understand what happens when you create and reference one.

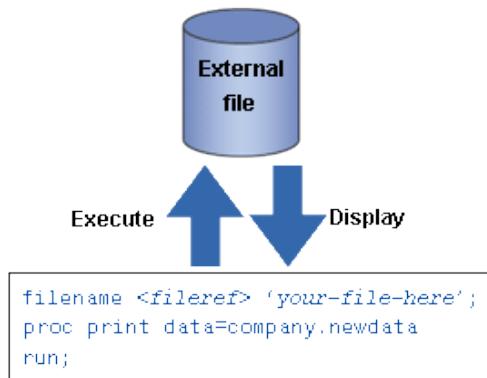
When you create a DATA step view,

- the DATA step is partially compiled
- the intermediate code is stored in the specified SAS data library with a member type of VIEW.



You reference a DATA step view in the same way that you reference a data file. When you reference the view in a subsequent DATA or PROC step,

- the compiler resolves the intermediate code and generates executable code for the host environment
- the generated code is executed as the DATA or PROC step requests observations.



You can use a DATA step view as you would use any other SAS data set, with the exception that you cannot write to the view except under very specific circumstances. Also, you should keep in mind that a SAS data view reads from its source files each time it is used, so if the data that it is accessing changes, the view will change also. Likewise, if the structure of the data that a view accesses changes, you will probably need to alter the view in order to account for this change.



In SAS 9.1, the OBSBUF= data set option enables you to specify how many observations to read at one time from the source data for the DATA step view. The default size of the view buffer is 32K, which means that the number of observations that can be read into the view buffer at one time depends on the observation length. If the observation length is larger than 32K, then only one observation can be read into the buffer at a time.

Which of the following statements about SAS DATA step views is true?

- Q.5.**
- A SAS DATA step view is a type of SAS data file.
  - You cannot reference a SAS DATA step view in a PROC step.
  - A SAS DATA step view always accesses the most recent data values when it is referenced.
  - You can read from or write to a SAS DATA step view in the same way that you would read from or write to any other SAS data set.

Remember that although data views conserve data storage space, processing them can require more resources than processing a data file. Let's look at a few situations where using a data view can adversely affect processing efficiency.

### Referencing a Data View Multiple Times in One Program

SAS executes a view each time it is referenced, even within one program. Therefore, if data is used many times in one program, it is more efficient to create and reference a temporary SAS data file than to create and reference a view.

#### Example

1. The following steps show the execution of the program below. This program references the SAS data view **Company.Newdata** three times in successive PROC steps.

```
proc print data=company.newdata;
run;
proc freq data=company.newdata;
run;
proc means data=company.newdata;
run;
```

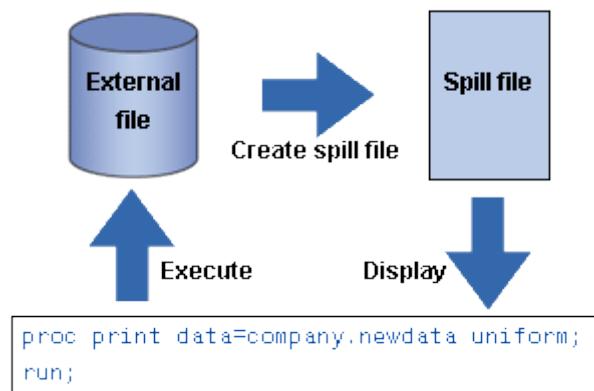
2. When the PROC PRINT step executes, the compiler resolves the intermediate code and generates executable code for **Newdata**.
3. The generated code is executed as the PROC PRINT step requests observations.
4. When the PROC FREQ step executes, the compiler executes the intermediate code and generates executable code for **Newdata**.
5. The generated code is executed as the PROC FREQ step requests observations.
6. When the PROC MEANS step executes, the compiler executes the intermediate code for a third time and generates executable code for **Newdata**.
7. The generated code is executed as the PROC MEANS step requests observations.

### Making Multiple Passes Through Data in a Data View

Expect a degradation in performance when you use a SAS data view with a procedure that requires multiple passes through the data. When multiple passes are requested, the view must build a cache, which is referred to as a spill file, that contains all generated observations. Then SAS reads the data in the spill file on each of the multiple passes through the data in order to ensure that subsequent passes read the same data that was read by previous passes.

For example, the UNIFORM option of the PRINT statement makes all the columns consistent from page to page by determining the longest value for a particular variable. In order to do this, SAS must make two passes through the data: one pass to find the longest value in the data, and one pass to print the data. If

if you use the **UNIFORM** option to print a data view, SAS creates a spill file as it generates observations from the view. Then SAS makes two passes through the observations that are in the spill file.



Some statistical procedures pass through the data more than once.

### Creating Data Views on Unstable Data

Avoid creating views on files whose structures often change. The view syntax describes the structure of the raw data; therefore, you need to make changes to the view each time the file changes.

For example, suppose you create a view that combines the data file **Company.Roster** with the data file **Company.Demog**. **Roster** contains the variables LastName and FirstName, and **Company.Demog** contains the variables LastName, Address and Age, as shown below.

| Company.Roster |           |        | Company.Demog |           |        |
|----------------|-----------|--------|---------------|-----------|--------|
| Variable       | Type      | Length | Variable      | Type      | Length |
| LastName       | Character | 20     | LastName      | Character | 20     |
| FirstName      | Character | 15     | Address       | Character | 45     |
|                |           |        | Age           | Numeric   | 3      |

Suppose that both **Company.Roster** and **Company.Demog** are sorted by LastName. You could use a MERGE statement to combine these two data files into a view named **Company.Roster\_View**, as shown below.

```
data company.roster_view/view=company.roster_view;
  merge company.roster company.demog;
  by lastname;
run;
```

Now suppose **Company.Roster** changes so that LastName is named Surname. Your data view must also be updated.

| Company.Roster | Company.Demog |
|----------------|---------------|
|----------------|---------------|

| Variable  | Type      | Length | Variable | Type      | Length |
|-----------|-----------|--------|----------|-----------|--------|
| Surname   | Character | 20     | LastName | Character | 20     |
| FirstName | Character | 15     | Address  | Character | 45     |
|           |           |        | Age      | Numeric   | 3      |

```
data roster_view/view=roster_view;
merge company.roster company.demog(rename=(LastName=Surname));
by lastname;
run;
```

If **Company.Roster** changed again so that Surname and FirstName were combined into one variable called FullName, the code for your data view would need additional changes. Although this is a simple example, you can see that a data view that is based on unstable data will require additional maintenance work.



## Comparative Example: Creating and Reading a SAS Data View

Suppose you have two SAS data sets, **Retail.Custview** and **Retail.Custdata**, that have been created from the same raw data file. **Retail.Custview** is a DATA step view, and **Retail.Custdata** is a data file. You can use these two data sets to compare the disk space that is required for each as well as the resources that are used to read from each:

1. Data View
2. Data File

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for using SAS DATA step views.

### Programming Techniques

#### 1. Data View

This program reads data from a raw data file and creates a SAS DATA step view named **Retail.Custview**, then reads from the new DATA step view. The first DATA step creates the data view **Retail.Custview**. The second DATA step reads from the DATA step view.

```
data retail.custview / view = retail.custview;
infile flat1;
input @1 Customer_ID      12.
      @13 Country        $2.
      @15 Gender          $1.
      @16 Personal_ID    $15.
      @31 Customer_Name   $40.
      @71 Customer_FirstName $20.
      @91 Customer_LastName $30.
      @121 Birth_Date     date9.
      @130 Customer_Address $45.
      @175 Street_ID      12.
      @199 Street_Number   $8.
      @207 Customer_Type_ID  8.;

run;
data _null_;
  set retail.custview;
```

```
run;
```

## 2. Data File

This program reads data from a raw data file and creates a SAS data file named **Retail.Custdata**, then reads from the new SAS data file. The first DATA step creates the data file **Retail.Custdata**. The second DATA step reads from the data file.

```
data retail.custdata;
infile flat1;
input @1 Customer_ID      12.
      @13 Country        $2.
      @15 Gender          $1.
      @16 Personal_ID    $15.
      @31 Customer_Name   $40.
      @71 Customer_FirstName $20.
      @91 Customer_LastName $30.
      @121 Birth_Date     date9.
      @130 Customer_Address $45.
      @175 Street_ID      12.
      @199 Street_Number   $8.
      @207 Customer_Type_ID  8.;

run;
data _null_;
  set retail.custdata;
run;
```

## General Recommendations

- Create a SAS DATA step view to avoid storing a raw data file and a copy of that data in a SAS data file.
- Use a SAS DATA step view if the content, but not the structure, of the flat file is dynamic.

Which of the following is an advantage of using a data view?

**Q.6.**

A data view can result in a savings of CPU time.

A data view saves storage space on disk.

A data view makes it easier to make multiple passes through data.

## Chapter Summary

### Reducing Data Storage Space for Character Variables

SAS stores character data as one character per byte. The default length for a character variable is 8 bytes. You can use the LENGTH statement to reduce the length of a character variable. You can also use other coding techniques to reduce the space that is needed for storing your character data.

### Reducing Data Storage Space for Numeric Variables

SAS stores numeric data in [floating-point representation](#). The default length for a numeric variable is 8 bytes. You can use a LENGTH statement to reduce the length of a numeric variable. Reading reduced-length numeric variables requires less I/O but more CPU resources than

reading full-length numeric variables. You should store only integer values in reduced-length numeric variables, and you should limit the values according to the length that you use. You can use PROC COMPARE to see the precision loss, if any, in the values of reduced-length numeric variables.

## **Compressing Data Files**

By default, a SAS data file is uncompressed. You can compress your data files in order to conserve disk space, although some files are not good candidates for compression. The file structure of a compressed data file is different from the structure of an uncompressed file. You use the COMPRESS= data set option or system option to compress a data file. You use the POINTOBS= data set option to enable SAS to access observations in compressed files directly rather than sequentially. You use the REUSE= data set option or system option to specify that SAS should reuse space in a compressed file when observations are deleted or updated.

## **Using SAS DATA Step Views to Conserve Data Storage Space**

You can leave your data in its original storage location and use SAS data views to access the data in order to reduce the amount of space needed for storing data on disk. A DATA step view is a specific type of data view that is created in a DATA step with the VIEW= option. You use the DESCRIBE statement to write the source code for a data view to the SAS log. Some of the advantages of using DATA step views rather than data files are that they always access the most recent data in dynamic files and that they require less disk space. However, there can be an effect on performance when you use a DATA step view.

## Chapter Quiz

Select the best answer for each question.

1. Which of the following statements about uncompressed SAS data files is true?
  - a. The descriptor portion is stored on whatever page has enough room for it.
  - b. New observations are always added in the first sufficient available space.
  - c. Deleted observation space is tracked.
  - d. New observations are always added at the end of the data set.
2. Which of the following statements about compressed SAS data files is true?
  - a. The descriptor portion is stored on whatever data set page has enough room for it.
  - b. Deleted observation spaced can be reused.
  - c. Compressed SAS data files have a smaller overhead than uncompressed SAS data files.
  - d. In a compressed SAS data set, each observation must be the same size.
3. Which of the following programs correctly creates reduced-length numeric variables?

|                                                                                                                                                                                                  |                                                                                                                                                                                                    |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>a.    <code>data temp;<br/>      infile file1;<br/>      input x 4.<br/>         y 3.<br/>         z 2.;<br/>      run;</code></p>                                                            | <p>c.    <code>data temp;<br/>      length x 4<br/>         y 3<br/>         z 2;<br/>      infile file1;<br/>      input x 4.<br/>         y 3.<br/>         z 2.;<br/>      run;</code></p>      |
| <p>b.    <code>data temp;<br/>      format x 4.<br/>         y 3.<br/>         z 2.;<br/>      infile file1;<br/>      input x 4.<br/>         y 3.<br/>         z 2.;<br/>      run;</code></p> | <p>d.    <code>data temp;<br/>      informat x 4.<br/>         y 3.<br/>         z 2.;<br/>      infile file1;<br/>      input x 4.<br/>         y 3.<br/>         z 2.;<br/>      run;</code></p> |
4. Which of the following statements about SAS data views is true?
  - a. SAS data views use less disk space but more CPU resources than SAS data files.
  - b. SAS data views can be created only in permanent SAS data libraries.
  - c. SAS data views use less CPU resources but more disk space than SAS data files.

- d.* SAS data views can be created only in temporary SAS data libraries.
5. Which of the following programs should you use to detect any loss of precision between the default-length numeric variables in **Company.Regular** and the reduced-length numeric variables in the data set **Company.Reduced**?
- a. `proc contents  
data=company.regular;  
compare  
data=company.reduced;  
run;`
- b. `proc compare  
base=company.regular  
compare=company.reduced;  
run;`
- c. `proc print data=company.regular;  
run;  
proc print data=company.reduced;  
run;`
- d. `proc datasets library=company;  
contents data=regular  
compare=reduced;  
run;`

# Chapter 10

## Utilizing Best Practices

### Introduction

This lesson demonstrates different ways of utilizing best SAS programming practices to optimize performance. As you compare the techniques described in this lesson, remember that differences in the use of resources are affected by which operating environment you work in and by the characteristics of your data.

This lesson is organized by topics that emphasize the following basic principles:

- Execute only necessary statements.
- Eliminate unnecessary passes of the data.
- Read and write only the data that you require.
- Store data in SAS data sets.
- Avoid unnecessary procedure invocation.

Each topic includes comparative examples that can improve the efficiency of your programs. Write programs to generate your own benchmarks, and adopt the programming techniques that produce the most savings for you.



This lesson does not cover the Scalable Performance Data Engine (SPD Engine), which is a SAS 9.1 technology for [threaded processing](#). For details about using the SPD Engine to improve performance, see the SAS documentation.

### Objectives

In this lesson, you learn to efficiently

- subset observations
- create new variables
- process and output data conditionally
- create multiple output data sets and sorted subsets
- modify variable attributes
- select observations from SAS data sets and external files
- subset variables
- read data from SAS data sets
- invoke SAS procedures.

## Executing Only Necessary Statements

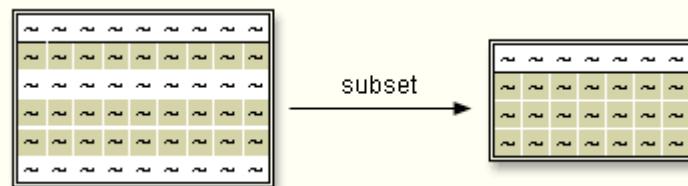
Best practices dictate that you should write programs that cause SAS to execute only necessary statements. When you execute the minimum number of statements in the most efficient order, you minimize the hardware resources that SAS uses. The resources that are affected include disk usage, memory usage, and CPU usage.

Techniques for executing only the statements you need include

- subsetting your data as soon as logically possible
- processing your data conditionally by using the most appropriate syntax for your data.

## Positioning of the Subsetting IF Statement

To subset your data based on a newly derived or computed variable, you must use the subsetting IF statement in a DATA step. As you output data to a SAS data set, you can subset it by processing only those observations that meet a specified condition.



The subsetting IF statement causes the DATA step to continue processing only those raw data records or observations that meet the condition of the expression specified in the IF statement. The resulting SAS data set or data sets contain a subset of the original external file or SAS data set.

Position the subsetting IF statement in the program so that it checks the subsetting condition as soon as it is logically possible, and so that unnecessary statements do not execute. When a statement is false, no further statements are processed for that observation.

Also, remember to subset data before performing calculations and to minimize the use of function calls or arithmetic operators. Unnecessary processing of unwanted observations results in higher expenditure of hardware resources.

### Comparative Example: Creating a Subset of Data

Suppose you want to create a subset of data, calculate six new variables, and conditionally output data by reading from the SAS data set **Retail.Order\_fact**. The data set should contain new variables for

- the month of the order
- the elapsed time between the order date and the delivery date
- the profit, based on the retail price, discount, and unit price
- total profit
- total discount
- total wait time.

The subset of data that includes only orders for the month of December is approximately 9.66% of the data.

You can accomplish this task by using a subsetting IF statement. Placement of this statement in the DATA step can affect the efficiency of the DATA step in terms of CPU time and real time. Notice the comparison between these two approaches:

1. A Subsetting IF Statement at the Bottom
2. A Subsetting IF Statement near the Top.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.

## Programming Techniques

### 1. A Subsetting IF Statement at the Bottom

This program calculates six new variables before the subsetting IF statement selects only observations whose values for Month are 12.

```
data profit;
retain TotalProfit TotalDiscount TotalWait Count 0;
set retail.order_fact;
MonthOfOrder=month(order_date);
WaitTime=sum(delivery_date,-order_date);
if discount gt . then
  CalcProfit=sum((total_retail_price*discount),-costprice_per_unit)
    *quantity;
else CalcProfit=sum(total_retail_price,-costprice_per_unit)
  *quantity;
TotalProfit=sum(totalprofit,calcprofit);
TotalDiscount=sum(totaldiscount,discount);
TotalWait=sum(totalwait,waittime);
Count+1;
if monthoforder=12;
run;
```

### 2. A Subsetting IF Statement near the Top

In this program, the subsetting IF statement is positioned immediately after the value for MonthofOrder has been calculated. If the value is not 12, then no further statements are processed for that observation. In this program, calculations are performed on a smaller number of observations, which results in greater program efficiency.

```
data profit;
retain TotalProfit TotalDiscount TotalWait Count 0;
set retail.order_fact;
MonthOfOrder=month(order_date);
if monthoforder=12;
WaitTime=sum(delivery_date,-order_date);
if discount gt . then
  CalcProfit=sum((total_retail_price*discount),-costprice_per_unit)
    *quantity;
else CalcProfit=sum(total_retail_price,-costprice_per_unit)
  *quantity;
TotalProfit=sum(totalprofit,calcprofit);
TotalDiscount=sum(totaldiscount,discount);
TotalWait=sum(totalwait,waittime);
Count+1;
run;
```

## General Recommendations

- Position the subsetting IF statement in a DATA step as soon as logically possible in order to save the most resources.

Why does the placement of a subsetting IF statement near the top of a program cause a decrease in CPU usage?

**Q1.**

CPU usage decreases because SAS does not have to read as much of the data into memory with a subsetting IF statement.

CPU usage decreases because SAS does not execute statements that are placed below the IF statement for the data that will not be kept.

CPU usage decreases because SAS does not move observations into the program data vector if they do not meet the criteria expressed in the subsetting IF statement.

CPU usage decreases because SAS does not write out as many observations.

Why is I/O the same regardless of whether you place a subsetting IF statement near the top or bottom of a program?

The same amount of data moves from the buffers in memory to the program data vector.

**Q2.**

The same amount of data moves between the program data vector and the output buffers in memory.

The same amount of processing occurs, regardless of the placement of the subsetting IF statement.

The placement of the subsetting IF statement does not affect the data that is read and written between the storage device and memory.

In the last comparative example you saw how a subsetting IF statement can be positioned in the DATA step so that no further statements are processed for that observation. Next, let's look at how different programming techniques can be used to

- create variables conditionally using DO groups
- create variables conditionally when calling functions.

Before viewing the sample code for these two comparative examples, let's review guidelines for using these techniques.

### Using Conditional Logic Efficiently

You can use conditional logic to change how SAS processes selected observations. Two techniques—IF-THEN/ELSE statements and SELECT statements—can be used interchangeably and perform comparably. Based on the characteristics of your data and depending on your environment, one of these techniques might give you better performance. Choose a technique that conserves your programming time and makes the program easiest to read.

| Technique              | Action                                                                   |
|------------------------|--------------------------------------------------------------------------|
| IF-THEN/ELSE statement | executes a SAS statement for observations that meet specific conditions. |
| SELECT statement       | executes one of several statements or groups of statements.              |



The number of conditions (values) tested and the type of variable tested affect CPU resources. For numeric variables, SELECT statements should always be slightly more efficient (use less

CPU time) than IF-THEN/ELSE statements. The performance gap between IF-THEN/ELSE and SELECT statements gradually widens as the number of conditions increases. For character variables, IF-THEN/ELSE statements are always more efficient than SELECT statements. The performance gap widens quickly between the two techniques as the number of conditions increases.

Use IF-THEN/ELSE statements when

- the data values are character values
- the data values are not uniformly distributed
- there are few conditions to check.

For best practices, follow these guidelines for writing efficient IF/THEN logic:

- For mutually exclusive conditions, use the ELSE IF statement rather than an IF statement for all conditions except the first.
- Check the most frequently occurring condition first, and continue checking conditions in descending order of frequency.
- When you execute multiple statements based on a condition, put the statements in a DO group.

Use SELECT statements when

- you have a long series of mutually exclusive numeric conditions
- data values are uniformly distributed.

Before writing conditional logic, determine the distribution of your data values. You can use the

- FREQ procedure to examine the distribution of the data values
- GCHART or GPLOT procedure to display the distribution graphically
- UNIVARIATE procedure to examine distribution statistics and to display the information graphically.

## Comparative Example: Creating Variables Conditionally Using DO Groups

Suppose you want to calculate an adjusted profit based on the values of the variable Order\_Type in the data set **Retail.Order\_fact**. For retail sales, which are represented by the value 1, the adjusted profit should be calculated as 105% of profit. For catalog sales, which are represented by the value 2, the adjusted profit should be calculated as 103% of profit. For internet sales, which are represented by the value 3, the adjusted profit should be equal to profit.

The following table shows that the values for the variable Order\_Type are not uniformly distributed.

| Order Type |           |         |
|------------|-----------|---------|
| Order_Type | Frequency | Percent |
| 1          | 3579850   | 75.23   |
| 2          | 635645    | 13.36   |
| 3          | 542850    | 11.41   |

The following table shows that the values for the variable Discount also are not uniformly distributed.

| Discount |           |         |
|----------|-----------|---------|
| Discount | Frequency | Percent |

| Discount   |           |         |
|------------|-----------|---------|
| Discount   | Frequency | Percent |
| .          | 4712585   | 99.04   |
| <b>30%</b> | 19740     | 0.41    |
| <b>40%</b> | 15170     | 0.32    |
| <b>50%</b> | 9805      | 0.21    |
| <b>60%</b> | 1045      | 0.02    |

Techniques for creating new variables conditionally include

1. IF-THEN/ELSE statements
2. SELECT statements.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.

## Programming Techniques

### 1 IF-THEN/ELSE Statements

This program uses IF-THEN/ELSE statements with DO groups to conditionally execute multiple statements that calculate an adjusted profit. Conditions are checked in descending order of frequency.

```

data retail.order_info_1;
set retail.order_fact;
if order_type=1 then
  do;          /* Retail Sale */
    Float=delivery_date-order_date;
    RevenueQuarter=qtr(order_date);
    AveragePrice=total_retail_price/quantity;
    if discount=. then NetPrice=total_retail_price;
    else NetPrice=total_retail_price-discount;
    Profit=netPrice-(quantity*costprice_per_unit)*1.05;
  end;
else if order_type=2 then
  do;          /* Catalog Sale */
    Float=delivery_date-order_date;
    RevenueQuarter=qtr(order_date);
    AveragePrice=total_retail_price/quantity;
    if discount=. then NetPrice=total_retail_price;
    else NetPrice=total_retail_price-discount;
    Profit=netprice-(quantity*costprice_per_unit)*1.03;
  end;
else
  do;          /* Internet Sale */
    Float=delivery_date-order_Date;
    RevenueQuarter=qtr(order_date);
    AveragePrice=total_retail_price/quantity;
    if discount=. then NetPrice=total_retail_price;
    else NetPrice=total_retail_price-discount;
    Profit=netprice-(quantity*costprice_per_unit);
  end;
run;

```

## 2. SELECT Statements

This program uses SELECT/WHEN statements with DO groups to conditionally execute multiple statements that calculate an adjusted profit. Conditions are checked in descending order of frequency.

```
data retail.order_info_2;
set retail.order_fact;
select(order_type);
when (1)
do; /* Retail Sale */
  Float=delivery_date-order_date;
  RevenueQuarter=qtr(order_date);
  AveragePrice=total_retail_price/quantity;
  if discount=. then NetPrice=total_retail_price;
  else NetPrice=total_retail_price-discount;
  Profit=netprice-(quantity*costprice_per_unit)*1.05;
end;
when (2)
do; /* Catalog Sale */
  Float=delivery_date-order_date;
  RevenueQuarter=qtr(order_date);
  AveragePrice=total_retail_price/quantity;
  if discount=. then NetPrice=total_retail_price;
  else NetPrice=total_retail_price-discount;
  Profit=netprice-(quantity*costprice_per_unit)*1.03;
end;
otherwise
do; /* Internet Sale */
  Float=delivery_date-order_date;
  RevenueQuarter=qtr(order_date);
  AveragePrice=total_retail_price/quantity;
  if discount=. then NetPrice=total_retail_price;
  else NetPrice=total_retail_price-discount;
  Profit=netprice-(quantity*costprice_per_unit);
end;
end;
run;
```

### General Recommendations

- Check the most frequently occurring condition first, and continue checking conditions in descending order of frequency, regardless of whether you use IF-THEN/ELSE or SELECT statements.
- When you execute multiple statements based on a condition, put the statements in a DO group.

When you write a program that creates a new variable that is based on a large number of numeric values that are evenly distributed, it is best practice to

use SELECT/WHEN statements that check for conditions in descending order of frequency.

## Q.3.

use SELECT/WHEN statements that check for conditions in ascending order of frequency.

use IF-THEN/ELSE statements that check for conditions in ascending order of frequency.

use IF-THEN/ELSE statements that check for conditions in descending order of frequency.



## Comparative Example: Creating Variables Conditionally When Calling Functions

Suppose you want to create a report that includes a new variable that is based on the value of an existing variable in the SAS data set **Retail.Order\_fact**. Values for the new Month variable are extracted from the existing variable Order\_Date by using the MONTH function.

The following table shows that the values for Month are fairly evenly distributed.

| Month of Order |           |         |
|----------------|-----------|---------|
| Month          | Frequency | Percent |
| Jan            | 386535    | 8.12    |
| Feb            | 319895    | 6.72    |
| Mar            | 350255    | 7.36    |
| Apr            | 421265    | 8.85    |
| May            | 443535    | 9.32    |
| Jun            | 429760    | 9.03    |
| Jul            | 438085    | 9.21    |
| Aug            | 443075    | 9.31    |
| Sep            | 299260    | 6.29    |
| Oct            | 373400    | 7.85    |
| Nov            | 393750    | 8.27    |
| Dec            | 459530    | 9.66    |

Techniques for creating new variables conditionally include

1. Parallel IF Statements
2. ELSE IF Statements, Many Function References
3. ELSE IF Statements, One Function Reference
4. SELECT Group.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.

### Programming Techniques

#### 1. Parallel IF Statements

This program calls the MONTH function 12 times. With these non-exclusive cases, each IF statement executes for each observation that is read from **Retail.Order\_Fact**. This is the least efficient approach.

```
data retail.orders;
set retail.order_fact;
if month(order_date)=1 then Month='Jan';
if month(order_date)=2 then Month='Feb';
if month(order_date)=3 then Month='Mar';
if month(order_date)=6 then Month='Jun';
if month(order_date)=7 then Month='Jul';
if month(order_date)=8 then Month='Aug';
if month(order_date)=9 then Month='Sep';
if month(order_date)=10 then Month='Oct';
```

```
if month(order_date)=11 then Month='Nov';
if month(order_date)=12 then Month='Dec';
run;
```

## 2. ELSE IF Statements, Many Function References

This program uses ELSE IF statements that call the function MONTH. Once the true condition is found, subsequent ELSE IF statements are not executed. This is more efficient than using parallel IF statements, but the MONTH function is executed many times.

```
data retail.orders;
set retail.order_fact;
if month(order_date)=1 then Month='Jan';
else if month(order_date)=2 then Month='Feb';
else if month(order_date)=3 then Month='Mar';
else if month(order_date)=4 then Month='Apr';
else if month(order_date)=5 then Month='May';
else if month(order_date)=6 then Month='Jun';
else if month(order_date)=7 then Month='Jul';
else if month(order_date)=10 then Month='Oct';
else if month(order_date)=11 then Month='Nov';
else if month(order_date)=12 then Month='Dec';
run;
```

## 3. ELSE IF Statements, One Function Reference

This program uses the MONTH function to find the value of Order\_Date, but only once. The MONTH function is called immediately after reading the data set and before any IF-THEN/ELSE statements execute. This is efficient.

```
data retail.orders(drop=mon);
set retail.order_fact;
mon=month(order_date);
if mon=1 then Month='Jan';
else if mon=2 then Month='Feb';
else if mon=3 then Month='Mar';
else if mon=4 then Month='Apr';
else if mon=5 then Month='May';
else if mon=6 then Month='Jun';
else if mon=7 then Month='Jul';
else if mon=8 then Month='Aug';
else if mon=9 then Month='Sep';
else if mon=10 then Month='Oct';
else if mon=11 then Month='Nov';
else if mon=12 then Month='Dec';
run;
```

## 4. SELECT Group

In this program, the SELECT statement calls the MONTH function only once, before WHEN statements execute and assign values for Month. This is efficient.

```
data retail.orders;
set retail.order_fact;
select(month(order_date));
when (1) Month='Jan';
when (2) Month='Feb';
when (3) Month='Mar';
when (4) Month='Apr';
when (5) Month='May';
```

```

when (6) Month='Jun';
when (7) Month='Jul';
when (8) Month='Aug';
when (11) Month='Nov';
when (12) Month='Dec';
otherwise;
end;
run;

```

### General Recommendations

- Avoid using parallel IF statements, which use the most resources and are the least efficient way to conditionally execute statements.
- Use IF-THEN/ELSE statements and SELECT blocks to be more efficient.
- To significantly reduce the amount of resources used, write programs that call a function only once instead of repetitively using the same function in many statements. SAS functions are convenient, but they can be expensive in terms of CPU resources.

Which of the following is true about using conditional processing?

**Q.4.**

You gain the largest efficiencies by using parallel IF statements.

To improve efficiency, minimize the function references in statements that execute conditionally.

Write conditional logic for SELECT statements that check conditions in ascending order of frequency.

SELECT statements that use function references are always less efficient.

You've seen different ways of efficiently creating variables conditionally, which reinforce the principle of executing only necessary statements. The next comparative example in this topic addresses techniques that can be used to create data in DO groups. Before viewing this example, you might want to review the following information about which statements are needed in DO groups.

### Using DO Groups Efficiently

You can conditionally execute only necessary statements by placing them in DO groups that are associated with IF-THEN/ELSE statements or with SELECT/WHEN statements. Groups of statements execute only when a particular condition is true. Remember to use the following criteria when choosing which technique is more efficient:

|                                         | IF-THEN/ELSE Statements | SELECT/WHEN Statements |
|-----------------------------------------|-------------------------|------------------------|
| The number of conditions                | few                     | many                   |
| The distribution of a variable's values | not uniform             | uniform                |
| The type of variable                    | character               | numeric                |

When using a DO group with IF-THEN/ELSE statements, add DO after the THEN clause, and add an END statement after all of the statements that you want executed as a group.

```

data orders;
set company.orders;

```

```

if order_type = 1 then
do;
<multiple executable statements here>
end;
else if order_type = 2 then
do;
<multiple executable statements here>
end;
else if order_type = 3 then
do;
<multiple executable statements here>
end;
run;

```



Use an IF-THEN DO group when you create multiple variables based on a condition.

When using a DO group with SELECT/WHEN statements, add DO after the WHEN condition, and add an END statement after all of the statements that you want executed as a group. Use an OTHERWISE statement to specify the statements that you want executed if no WHEN condition is met.

```

data orders;
set company.orders;
select (order_type);
when (1)
do;
<multiple executable statements here>
end;
when (2)
do;
<multiple executable statements here>
end;
when (3)
do;
<multiple executable statements here>
end;
otherwise;
end;
run;

```

Remember that IF-THEN/ELSE and SELECT/WHEN logic require that there be no intervening statements between the IF and the ELSE conditions or between the SELECT and the WHEN conditions.



### Comparative Example: Creating Data in DO Groups

Suppose you want to identify which customer groups are Club Members, Club Gold Members, or Internet/Catalog members, based on data from the data set **Retail.Customer\_hybrid**. You also want to identify the nature of customer activity as "inactive", "low activity", "medium activity", or "high activity".

The following table shows the distribution of values for Customer\_Type\_ID.

| Customer Type ID |           |         |
|------------------|-----------|---------|
| Customer_Type_ID | Frequency | Percent |
| 1010             | 568446    | 12.39   |
| 1020             | 579156    | 12.62   |

| Customer Type ID |           |         |  |
|------------------|-----------|---------|--|
| Customer_Type_ID | Frequency | Percent |  |
| 1030             | 571608    | 12.46   |  |
| 1040             | 574209    | 12.52   |  |
| 2010             | 566457    | 12.35   |  |
| 2020             | 571914    | 12.47   |  |
| 2030             | 579054    | 12.62   |  |
| 3010             | 576810    | 12.57   |  |

Techniques for creating new variables based on the values of specific variables include

1. SELECT, IF/SELECT Statements
2. Nested SELECT Statements
3. Serial IF Statements
4. IF-THEN/ELSE IF Statements with a Link.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for executing only necessary statements.

## Programming Techniques

### 1. SELECT, IF/SELECT Statements

This program creates a permanent SAS data set named **Retail.Customers** by reading the **Retail.Customer\_hybrid** data set. SELECT/WHEN logic and SELECT/WHEN statements in an IF/THEN DO group populate the variables Customer\_Group and Customer\_Activity. If the value of the first two digits of Customer\_Type\_ID is 10, 20, or 30, then Customer\_Group is populated. If the value of the first two digits of Customer\_Type\_ID is 10 or 20, then Customer\_Activity is populated by reading the last two digits of Customer\_Type\_ID.

```
data retail.customers;
length Customer_Group $ 26 Customer_Activity $ 15;
set retail.customer_hybrid;
  select(substr(put(customer_type_ID,4.),1,2));
    when ('10') customer_group='Orion Club members';
    when ('20') customer_group='Orion Club Gold members';
    when ('30') customer_group='Internet/Catalog Customers';
    otherwise;
  end;
if substr(put(customer_type_ID,4.),1,2) in ('10', '20') then
  do;
    select(substr(put(customer_type_ID,4.),3,2));
      when ('10') customer_activity='inactive';
      when ('20') customer_activity='low activity';
      when ('30') customer_activity='medium activity';
      when ('40') customer_activity='high activity';
      otherwise;
    end;
  end;
run;
```

### 2. Nested SELECT Statements

This program creates a permanent SAS data set named **Retail.Customers** by reading the **Retail.Customer\_hybrid** data set. Nested SELECT statements are used to populate the variables Customer\_Group and Customer\_Activity.

```
data retail.customers;
length Customer_Group $ 26 Customer_Activity $ 15;
set retail.customer_hybrid;
select(substr(put(customer_type_ID,4.),1,2));
when ('10')
do;
customer_group='Orion Club members';
select(substr(put(customer_type_ID,4.),3,2));
when ('10') customer_activity='inactive';
when ('20') customer_activity='low activity';
when ('30') customer_activity='medium activity';
when ('40') customer_activity='high activity';
otherwise;
end;
end;
when ('20')
do;
customer_group='Orion Club Gold members';
select(substr(put(customer_type_ID,4.),3,2));
when ('10') customer_activity='inactive';
when ('20') customer_activity='low activity';
when ('30') customer_activity='medium activity';
when ('40') customer_activity='high activity';
otherwise;
end;
end;
when ('30') customer_group='Internet/Catalog Customers';
otherwise;
end;
run;
```

### 3. Serial IF Statements

This program creates a permanent SAS data set named **Retail.Customers** by reading the **Retail.Customer\_hybrid** data set. Serial IF statements are used to populate the variables Customer\_Group and Customer\_Activity.

```
data retail.customers;
length Customer_Group $ 26 Customer_Activity $ 15;
set retail.customer_hybrid;
if substr(put(customer_type_ID,4.),1,2)='10' then
customer_group='Orion Club members';
if substr(put(customer_type_ID,4.),1,2)='20' then
customer_group='Orion Club Gold members';
if substr(put(customer_type_ID,4.),1,2)='30' then
customer_group='Internet/Catalog Customers';
if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
substr(put(customer_type_ID,4.),3,2)='10' then
customer_activity='inactive';
if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
substr(put(customer_type_ID,4.),3,2)='20' then
customer_activity='low activity';
if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
substr(put(customer_type_ID,4.),3,2)='30' then
customer_activity='medium activity';
```

```

if substr(put(customer_type_ID,4.),1,2) in ('10', '20') and
substr(put(customer_type_ID,4.),3,2)='40' then
  customer_activity='high activity';
run;

```

#### 4. IF-THEN/ELSE IF Statements with a Link

This program creates a permanent SAS data set named **Retail.Customers** by reading the **Retail.Customer\_hybrid** data set. IF-THEN/ELSE IF statements are used with a link to populate the variables Customer\_Group and Customer\_Activity.

```

data retail.customers;
length Customer_Group $ 26 Customer_Activity $ 15;
set retail.customer_hybrid;
if substr(put(customer_type_ID,4.),1,2)='10' then
  do;
    customer_group='Orion Club members';
    link activity;
  end;
else if substr(put(customer_type_ID,4.),1,2)='20' then
  do;
    customer_group='Orion Club Gold members';
    link activity;
  end;
else if substr(put(customer_type_ID,4.),1,2)='30' then
  customer_group='Internet/Catalog Customers';
return;
activity:
if substr(put(customer_type_ID,4.),3,2)='10' then
  customer_activity='inactive';
else if substr(put(customer_type_ID,4.),3,2)='20' then
  customer_activity='low activity';
else if substr(put(customer_type_ID,4.),3,2)='30' then
  customer_activity='medium activity';
else if substr(put(customer_type_ID,4.),3,2)='40' then
  customer_activity='high activity';
return;
run;

```

#### General Recommendations

- Avoid serial IF statements because they use extra resources.

If you need to create two variables, x and y, based on the value of the variable z, which code is most efficient?

**Q5.**

```

if z=5 then x=2 and y=7;

if z=5 then x=2;
if z=5 then y=7;
if z=5 then x=2;
  else if z=5 then y=7;
if z=5 then
  do;
    x=2;
    y=7;
  end;

```

## **Eliminating Unnecessary Passes through the Data**

Best practices dictate that you should eliminate unnecessary passes through the data. To minimize I/O operations and CPU time, avoid reading or writing data more than necessary. Accomplish this by taking advantage of one-step processing, which can lead to efficiencies.

### **Using a Single DATA or PROC Step to Enhance Efficiency**

Whenever possible, use a single DATA or PROC step to enhance efficiency. Techniques that minimize passes through the data include

- using a single DATA step to create multiple output data sets
- using the SORT procedure with a WHERE statement to create sorted subsets
- using the DATASETS procedure to modify variable attributes.

Before viewing comparative examples that address these techniques, it might be helpful to review the following information about these practices.

#### **Using a Single DATA Step to Create Multiple Output Data Sets**

It is good programming practice to take advantage of the DATA step's ability to create multiple output data sets at the same time. This is more efficient than using a series of individual DATA steps. Using a single DATA step saves resources because input data is read only once.

The following program demonstrates how to use a single DATA step to read data once and create three subsets of data.

```
data Sales_managers Account_managers Finance_managers;
  set company.organization;
  if job_title='Sales Manager' then
    output Sales_managers;
  else if job_title='Account Manager' then
    output Account_managers;
  else if job_title='Finance Manager' then
    output Finance_managers;
run;
```

#### **Using the SORT Procedure with a WHERE Statement to Create Sorted Subsets**

It is good programming practice to take advantage of the SORT procedure's ability to sort and subset in the same PROC step. This is more efficient than using two separate steps to accomplish this—a DATA step to subset followed by a procedure step that sorts.

The following program demonstrates how you can sort data and select only those observations that meet the conditions of the WHERE statement—in one step. Processing a data set only once saves I/O and CPU resources.

```
proc sort data=company.organization
  out=company.managers;
  by job_title;
  where job_title in('Sales Manager',
    'Account Manager',
    'Finance Manager');
run;
```

## Using the DATASETS Procedure to Modify Variable Attributes

Use PROC DATASETS instead of a DATA step to modify data attributes. The DATASETS procedure uses fewer resources than the DATA step because it processes only the descriptor portion of the data set, not the data portion. PROC DATASETS retains the sort flag, as well as indexes.

```
proc datasets lib=company;
  modify organization;
  rename Job_title=Title;
quit;
```



You cannot use the DATASETS procedure to modify the type, length, or position of variables because these attributes directly affect the data portion of the data set. To perform these operations, use the DATA step.



## Comparative Example: Creating Multiple Subsets of a SAS Data Set

Suppose you want to create five subsets of data from the data set **Retail.Customer**. You need a subset for each of five countries. Techniques for creating multiple subsets include writing

1. Multiple DATA Steps
2. A Single DATA Step.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for eliminating unnecessary passes through the data.

### Programming Techniques

#### 1. Multiple DATA Steps

This program includes multiple DATA steps and subsequently reads data five times from the same **Retail.Customer** data set. Individual subsetting IF statements appear in five separate DATA steps.

```
data retail.UnitedStates;
  set retail.customer;
  if country='US';
run;
data retail.France;
  set retail.customer;
  if country='FR';
run;
data retail.Italy;
  set retail.customer;
  if country='IT';
run;
data retail.Germany;
  set retail.customer;
  if country='DE';
run;
data retail.Spain;
  set retail.customer;
  if country='ES';
```

```
run;
```

## 2. A Single DATA Step

This program uses only one DATA step to create five output data sets. The data set **Retail.Customer** is read only once. Also, IF-THEN/ELSE statements are used to conditionally output data to specific data sets.

```
data retail.UnitedStates  
      retail.France  
      retail.Italy  
      retail.Germany  
      retail.Spain;  
set retail.customer;  
if country='US' then output retail.UnitedStates;  
else if country='FR' then output retail.France;  
else if country='IT' then output retail.Italy;  
else if country='DE' then output retail.Germany;  
else if country='ES' then output retail.Spain;  
run;
```

### General Recommendations

- When creating multiple subsets from a SAS data set, use a single DATA step with IF-THEN/ELSE IF logic to output to appropriate data sets.

Why is there a savings in I/O and CPU usage when you create multiple subsets with one DATA step?

**Q.6.**

SAS writes the data only once when you create multiple subsets with one DATA step.

SAS reads the data only once when you create multiple subsets with one DATA step.

SAS reads and writes the data only once when you create multiple subsets with one DATA step.

SAS reads only the data that is required for each subset with one DATA step.



### Comparative Example: Creating a Sorted Subset of a SAS Data Set

Suppose you want to create a sorted subset of a SAS data set named **Retail.Customer**. You want only data for customers in the United States, France, Italy, Germany, and Spain.

Techniques for creating sorted subsets of SAS data sets include

1. A DATA Step and PROC SORT
2. PROC SORT with a WHERE Statement.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for eliminating unnecessary passes through the data.

### Programming Techniques

## 1. A DATA Step and PROC SORT

This program has two steps. The first step creates a SAS data set by subsetting observations based on the value of the variable Country. The second step sorts the data according to the values for each country. Passing through all the data once and the subset twice can increase I/O and CPU operations.

```
data retail.CountrySubset;
  set retail.customer;
  where country in('US','FR','IT','DE','ES');
run;

proc sort data=retail.CountrySubset;
  by country;
run;
```

## 2. PROC SORT with a WHERE Statement

In one step, this program sorts data and selects only those observations that meet the conditions of the WHERE statement. Processing only one data set once saves CPU and I/O resources.

Note that if this program did not create a second data set named **Retail.CountrySubset**, it would write over the data set named **Retail.Customer** with only part of the data.

```
proc sort data=retail.customer out=retail.CountrySubset;
  by country;
  where country in('US','FR','IT','DE','ES');
run;
```

## General Recommendations

- When you need to process a subset of data with a procedure, use a WHERE statement in the procedure instead of creating a subset of data and reading that data with the procedure.
- Write one program step that both sorts and subsets. This approach can take less programmer time and debugging time than writing separate program steps that subset and sort.

Why is this program not efficient?

```
data customer;
  set retail.customer;
run;
proc print data=customer;
run;
```

Q.7.

It is not efficient to read a permanent data set and create a temporary one. SAS reads temporary data sets faster than permanent ones.

It is not efficient to create a temporary data set by reading a permanent one. SAS creates permanent data sets faster than temporary ones.

This program copies a SAS data file from a permanent library to the temporary WORK library, which incurs unnecessary CPU and I/O costs.

PROC PRINT reads permanent data sets more efficiently than temporary data sets.

How could you make this program more efficient?

```
data females;
  set retail.customer;
  where customer_gender='F';
run;
proc means data=females;
  var customer_age;
```

Q.8.

```
run;
```

Replace the WHERE statement in the DATA step with a subsetting IF statement.

Omit the DATA step and use a WHERE statement to subset data in a PROC MEANS step.

Omit the DATA step and use a subsetting IF statement to subset data in a PROC MEANS step.

Create a new permanent data set in the DATA step.



## Comparative Example: Changing the Variable Attributes of a SAS Data Set

Suppose you want to change the variable attributes in **Retail.NewCustomer** to make them consistent with those in the **Retail.Customer** data set. The data set **Retail.NewCustomer** contains 89954 observations and 12 variables.

The following table shows the variable names and formats in each SAS data set.

| SAS Data Set       | Variable Name            | Variable Format         |
|--------------------|--------------------------|-------------------------|
| Retail.Customer    | Country<br>Birth_Date    | \$COUNTRY.<br>DATE9.    |
| Retail.NewCustomer | Country_ID<br>Birth_Date | \$COUNTRY.<br>MMDDYYP10 |

Techniques for changing the variable attributes of a SAS data set include

1. A DATA Step
2. PROC DATASETS.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for eliminating unnecessary passes through the data.

### Programming Techniques

#### 1. A DATA Step

This program uses a DATA step with a RENAME statement and a FORMAT statement to modify attributes for the variables Country\_ID and Birth\_Date.

```
data retail.newcustomer;
  set retail.newcustomer;
  rename Country_ID=country;
  format birth_date date9. ;
run;
```

#### 2. PROC DATASETS

This program uses PROC DATASETS to modify the names and formats of the variables Country\_ID and Birth\_Date.

```
proc datasets lib=retail;
  modify newcustomer;
  rename Country_ID=country;
```

```
format birth_date date9.;  
quit;
```

## General Recommendations

- To save significant resources, use the DATASETS procedure instead of a DATA step to change the attributes of a SAS data set.

Why does PROC DATASETS use fewer resources to change variable attributes than a DATA step?

**Q.9.**

PROC DATASETS does not process the data portion of the data set.

PROC DATASETS uses RUN-group processing.

PROC DATASETS reads only the data for the variables that need new attributes.

Procedures are always more efficient than the DATA step.

## Reading and Writing Only Essential Data

Best practices dictate that you should write programs that read and write only essential data. If you process fewer observations and variables, you conserve resources. This topic covers many different techniques that can improve performance when you

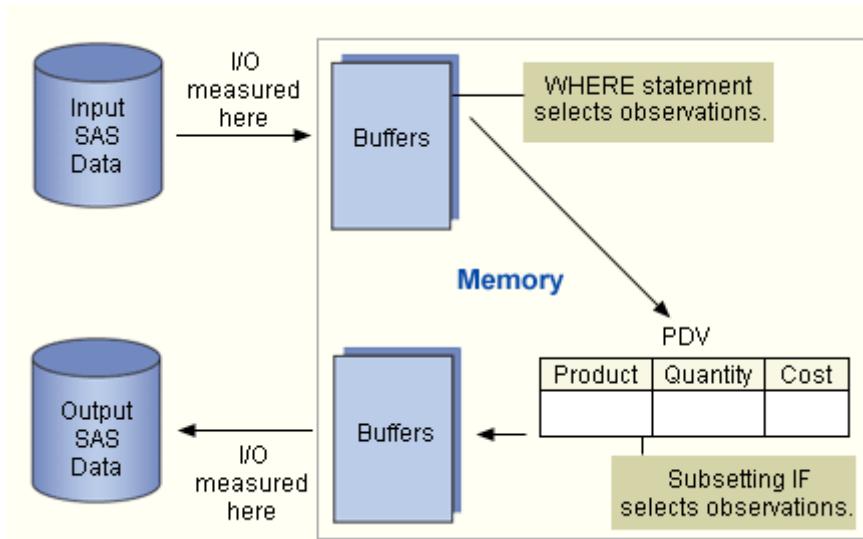
- select observations from SAS data sets
- select observations from external files
- keep or drop variables when creating or reporting on SAS data sets.

Let's begin by looking at how two useful statements differ in how they select observations to subset.

### Selecting Observations Using Subsetting IF versus WHERE Statements

You can use WHERE statements or subsetting IF statements to subset data. Although both statements test a condition to determine whether SAS should process an observation, the WHERE statement is more efficient.

The following graphic illustrates differences in how these statements process data.



I/O operations are measured as data moves between the disk that contains input SAS data and the buffer in memory, and when data moves from the output buffer to the disk that contains output data sets. Input data is not affected by the WHERE statement or subsetting IF statement. However, output data is affected by both.

CPU time is measured when data must be processed in the program data vector. CPU time can be saved if fewer observations are processed.

A WHERE statement and an IF statement make different use of the program data vector. The WHERE statement examines what is in the input page buffer and selects observations before they are loaded in the program data vector, which results in a savings in CPU operations. The subsetting IF statement loads all observations sequentially into the program data vector. If the statement finds a match and the statement is true, then the data is processed and is written to the output page buffer.

WHERE statements work on variables that are already in SAS data sets. IF statements can work on any variable in the program data vector, including new or old variables.



### Comparative Example: Creating a Subset of a SAS Data Set

Suppose you want to create a subset of the data set **Retail.Customer**. You want to include data for only the United Kingdom. The subset contains approximately 5.56% of the **Retail.Customer** data.

Techniques for subsetting observations include

1. Subsetting IF Statement
2. WHERE Statement.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

## Programming Techniques

## 1. Subsetting IF Statement

This program uses the IF statement to select observations if the value for Country is *GB*.

```
data retail.UnitedKingdom;
set retail.customer;
if country='GB';
run;
```

## 2. WHERE Statement

This program uses the WHERE statement to select observations when the value for Country is *GB*. This is more efficient than using a subsetting IF statement.

```
data retail.UnitedKingdom;
set retail.customer;
where country='GB';
run;
```

### General Recommendations

- To save CPU resources, use a WHERE statement instead of a subsetting IF statement to subset a SAS data set.

### Other Differences between the IF and WHERE Statements

Review the following table to note other differences between the IF and WHERE statements.

|                                    | The IF Statement                                                                                                                                                                             | The WHERE Statement                                                                              |
|------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| Selecting Data                     | can select records from external files, observations from SAS data sets, observations created with an INPUT statement, or observations based on the value of a computed or derived variable. | can only select observations from SAS data sets.                                                 |
| Conditional Execution              | can be executed conditionally as part of an IF statement.                                                                                                                                    | cannot be executed conditionally as part of an IF statement.                                     |
| Grouping Data Using a BY Statement | produces the same data set when a BY statement accompanies a SET, MERGE, or UPDATE statement.                                                                                                | produces a different data set when a BY statement accompanies a SET, MERGE, or UPDATE statement. |
| Merging Data                       | selects observations after combining current observations.                                                                                                                                   | applies the selection criteria to each input data set before combining observations.             |



If you use the WHERE= data set option and the WHERE statement in the same DATA step, SAS ignores the WHERE statement for input data sets. The WHERE= data set option and the WHERE statement call the same SAS routine.

Can you use the WHERE statement to refer to new variables?

Q.10.

You cannot use the WHERE statement to refer to new variables, because SAS evaluates the WHERE statement in the page buffer.

You can use the WHERE statement to refer to new variables as long as the new variables are created from variables in the original data set.

**Q11.**

You cannot use the WHERE statement to refer to new variables because the WHERE statement cannot use functions.

You can use the WHERE statement to refer to new variables in a DATA step, but you cannot use a WHERE statement in a procedure.

Which subsetting technique can you use to read from an external file?

You can use either the subsetting IF statement or the WHERE statement.

You can use only the WHERE statement.

You can use only the subsetting IF statement.

You can use the SET statement.

### Using the WHERE Statement with the OBS= and FIRSTOBS= Options

Another way to read and write only essential data is to process a segment of subsetted data. You accomplish this specialized task by using a WHERE expression in conjunction with the OBS= and FIRSTOBS= data set options. This programming feature is available in SAS 8.1 and later.

In the following example, the WHERE expression selects observations before the OBS= and FIRSTOBS= options are applied. The values specified for OBS= and FIRSTOBS= are the logical observation numbers in the subset, not the physical observation numbers in the data set.

```
proc print data=company.organization(firstobs=5 obs=8);
  var employee_id employee_gender salary;
  where salary>40000;
run;
```

| Obs | Employee_ID | Employee_Gender | Salary    |
|-----|-------------|-----------------|-----------|
| 101 | 120201      | Male            | \$43,280  |
| 157 | 120257      | Female          | \$156,245 |
| 158 | 120258      | Male            | \$83,305  |
| 159 | 120259      | Male            | \$433,800 |

FIRSTOBS = 5 is the fifth observation in the subset, whereas it was observation 101 in the data set **Company.Staff**.

OBS = 8 is the eighth observation in the subset, whereas it was observation 159 in the data set **Company.Staff**.

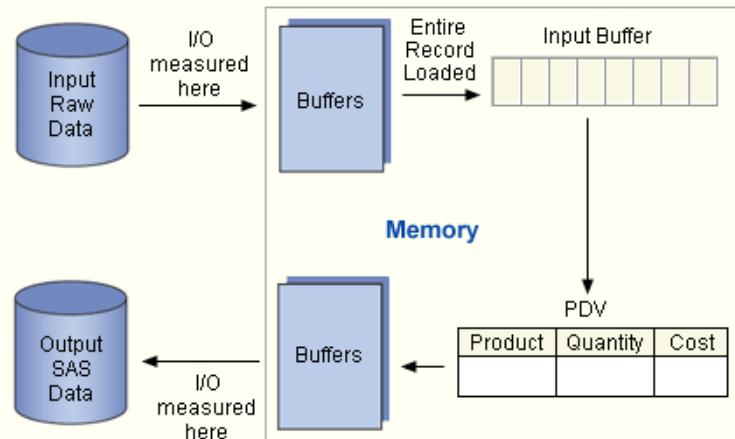
Now that you've seen techniques for efficiently subsetting observations that are read from SAS data sets, let's look at techniques for subsetting records that are read from external files.

Before viewing a comparative example that illustrates these techniques, it might be useful to review which resources are affected as SAS reads and processes data that is read from external files.

### Selecting Observations When Reading Data from External Files

Positioning a subsetting IF statement in a DATA step so that it reads only the variables that are needed to select the subset—before reading all the data—can reduce the overhead required for processing data.

The following graphic illustrates how data is read from an external file, loaded into the input buffer, and read into the program data vector.



Remember that I/O operations are measured as data moves between disks and buffers—for both input and output data. All records are loaded into the input buffer before moving to the program data vector for processing, so I/O is not affected by the placement of a subsetting IF statement in the DATA step.

You can reduce the CPU resources that are required for processing data by limiting what is read into the program data vector. Position a subsetting IF statement after an INPUT statement that reads only the data that is required in order to check for specific conditions. Subsequent statements do not execute and do not process variable values for unwanted observations.



Converting raw character fields to SAS character variables requires less CPU time than converting raw numeric fields to the real binary encoding of SAS numeric variables.



## Comparative Example: Creating a Subset of Data by Reading Data from an External File

Suppose you want to create a SAS data set by reading a subset of data from an external file that is referenced by the fileref **Customerdata**. You want the subset to contain only customers in the United Kingdom.

The subset is approximately 5.56% of the countries in the external file, which contains 89,954 records and 12 fields.

Techniques for doing this include

1. Reading All Variables and Subsetting
2. Reading Selected Variables and Subsetting.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

### Programming Techniques

#### 1. Reading All Variables and Subsetting

In this program, the INPUT statement reads the values for all variables before the subsetting IF statement checks for the value of Country. Then, if the value for Country is *GB*, the observation is written to the output data set **Retail.UnitedKingdom**.

```
data retail.UnitedKingdom;
infile customerdata;
input @1 Customer_ID      12.
      @13 Country      $2.
      @15 Gender       $1.
      @16 Personal_ID   $15.
      @31 Customer_Name $40.
      @71 Customer_FirstName $20.
      @91 Customer_LastName $30.
      @121 Birth_Date    date9.
      @130 Customer_Address $45.
      @175 Street_ID     12.
      @199 Street_Number   $8.
      @207 Customer_Type_ID  8.;

if country='GB';
run;
```

## 2. Reading Selected Variables and Subsetting

In this program, the first INPUT statement reads only the value for Country and holds the record in the input buffer using the single trailing @ sign. Then the program uses a subsetting IF statement to check for the value of Country. If the value for Country is **not** *GB*, values for other variables are not read in or written to the output data set **Retail.UnitedKingdom**. If the value for Country is **is** *GB*, values for other variables are input and written to the output data set **Retail.UnitedKingdom**.

```
data retail.UnitedKingdom;
infile customerdata;
input @13 Country $2. @;
if country='GB';
input @1 Customer_ID      12.
      @15 Gender       $1.
      @16 Personal_ID   $15.
      @31 Customer_Name $40.
      @71 Customer_FirstName $20.
      @91 Customer_LastName $30.
      @121 Birth_Date    date9.
      @130 Customer_Address $45.
      @175 Street_ID     12.
      @199 Street_Number   $8.
      @207 Customer_Type_ID  8.;

run;
```

## General Recommendations

- Position a subsetting IF statement in a DATA step so that only variables that are necessary to select the record are read before subsetting. This can result in significant savings in CPU time. There is no difference in I/O or memory usage between the two techniques.
- When selecting rows of data from an external file, read the field(s) on which the selection is being made before reading all the fields into the program data vector.
- Use the single trailing @ sign to hold the input buffer so that you can continue to read the record when the variable(s) satisfy the IF condition.

Which of the following statements is true?

- Q.12.**
- You can reduce I/O by using a subsetting IF statement to read only the data that is needed for checking specific conditions.
  - You can reduce I/O by using a subsetting IF statement to limit what is read into the program data vector.
  - You can reduce CPU processing by using a subsetting IF statement to limit what is loaded into the input buffer.
  - You can reduce CPU processing by using a subsetting IF statement to limit what is read into the program data vector.

What happens to CPU resources as the subset of data that is read from a raw data file becomes larger?

- Q.13.**
- As the subset becomes larger, a program uses more CPU resources.
  - As the subset becomes larger, a program uses fewer CPU resources.
  - As the subset becomes larger, CPU resources do not change.
  - As the subset becomes larger, CPU usage becomes greater than it would be for reading the entire data set.

In addition to subsetting observations, you can subset variables by using statements or options that efficiently read and write only essential data.

Before viewing two comparative examples that illustrate how to best limit which variables are read and processed, let's review how these useful statements and options work.

### Subsetting Variables with the KEEP= and DROP= Statements and Options

To subset variables, you can use

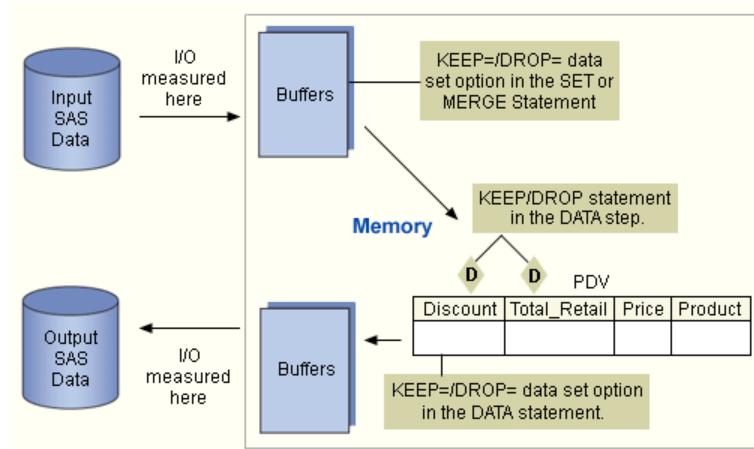
- the DROP and KEEP statements
- the DROP= and KEEP= data set options.



You cannot use the DROP and KEEP statements and the DROP= and KEEP= output data set options in the same step.

Use of the KEEP= data set option and the DROP= data set option can affect resource usage, depending on whether they are used in a SET or MERGE statement or in a DATA statement.

The following figure shows how options in these statements process data.



When used in the SET or MERGE statement, the KEEP= and DROP= data set options affect which variables are read into the program data vector. Reading only the variables that need to be processed in the DATA step can improve efficiency.

When used in the DATA statement, these same options put drop flags on variables to be excluded and affect which variables are written to the output data set.

The DROP and KEEP statements work just like the KEEP= or DROP= options in the DATA statement.

The following table describes differences in how the KEEP statement and the KEEP= data set option write variables to SAS data sets.

| KEEP Statement                                                                                                | KEEP= Output Data Set Option                                                                                  | KEEP= Input Data Set Option                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Causes a DATA step to write only the variables listed in the KEEP statement to one or more data sets.         | Causes a DATA step to write only the variables listed in the KEEP= variable list to the output data set.      | In the absence of a KEEP=/DROP= output data set option, causes the DATA step to write only the variables listed in the KEEP= variable list.                |
| In the absence of a KEEP=/DROP= input data set option, enables the DATA step to process all of the variables. | In the absence of a KEEP=/DROP= input data set option, enables the DATA step to process all of the variables. | Enables processing of only the variables listed in the KEEP= variable list.                                                                                |
| Applies to all data sets that are created within the same DATA step.                                          | Can write different variables to different data sets.                                                         | In the absence of a KEEP=/DROP= output data set option or KEEP/DROP statement, enables processing of only the variables listed in the KEEP= variable list. |
| Available only in the DATA step.                                                                              | Available in the DATA step or in a PROC step.                                                                 | Available in the DATA step or in a PROC step.                                                                                                              |

The following table describes differences in how the DROP statement and the DROP= data set option write variables to SAS data sets.

| DROP Statement              | DROP= Output Data Set Option | DROP= Input Data Set Option     |
|-----------------------------|------------------------------|---------------------------------|
| Causes a DATA step to write | Causes a DATA step to write  | In the absence of a KEEP=/DROP= |

|                                                                                                               |                                                                                                               |                                                                                                                                                                |
|---------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| only the variables not listed in the DROP statement to one or more data sets.                                 | only the variables not listed in the DROP= variable list to the output data set.                              | output data set option, enables processing of only the variables not listed in the DROP= variable list.                                                        |
| In the absence of a KEEP=/DROP= input data set option, enables the DATA step to process all of the variables. | In the absence of a KEEP=/DROP= input data set option, enables the DATA step to process all of the variables. | Enables processing of only the variables not listed in the DROP= variable list.                                                                                |
| Applies to all data sets created within the same DATA step.                                                   | Can write different variables to different data sets.                                                         | In the absence of a KEEP=/DROP= output data set option or KEEP/DROP statement, enables processing of only the variables not listed in the DROP= variable list. |
| Available only in the DATA step.                                                                              | Available in the DATA step or in a PROC step.                                                                 | Available in the DATA step or in a PROC step.                                                                                                                  |



## Comparative Example: Creating a Report That Contains Average and Median Statistics

Suppose you want to create a report that contains the average and median values for the variable Profit, based on data that is read from the data set **Retail.Order\_fact**. Depending on the number of variables eliminated, it might be more efficient to use the KEEP= option in a SET statement to limit which variables are read.

Techniques for reading and writing variables to a data set include

1. Without the KEEP= Statement
2. KEEP= in the DATA Statement
3. KEEP= in the DATA and SET Statements
4. KEEP= in the SET and MEANS Statements.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

### Programming Techniques

#### 1. Without the KEEP= Statement

This program reads all variables from the data set **Retail.Order\_fact** and does not restrict which variables are written to the output data set **Retail.Profit**. PROC MEANS reads all the variables from the data set.

```
data retail.profit;
  set retail.order_fact;
  if discount=. then
    Profit=(total_retail_price-costPrice_Per_Unit)*quantity;
    else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
  run;
  proc means data=retail.profit mean median maxdec=2;
    title 'Order Information';
    class employee_id;
    var profit;
  run;
```

## 2. KEEP= in the DATA Statement

This program uses the KEEP= data set option in the DATA statement to write two variables to the output data set **Retail.Profit**. PROC MEANS reads only two variables from the data set.

```
data retail.profit(keep=employee_id profit);
set retail.order_fact;
if discount=. then
  Profit=(total_retail_price-costprice_per_unit)*quantity;
else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit mean median maxdec=2;
  title 'Order Information';
  class employee_id;
  var profit;
run;
```

## 3. KEEP= in the DATA and SET Statements

This program uses the KEEP= option in the SET statement to read six variables from **Retail.Order\_fact**, and it uses the KEEP= data set option in the DATA statement to write two variables to the output data set **Retail.Profits**. PROC MEANS reads only two variables from the data set.

```
data retail.profits(keep=employee_id profit);
set retail.order_fact(keep=employee_id total_retail_price discount
                      costprice_per_unit quantity);
if discount=. then
  Profit=(total_retail_price-costprice_per_unit)*quantity;
else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit mean median maxdec=2;
  title 'Order Information';
  class employee_id;
  var profit;
run;
```

## 4. KEEP= in the SET and MEANS Statements

This program uses the KEEP= option in the SET statement to read selected variables from **Retail.Order\_fact**, and it uses the KEEP= data set option in the MEANS statement to process only the variables that are needed for the statistical report. You might do this if you need additional variables in **Retail.Profits** for further processing, but only two variables for processing by PROC MEANS.

```
data retail.profit;
set retail.order_fact(keep=employee_id total_retail_price discount
                      costprice_per_unit quantity);
if discount=. then
  Profit=(total_retail_price-costprice_per_unit)*quantity;
else Profit=((total_retail_price*discount)-costprice_per_unit)*quantity;
run;
proc means data=retail.profit(keep=employee_id profit) mean median maxdec=2;
  title 'Order Information';
  class employee_id;
  var profit;
run;
```

## General Recommendations

- To reduce both CPU time and I/O operations, avoid reading and writing variables that are not needed .

What is the difference between using the DROP= or KEEP= data set option in the DATA statement versus the SET statement?

Using these data set options in the DATA statement controls which variables SAS reads from the SAS data set. Using these data set options in the SET statement controls which variables SAS writes from the program data vector to the SAS data set.

**Q14.**

There is no difference in how these data set options behave when they are used in the DATA statement or the SET statement.

Using these data set options in the DATA statement controls which variables are written to the SAS data set from the program data vector. Using these data set options in the SET statement controls which variables are included in the program data vector and are made available for processing.

Using these options in the DATA statement does not allow variables to be read from a disk into the page buffer.

## Comparative Example: Creating a SAS Data Set That Contains Only Certain Variables

Suppose you want to read data from an external file that is referenced by the fileref **Rawdata** and to create a SAS data set that contains only the variables Customer\_ID, Country, Gender, and Customer\_Name.

Techniques for accomplishing this task include

1. Reading All Fields
2. Reading Selected Fields.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for reading and writing only essential data.

### Programming Techniques

#### **1. Reading All Fields**

In this program, the KEEP= data set option writes only the variables that are needed to the output data set, whereas the INPUT statement reads all fields from the external file.

```
data retail.customers(keep=Customer_ID Country Gender Customer_Name);
  infile rawdata;
  input @1 Customer_ID      12.
        @13 Country       $2.
        @15 Gender        $1.
        @16 Personal_ID   $15.
        @31 Customer_Name $40.
        @71 Customer_FirstName $20.
        @91 Customer_LastName $30.
        @121 Birth_Date    date9.
        @130 Customer_Address $45.
        @175 Street_ID     12.
        @199 Street_Number  $8.
        @207 Customer_Type_ID  8.;

run;
```

#### **2. Reading Selected Fields**

In this program, the INPUT statement reads selected fields from the external file, and by default, these are written to the output data set.

```
data retail.customers;
infile rawdata;
input @1 Customer_ID      12.
      @13 Country        $2.
      @15 Gender          $1.
      @31 Customer_Name   $40.;

run;
```

## General Recommendations

- When possible, read only the fields you need from an external data file to save CPU and real-time resources.
- To save CPU resources, avoid converting numerics that you do not need in further processing.



Remember that numeric data is moved into the program data vector after being converted to real binary, floating point numbers; multiple digits are stored in one byte. Character data is moved into the program data vector with no conversion; one character is stored in one byte.

When reading from a raw data file, what happens to CPU and I/O resources as the number of variables created becomes larger?

**Q15.**

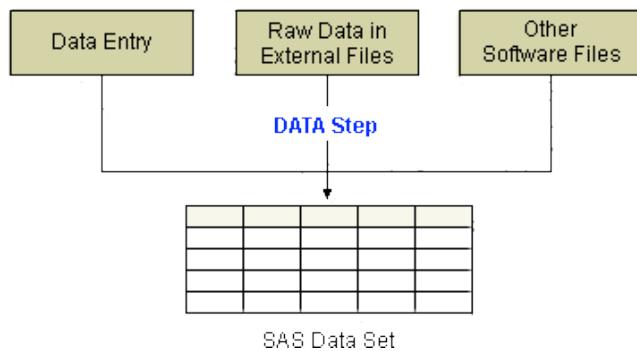
- The CPU usage increases, but the I/O remains the same.
- The CPU usage decreases, but the I/O remains the same.
- Both I/O and CPU usage increase.
- The I/O increases, but the CPU usage decreases.

## Storing Data in SAS Data Sets

In many cases, it is best practice for you to store data in SAS data sets. You can optimize performance if you know when you should create a SAS data set and when you should read data directly from an external file.

Before viewing the comparative example that illustrates different techniques for reading from a SAS data set versus from an external file, consider the following advantages of storing data in SAS data sets.

When you use SAS to repeatedly analyze or manipulate any particular group of data, it is more efficient to create a SAS data set than to read the raw data each time. Although SAS data sets can be larger than external files and can require more disk space, reading from SAS data sets saves CPU time that is associated with reading a raw data file.



Other reasons for storing data in SAS data sets, rather than external files, include:

- When the data is already in a SAS data set, you can use a SAS procedure, function, or routine on the data without further conversion.
- SAS data sets are self-documenting.

The descriptor portion of a SAS data set documents information about the data set such as

- data set labels
- variable labels
- variable formats
- informats
- descriptive variable names.

Create a temporary SAS data set if the data set is used for intermediate tasks such as merging and if it is needed in that SAS session only. Create a temporary SAS data set when the external file on which the data set is based might change between SAS sessions.

### Comparative Example: Creating a SAS Data Set or Reading from an External File

Suppose you want to create a SAS data set that contains a large number of variables. One way to accomplish this task is to read from raw data fields in an external file that is referenced by the fileref **Rawdata**. Another way to accomplish this is to read the same data values from an existing SAS data set named **Retail.Customer**.

1. Techniques for accomplishing this task include
2. Reading from an External File
3. Reading from a SAS Data Set.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for storing data in SAS data sets.

## Programming Techniques

### 1. Reading from an External File

In this program, the INPUT statement reads fields of data from an external file that is referenced by the fileref **Rawdata** and creates 12 variables. For benchmarking purposes, the DATA statement creates a **\_NULL\_** data set, testing for the effects of the reading

operation rather than the output processing.

```
data _null_;
infile rawdata;
input @1 Customer_ID      12.
      @13 Country        $2.
      @15 Gender          $1.
      @16 Personal_ID    $15.
      @31 Customer_Name   $40.
      @71 Customer_FirstName $20.
      @91 Customer_LastName $30.
      @121 Birth_Date     date9.
      @130 Customer_Address $45.
      @175 Street_ID      12.
      @199 Street_Number   $8.
      @207 Customer_Type_ID  8.;
run;
```

## 2. Reading from a SAS Data Set

In this program, the SET statement reads data directly from an existing SAS data set. As in the previous program, the DATA statement uses \_NULL\_ instead of naming a data set.

```
data _null_;
  set retail.customer;
run;
```

### General Recommendations

- To save CPU resources, you can read variables from a SAS data set instead of reading data from an external file.
- To reduce I/O operations, you can read variables from a SAS data set instead of reading data from an external file. However, savings in I/O operations are largely dependent on the block size of the external data file and on the page size of the SAS data set.

When you compare the size of an external file to the size of a SAS data set that is created from that file, which of the following statements is true?

**Q16.**

The SAS data set cannot be larger than the external file.

The SAS data set can be larger than an external file because the SAS data set includes a descriptor portion.

External files are always larger than SAS data sets.

The SAS data set can never be larger than the external file.

### Avoiding Unnecessary Procedure Invocation

Best practices dictate that you avoid unnecessary procedure invocation. One way to do this is to take advantage of procedures that accomplish multiple tasks with one invocation.

Several procedures enable you to create multiple reports by invoking the procedure only once. These include

- the SQL procedure
- the DATASETS procedure

- the FREQ procedure
- the TABULATE procedure.



BY-group processing can also minimize unnecessary invocations of procedures.

To illustrate this principle, let's examine features of the DATASETS procedure.

### **Executing the DATASETS Procedure**

The DATASETS procedure can use RUN-group processing to process multiple sets of statements. RUN-group processing enables you to submit groups of statements without ending the procedure.

When the DATASETS procedure executes,

- SAS reads the program statements that are associated with one task until it reaches a RUN statement or an implied RUN statement.
- SAS executes all of the preceding statements immediately, and then continues reading until it reaches another RUN statement or an implied RUN statement.

To execute the last task, you must use a RUN statement or a QUIT statement.

```
proc datasets lib=company;
  modify orders;
    rename quantity=Units_Ordered;
    format costprice_per_unit dollar13.2;
    label delivery_date='Date of Delivery';
  run;
  modify customers;
    format customer_birthdate mmddyy10.
  run;
quit;
```

You can terminate the PROC DATASETS execution by submitting

- a DATA statement
- a PROC statement
- a QUIT statement.

### **RUN-Group Processing**

If you can take advantage of RUN-group processing, you can avoid unnecessary procedure invocation. For best programming practices, you need to understand how RUN-group processing affects the execution of SAS statements. The procedures that support RUN-group processing include

- CHART, GCHART
- PLOT, GPLOT
- GIS, GMAP
- GLM
- REG
- DATASETS.

### **Using Different Types of RUN Groups with PROC DATASETS**

To illustrate how RUN-group processing works, this discussion focuses on the DATASETS procedure. The comparative example that follows includes programs that use PROC DATASETS to modify the descriptor portion of data sets. Before you examine the code to consider efficient programming techniques, review how the principles associated with RUN-group processing apply to PROC DATASETS.

The DATASETS procedure supports four types of RUN groups. Each RUN group is defined by the statements that compose it and by what causes it to execute.

Some statements in PROC DATASETS act as implied RUN statements because they cause the RUN group that precedes them to execute.

The following list identifies which statements compose a RUN group and what causes each RUN group to execute:

- The PROC DATASETS statement always executes immediately. No other statement is necessary to cause the PROC DATASETS statement to execute. Therefore, the PROC DATASETS statement alone is a RUN group.
- The MODIFY statement and any of its subordinate statements form a RUN group. These RUN groups always execute immediately. No other statement is necessary to cause a MODIFY RUN group to execute.
- The APPEND, CONTENTS, and COPY statements (including EXCLUDE and SELECT, if present) form their own separate RUN groups. Every APPEND statement forms a single-statement RUN group; every CONTENTS statement forms a single-statement RUN group; and every COPY step forms a RUN group. Any other statement in the procedure, except those that are subordinate to either the COPY or MODIFY statement, causes the RUN group to execute.

Additionally, one or more of the following statements form a RUN group:

- AGE
- EXCHANGE
- CHANGE
- REPAIR.

If any of these statements appear in sequence in the PROC step, the sequence forms a RUN group. For example, if a REPAIR statement appears immediately after a SAVE statement, the REPAIR statement does not force the SAVE statement to execute; it becomes part of the same RUN group. To execute the RUN group, submit one of the following statements:

- PROC DATASETS
- MODIFY
- APPEND
- QUIT
- CONTENTS
- RUN
- COPY
- another DATA or PROC step.



### Comparative Example: Modifying the Descriptor Portion of SAS Data Sets

Suppose you want to use the DATASETS procedure to modify the data sets **NewCustomer**, **NewOrders**, and **NewItems**.

Techniques for accomplishing this task include using

1. Multiple DATASETS Procedures
2. A Single DATASETS Procedure.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results may vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for avoiding unnecessary procedure invocation.

## Programming Techniques

### 1. Multiple DATASETS Procedures

This program invokes PROC DATASETS three times to modify the descriptor portion of the data set **NewCustomer**, two times to modify the descriptor portion of the data set **NewOrders**, and once to change the name of the data set **NewItems**.

```
proc datasets lib=company;
  modify newcustomer;
  rename Country_ID=Country
    Name=Customer_Name;
  quit;
proc datasets lib=company;
  modify newcustomer;
  format birth_date date9. ;
  quit;
proc datasets lib=company;
  modify newcustomer;
  label birth_date='Date of Birth';
  quit;
proc datasets lib=company;
  modify neworders;
  rename order=Order_ID
    employee=Employee_ID
    customer=Customer_ID;
  quit;
proc datasets lib=company;
  modify neworders;
  format order_date date9. ;
  quit;
proc datasets lib=company;
  change newitems=NewOrder_Items;
  quit;
```

### 2. Single DATASETS Procedure

This program invokes PROC DATASETS once to modify the descriptor portion of the data sets **NewCustomer** and **NewOrders**, and to change the name of the data set **NewItems**. This technique is more efficient.

```
proc datasets lib=company;
  modify newcustomer;
  rename country_ID=Country
    name=Customer_Name;
  format birth_date date9. ;
  label birth_date='Date of Birth';
  modify neworders;
  rename order=Order_ID
```

```
employee=Employee_ID  
customer=Customer_ID;  
format order_date date9.;  
change newitems>NewOrder_Items;  
quit;
```

### General Recommendations

- Invoke the DATASETS procedure once and process all the changes for a library in one step to save CPU and I/O resources—at the cost of memory resources.



Because the specified library could change between invocations of the DATASETS procedure, the procedure is reloaded into memory for each invocation.

Which technique is most efficient for modifying the descriptor portion of SAS data sets?

- Q17.
- multiple PROC CONTENTS procedures.
  - a single PROC CONTENTS procedure.
  - multiple DATASETS procedures.
  - a single DATASETS procedure.

## **Chapter Summary**

### **Executing Only Necessary Statements**

You minimize the CPU time that SAS uses when you execute the minimum number of statements in the most efficient order.

For a more efficient program, place the subsetting IF statement as soon as logically possible in a DATA step when creating a subset of data.

Review guidelines for using conditional logic efficiently with IF-THEN/ELSE statements or SELECT statements. Remember to minimize the number of statements that use SAS functions or arithmetic operators.

Conditionally execute only necessary statements by placing statements in groups that are associated with IF-THEN/ELSE statements or SELECT/WHEN statements. Groups of statements execute only when a particular condition is true. Review the criteria for using DO groups efficiently.

### **Eliminating Unnecessary Passes Through the Data**

You should avoid reading or writing data more than necessary in order to minimize I/O operations.

There are a variety of techniques that you can use. For example, use a single DATA step to create multiple output data sets from one pass of the input data, rather than using multiple DATA steps to process the input data each time that you create an output data set. Create sorted subsets by subsetting data with the SORT procedure rather than subsetting data in a DATA step and then sorting. Change variable attributes by using PROC DATASETS rather than a DATA step.

### **Reading and Writing Only Essential Data**

If you process fewer observations and variables, SAS performs fewer I/O operations. To limit the number of observations that are processed, you can use the subsetting IF statement and the WHERE statement. Best programming practices can be applied if you understand other differences between subsetting IF and WHERE statements. You can also improve performance by applying OBS= and FIRSTOBS= processing with a WHERE statement.

To select observations when reading data from external files, position a subsetting IF statement in a DATA step so that it reads only the variables that are needed to select the subset—before reading all the data. This can reduce the overhead required to process data.

To limit the number of variables that are processed, you can use

- the DROP and KEEP statements
- the DROP= and KEEP= data set options.

In the SET statement, the DROP= or KEEP= data set option controls which variables are read and subsequently processed. In the DATA statement, the DROP= or KEEP= data set option controls which variables are written to a data set after processing. Using the SET statement with these options is the most efficient and best practice.

### **Storing SAS Data in SAS Data Sets**

When you use SAS to repeatedly analyze or manipulate any particular group of data, create a SAS data set instead of reading the raw data each time.

Reading data from an external file versus reading from a SAS data set greatly increases CPU usage.

### **Avoiding Unnecessary Procedure Invocation**

Invoking procedures once rather than multiple times can be the most efficient way to process data. Several procedures enable you to create multiple reports by invoking the procedure only once.

Using a single DATASETS procedure instead of multiple DATASETS procedures to modify the descriptor portion of a data set results in a noticeable savings in both CPU and I/O operations. Also, you can take advantage of RUN-group processing to submit groups of statements without ending the procedure.

## Chapter Quiz

Select the best answer for each question and click Score My Quiz.

1. Placing the subsetting IF statement at the top rather than near the bottom of a DATA step results in a savings in CPU usage. What happens if the subset is large rather than small?
  - a. The savings in CPU usage increases as the subset grows larger because the I/O increases.
  - b. The savings in CPU usage decreases as the subset grows larger. However, placing the subsetting IF statement at the top of a DATA step always uses less resources than placing it at the bottom.
  - c. The savings in CPU usage remains constant as the subset grows larger. However, placing the subsetting IF statement near the bottom of a data set is preferable.
  - d. The savings in CPU usage decreases as the subset grows larger. However, placing the subsetting IF statement near the bottom of a data set increases the I/O.
2. Which of the following statements is true about techniques that are used for modifying data and attributes?
  - a. You can use PROC DATASETS to modify both data values and variable attributes.
  - b. You can use PROC DATASETS to modify only data values.
  - c. You can use the DATA step to modify both data values and variable attributes.
  - d. You can use the DATA step to modify only variable attributes.
3. For selecting observations, is a subsetting IF statement or a WHERE statement more efficient? Why?
  - a. A subsetting IF statement is more efficient because it loads all observations sequentially into the program data vector.
  - b. A subsetting IF statement is more efficient because it examines what is in the input buffer and selects observations before they are loaded into the program data vector, which results in a savings in CPU operations.
  - c. A WHERE statement is more efficient because it loads all observations sequentially into the program data vector.
  - d. A WHERE statement is more efficient because it examines what is in the input buffer and selects observations before they are loaded into the program data vector, which results in a savings in CPU operations.
4. When is it more advantageous to create a temporary SAS data set rather than a permanent SAS data set?
  - a. When the external file on which the data set is based might change between SAS sessions.
  - b. When the external file on which the data set is based does not change between SAS sessions.
  - c. When the data set is needed for more than one SAS session.
  - d. When you are converting raw numeric values to SAS data values.

5. When you compare the technique of using multiple DATASETS procedures to using a single DATASETS procedure to modify the descriptor portion of a data set, which is true?
  - a. A one-step DATASETS procedure results in an increase in I/O operations.
  - b. Multiple DATASETS procedures result in a decrease in I/O operations.
  - c. A one-step DATASETS procedure results in a decrease in CPU usage.
  - d. Multiple DATASETS procedures result in a decrease in CPU usage.

# Chapter 11

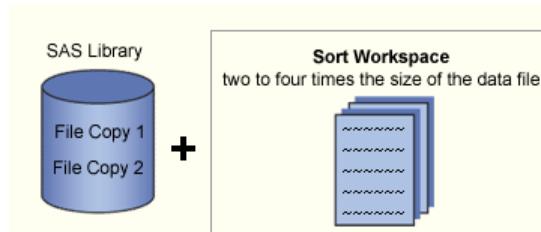
## Selecting Efficient Sorting Strategies

### Introduction

Sometimes you need to group observations by the values of a particular variable or order the observations in a particular way, such as alphabetically, in order to

- reorder the data for reporting
- reduce data retrieval time
- enable BY-group processing in both DATA and PROC steps.

The SORT procedure is one technique that can be used to group or order data. However, the SORT procedure can use a high volume of resources. When an uncompressed data file is sorted using the SORT procedure, SAS requires enough space in the data library for two copies of the data file, plus a workspace that is approximately two to four times the size of the data file.



In some cases, you might be able to use techniques other than the SORT procedure to group or order observations. In other cases, you might be able to use options or techniques with the SORT procedure that enable you to minimize resource usage.



This lesson does not cover the Scalable Performance Data Engine (SPD Engine), which is a SAS 9.1 technology for [threaded processing](#). For details about using the SPD Engine to improve performance, see the SAS documentation.

### Objectives

In this lesson, you learn to

- apply techniques that enable you to avoid unnecessary sorts
- calculate and allocate sort resources
- use strategies for sorting large data sets
- eliminate duplicate observations efficiently.

## Avoiding Unnecessary Sorts

In some cases you can avoid a sort by using

- BY-group processing with an index
- BY-group processing with the NOTSORTED option
- a CLASS statement
- the SORTEDBY= data set option.

### Using BY-Group Processing with an Index

BY-group processing is a method of processing observations from one or more SAS data sets that are grouped or ordered by the values of one or more common variables. You can use BY-group processing in both DATA steps and PROC steps.

#### General form, BY statement:

**BY variable(s);**

where *variable(s)* names each variable by which the data set is sorted or indexed.

The most common use of BY-group processing in the DATA step is to combine two or more SAS data sets by using the BY statement with a SET, MERGE, UPDATE, or MODIFY statement. When you use a SET, MERGE, or UPDATE statement, the data sets must first be ordered on the values of the BY variable unless you index the data sets. You can also use the NOTSORTED option in the BY statement with a SET statement.

When BY-group processing is used with an index that is based on one of the BY variables, the data can be sequenced without using the SORT procedure. The data can be sequenced by different variables if multiple indexes are used. Because indexes are updated automatically, there is no need to re-sort a data set when observations are modified or added.

However, using BY-group processing with an index has two disadvantages:

- It is generally less efficient than sequentially reading a sorted data set because processing BY groups typically means retrieving the entire file.
- It requires storage space for the index.



A BY statement does not use an index if the BY statement includes the DESCENDING or NOTSORTED option or if SAS detects that the data file is physically stored in sorted order on the BY variables.



If you use a MODIFY statement, the data does not need to be ordered. However, your program might run more efficiently with ordered data.



#### Comparative Example: Using BY-Group Processing with an Index to Avoid a Sort

Suppose you want to use an existing data set, **Retail.Order\_fact**, to create a new SAS data set that is ordered by the variable **Order\_Date**. You could accomplish this task using

1. BY-Group Processing with an Index, Data in Random Order

2. Presorted Data in a DATA Step
3. PROC SORT Followed by a DATA Step.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for BY-group processing with an index or sort.

## Programming Techniques

### **1. BY-Group Processing with an Index, Data in Random Order**

In this example, the SAS data set **Retail.Order\_fact** is indexed on the variable **Order\_Date**. The data in **Retail.Order\_fact** is in random order.

```
data _null_;
  set retail.order_fact;
  by order_date;
run;
```

### **2. Presorted Data in a DATA Step**

In this example, the SAS data set **Retail.Order\_fact** is sorted on the variable **Order\_Date** before it is read using the DATA step.

```
data _null_;
  set retail.order_fact;
  by order_date;
run;
```

### **3. PROC SORT Followed by a DATA STEP**

In this example, the SAS data set **Retail.Order\_fact** is sorted using the SORT procedure. The data is then read using the DATA step.

```
proc sort data=retail.order_fact;
  by order_date;
run;
data _null_;
  set retail.order_fact;
  by order_date;
run;
```

## General Recommendations

- To conserve resources, use sort order rather than an index for BY-group processing.
- Although using an index for BY-group processing is less efficient than using sort order, it might be the best choice if resource limitations make sorting a file difficult.

In which of the following situations is BY-group processing with a SAS data set least efficient?

**Q1.**

The data is indexed but not sorted.

The data is sorted but not indexed.

The data is neither sorted, indexed, or grouped.

## Using the NOTSORTED Option

You can also use the **NOTSORTED** option with a BY statement to create ordered or grouped reports without sorting the data. The NOTSORTED option specifies that observations that have the same BY value are grouped together but are not necessarily sorted in alphabetical or numeric order.

**General form, BY statement with the NOTSORTED option:**

**BY variable(s) NOTSORTED;**

where *variable(s)* names each variable by which the data set is sorted or indexed.

The NOTSORTED option can appear anywhere in the BY statement and is useful if you have data that is in logical categories or groupings such as chronological order. The NOTSORTED option works best when observations that have the same BY value are stored together.



The NOTSORTED option turns off sequence checking. If your data is not grouped, using the NOTSORTED option can produce a large amount of output.



The NOTSORTED option cannot be used with the MERGE or UPDATE statements.

**Example**

Suppose you want to use the PRINT procedure to print the contents of the data set **Retail.Europe**, which contains data about European customers. The data set includes values for country names (*Country\_Name*) as well as two-letter country codes (*Country\_Code*).

The data is grouped and sorted by the values of *Country\_Code*. However, you want the observations in the output to be grouped by the values of *Country\_Name*.

Country Codes and Country Names

| Country Code | Country Name   |
|--------------|----------------|
| BE           | Belgium        |
| DE           | Germany        |
| DK           | Denmark        |
| ES           | Spain          |
| FI           | Finland        |
| FR           | France         |
| GB           | United Kingdom |
| GR           | Greece         |
| IE           | Ireland        |
| IT           | Italy          |
| LU           | Luxembourg     |
| NL           | Netherlands    |

SAS Data Set Retail.Europe, Selected Observations

| Obs | Customer_ID | Customer_Country | Customer_Name | Country |
|-----|-------------|------------------|---------------|---------|
|-----|-------------|------------------|---------------|---------|

| Obs | Customer_ID | Customer_Country | Customer_Name        | Country |
|-----|-------------|------------------|----------------------|---------|
| 374 | 93803       | BE               | Luc Vandeloo         | Belgium |
| 375 | 93980       | BE               | Saida Van Itterbeeck | Belgium |
| 376 | 94040       | BE               | Pat Sodergard        | Belgium |
| 377 | 13          | DE               | Markus Sepke         | Germany |
| 378 | 19          | DE               | Oliver S. Füßling    | Germany |
| 379 | 50          | DE               | Gert-Gunter Mendler  | Germany |

You can use the NOTSORTED option with a BY statement to accomplish this task without using the SORT procedure.

```
proc print data=retail.europe;
  by country_name notsorted;
run;
```

PROC PRINT output shows that the data is grouped but is not in sorted order. For example, observations in which the value of Country\_Name is *Germany* are followed by observations in which the value of Country\_Name is *Denmark*.

| PROC PRINT Output, Selected Observations |             |                             |              |  |
|------------------------------------------|-------------|-----------------------------|--------------|--|
| Country_Name=Germany                     |             |                             |              |  |
| Obs                                      | Customer_ID | Customer_Name               | Country_Code |  |
| 2340                                     | 65751       | Dieter Krüger               | DE           |  |
| 2341                                     | 65769       | Fredy Ensinger              | DE           |  |
| 2342                                     | 65772       | Dieter Wein                 | DE           |  |
| 2343                                     | 65784       | Claudia Martin              | DE           |  |
| 2344                                     | 65793       | York Bückel                 | DE           |  |
| Country_Name=Denmark                     |             |                             |              |  |
| Obs                                      | Customer_ID | Customer_Name               | Country_Code |  |
| 2345                                     | 1331        | Else Gade Jensen            | DK           |  |
| 2346                                     | 1531        | Maria Bargmann Hersom       | DK           |  |
| 2347                                     | 1751        | Flemming Schønberg Schleiss | DK           |  |
| 2348                                     | 2175        | Marlene Lærke Shah          | DK           |  |
| 2349                                     | 2486        | Lars Flensted-Jensen        | DK           |  |

### Using FIRST. and LAST.

The NOTSORTED option can be used with FIRST.variable and LAST.variable, which are temporary automatic variables in the PDV that identify the first and last observations in each BY group.

These temporary variables are available for DATA step programming but are not added to the output data set. Their values indicate whether an observation is

- the first one in a BY group
- the last one in a BY group
- neither the first nor the last one in a BY group

- both first and last, as is the case when there is only one observation in a BY group.

You can take actions conditionally, based on whether you are processing the first observation of a BY group or the last.

When an observation is the first in a BY group, SAS sets the value of FIRST.variable to 1. For all other observations in the BY group, the value of FIRST.variable is 0. Likewise, if an observation is the last in a BY group, SAS sets the value of LAST.variable to 1. For all other observations in the BY group, the value of LAST.variable is 0.

### Example

The following program creates a new SAS data set **Work.New**. In the input data set, observations that have the same value for Ordername (*Retail*, *Catalog*, or *Internet*) are grouped together.

```
data work.new;
  set company.sales
  by ordername notsorted;
run;
```

When the program is submitted, SAS creates the temporary variables FIRST.Ordername and LAST.Ordername. These variables can be used during the DATA step, but they do not become variables in the new data set. The value 1 flags the beginning and end of each value in the BY group:

| Observations |             |           | Corresponding FIRST. and LAST. Values |                |
|--------------|-------------|-----------|---------------------------------------|----------------|
| Obs          | Customer_ID | OrderName | FIRST.OrderName                       | LAST.OrderName |
| 1            | 11791       | Retail    | 1                                     | 0              |
| 2            | 8406        | Retail    | 0                                     | 0              |
| 3            | 71020       | Retail    | 0                                     | 0              |
| 4            | 21735       | Retail    | 0                                     | 1              |
| 5            | 82141       | Catalog   | 1                                     | 0              |
| 6            | 30993       | Catalog   | 0                                     | 1              |
| 7            | 579         | Internet  | 1                                     | 0              |
| 8            | 77184       | Internet  | 0                                     | 1              |

### Using the GROUPFORMAT Option

The GROUPFORMAT option uses the formatted values of a variable instead of the internal values to determine where a BY group begins and ends, and how FIRST.variable and LAST.variable are computed.

**General form, BY statement with the GROUPFORMAT option:**

**BY variable(s) GROUPFORMAT;**

where variable(s) names each variable by which the data set is sorted or indexed.

The GROUPFORMAT option

- is available only in the DATA step

- is useful when you define formats for grouped data
- enables the DATA step to process the same groups of data as a summary procedure or PROC REPORT.

When the GROUPFORMAT option is used, the data set must be sorted by the GROUPFORMAT variable or grouped by the formatted values of the GROUPFORMAT variable.

### Example

Suppose you want to create a summary report that includes the number of orders for each quarter in 2002. The data for the report is stored in the SAS data set **Company.Orders**.

SAS Data Set Company.Orders, First Five Observations

| Obs | Order_ID   | Order_Type    | Employee_ID | Customer_ID | Order_Date | Delivery_Date |
|-----|------------|---------------|-------------|-------------|------------|---------------|
| 1   | 1230000033 | Internet Sale | 99999999    | 8818        | 01JAN1998  | 07JAN1998     |
| 2   | 1230000204 | Internet Sale | 99999999    | 47793       | 01JAN1998  | 04JAN1998     |
| 3   | 1230000268 | Internet Sale | 99999999    | 71727       | 01JAN1998  | 03JAN1998     |
| 4   | 1230000487 | Internet Sale | 99999999    | 74503       | 01JAN1998  | 04JAN1998     |
| 5   | 1230000494 | Internet Sale | 99999999    | 8610        | 01JAN1998  | 07JAN1998     |

By creating a format for the data and using the GROUPFORMAT and NOTSORTED options, you cause SAS to create the variables FIRST.Order\_Date and LAST.Order\_Date based on the formatted values, not the internal values. This groups the data without requiring the creation of a new variable.

```
proc format;
value qtrfmt '01jan2002' d - '31mar2002' d = '1'
              '01apr2002' d - '30jun2002' d = '2'
              '01Jul2002' d - '30sep2002' d = '3'
              '01Oct2002' d - '31dec2002' d = '4';

run;
data company.quarters(keep=Count order_date
    rename=(order_date=Quarter));
    set retail.orders;
    format order_date qtrfmt.;

by order_date groupformat notsorted;
where year(order_date)=2002;
if first.order_date then Count=0;
Count +1;
if last.order_date;
run;
```

SAS Data Set Company.Quarters

| Obs | Quarter | Count |
|-----|---------|-------|
| 1   | 1       | 4545  |
| 2   | 2       | 5330  |
| 3   | 3       | 5508  |
| 4   | 4       | 5649  |

### Using the CLASS Statement

You can also use a **CLASS statement** to avoid a sort. Unlike the BY statement, the CLASS statement does not require the data to be presorted using the CLASS values, nor does it require an index that is based on the CLASS variables.

If the data cannot be sorted, the CLASS statement is more efficient than the BY statement in terms of CPU time, memory, and I/O usage. However, unlike using the BY statement, presorting the data for use with a CLASS statement does not provide a significant benefit.

**General form, CLASS statement:**

**CLASS** *variable(s)* </ *options*>;

where *variable(s)* specifies one or more variables that the procedure uses to group the data.

Remember that a CLASS statement specifies the variables whose values define the subgroup combinations for an analysis by a SAS procedure. You can use the CLASS statement with the following Base SAS procedures:

- MEANS
- TABULATE
- SUMMARY
- UNIVARIATE.

Variables in a CLASS statement are referred to as **class variables**. Class variables can be numeric or character. Class variables can have continuous values, but they typically have a few discrete values that define the classifications of the variable.



The comparison of the use of CLASS and BY statements is appropriate for Base SAS procedures only.

**Example**

The **Company.Orders** data set contains the variable Order\_Type, which has three discrete values:

- *Retail*
- *Catalog*
- *Internet*.

Suppose you want to show the average retail price, cost per unit, and discount for each value of Order\_Type. You could use the MEANS procedure with either a BY statement or a CLASS statement to complete this task. The statistics created with either of these techniques are the same. However, the report layouts differ.

When the BY statement is used, SAS creates a report for each value of the BY variable. The statistics for each value of Order\_Type appear in a separate tabular report.

```
proc sort data=company.order_fact(keep=order_type
    quantity total_retail_price
    costprice_per_unit discount) out=company.orders;
  by order_type;
run;
proc means data=company.orders mean;
  by order_type;
```

```

var total_retail_price -- discount;
freq quantity;
run;

```

Output, PROC MEANS with a BY Statement

| Order_Type=Retail  |             |
|--------------------|-------------|
| Variable           | Mean        |
| Total_Retail_Price | 177.0073169 |
| CostPrice_Per_Unit | 38.3716413  |
| Discount           | 0.3881087   |

| Order_Type=Catalog |             |
|--------------------|-------------|
| Variable           | Mean        |
| Total_Retail_Price | 196.3417829 |
| CostPrice_Per_Unit | 40.8664533  |
| Discount           | 0.3885046   |

| Order_Type=Internet |             |
|---------------------|-------------|
| Variable            | Mean        |
| Total_Retail_Price  | 200.2136179 |
| CostPrice_Per_Unit  | 41.6397101  |
| Discount            | 0.3945766   |

When the CLASS statement is used, only one report is created. The statistics for each value of Order\_Type are consolidated into one tabular report.

```

proc means data=company.orders(keep=order_type
                                quantity total_retail_price
                                costprice_per_unit discount) mean;
  class order_type;
  var total_retail_price -- discount;
  freq quantity;
run;

```

Output, PROC MEANS with a CLASS Statement

| Order_Type | N Obs   | Variable           | Mean        |
|------------|---------|--------------------|-------------|
| Retail     | 1184633 | Total_Retail_Price | 177.0073169 |
|            |         | CostPrice_Per_Unit | 38.3716413  |
|            |         | Discount           | 0.3881087   |
| Catalog    | 222195  | Total_Retail_Price | 196.3417829 |
|            |         | CostPrice_Per_Unit | 40.8664533  |
|            |         | Discount           | 0.3885046   |
| Internet   | 190489  | Total_Retail_Price | 200.2136179 |
|            |         | CostPrice_Per_Unit | 41.6397101  |
|            |         | Discount           | 0.3945766   |

### Comparative Example: Using a BY or CLASS Statement to Avoid a Sort

Suppose you want to create a summary report that shows the average retail price, cost per unit, and discount for each type of order in the **Retail.Order\_fact** data set. Among the techniques you could use are

1. PROC MEANS with a BY Statement, Presorted
2. PROC MEANS with a CLASS Statement
3. PROC MEANS with a CLASS Statement, Presorted
4. PROC SORT and PROC MEANS with a BY Statement.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for using a BY or CLASS statement to avoid a sort.

## Programming Techniques

### **1. PROC MEANS with a BY Statement, Presorted**

The following program creates a report for each value of the BY variable Order\_Type. Each report contains the mean value for the variables Total\_Retail\_Price, CostPrice\_Per\_Unit, and Discount. The input data is presorted by the value of Order\_Type.

```
proc means data=retail.orders mean;
  by order_type;
  var total_retail_price costprice_per_unit discount;
  freq quantity;
run;
```

### **2. PROC MEANS with a CLASS Statement**

The following program uses a CLASS statement to create a single, tabular report that includes the mean value of Total\_Retail\_Price, CostPrice\_Per\_Unit, and Discount for each category of Order\_Type.

```
proc means data=retail.order_fact(keep=order_type
  quantity total_retail_price
  costprice_per_unit discount) mean;
  class order_type;
  var total_retail_price costprice_per_unit discount;
  freq quantity;
run;
```

### **3. PROC MEANS with a CLASS Statement, Presorted**

In the following program, the input data set **Retail.Order\_fact** is presorted by the value of Order\_Type. The PROC MEANS step uses a CLASS statement to create a single, tabular report that includes the mean value of Total\_Retail\_Price, CostPrice\_Per\_Unit, and Discount for each category of Order\_Type.

```
proc means data=retail.orders(keep=order_type
  quantity total_retail_price
  costprice_per_unit discount) mean;
  class order_type;
  var total_retail_price costprice_per_unit discount;
  freq quantity;
run;
```

### **4. PROC SORT and PROC MEANS with a BY Statement**

In the following program, the PROC SORT step first sorts the input data set **Retail.Order\_fact** by the value of Order\_Type. The PROC MEANS step then creates a report for each value of Order\_Type. Each report contains the mean value for the variables Total\_Retail\_Price, CostPrice\_Per\_Unit, and Discount.

```
proc sort data=retail.order_fact(keep=order_type
```

```

quantity total_retail_price
costprice_per_unit discount) out=retail.orders;
by order_type;
run;
proc means data=retail.orders mean;
by order_type;
var total_retail_price costprice_per_unit discount;
freq quantity;
run;

```

## General Recommendations

- If you can presort the data, use just a BY statement rather than a CLASS statement or a BY statement with a separate SORT procedure.
- If you cannot sort the data, use the CLASS statement.
- Do not presort the data for use with a CLASS statement. Presorting the data does not provide a significant benefit.

Which option can be used with a BY statement to specify that observations that have the same BY value are grouped together but are not necessarily sorted?

- Q.2.** NOTSORTED  
GROUPFORMAT  
CLASS

The GROUPFORMAT option

- Q.3.** does not require the data to be sorted or grouped.  
can be used with a CLASS statement.  
uses the formatted values of the BY variable to determine where BY groups begin and end.
- When used with Base SAS procedures, the CLASS statement  
can be used only with character variables.  
does not require the data to be presorted using the BY-variable values.  
requires an index that is based on the BY-variable values.

## Using the SORTEDBY= Data Set Option

If you are working with input data that is already sorted, you can specify how the data is ordered by using the SORTEDBY= data set option.

### General form, SORTEDBY= data set option:

**SORTEDBY=***by-clause </collate-name> | \_NULL\_*

where

- *by-clause* indicates the data order
- *collate-name* names the collating sequence that is used for the sort

- NULL removes any existing sort information.



By default, the collating sequence is that of your operating environment. For details on collating sequences, see the SAS documentation for your operating environment.

Although the SORTEDBY= option does not sort a data set, it sets the value of the **Sorted** flag. It does not set the value of the **Validated** sort flag. (PROC SORT sets the Validated sort flag.) To see the values of these flags (YES or NO), use PROC CONTENTS.

```
proc contents;
  data=company.transactions;
run;
```

Partial PROC CONTENTS Output

|                            |                           |                             |      |
|----------------------------|---------------------------|-----------------------------|------|
| <b>Data Set Name</b>       | COMPANY.TRANSACTIONS      | <b>Observations</b>         | 5504 |
| <b>Member Type</b>         | DATA                      | <b>Variables</b>            | 3    |
| <b>Engine</b>              | V9                        | <b>Indexes</b>              | 0    |
| <b>Created</b>             | 22:55 Friday, May 2, 2004 | <b>Observation Length</b>   | 32   |
| <b>Last Modified</b>       | 22:55 Friday, May 2, 2004 | <b>Deleted Observations</b> | 0    |
| <b>Protection</b>          |                           | <b>Compressed</b>           | NO   |
| <b>Data Set Type</b>       |                           | <b>Sorted</b>               | YES  |
| <b>Label</b>               |                           |                             |      |
| <b>Data Representation</b> | WINDOWS                   |                             |      |
| <b>Encoding</b>            | wlatin1 Western (Windows) |                             |      |
| <b>Sort Information</b>    |                           |                             |      |
| <b>Sortedby</b>            | Invoice                   |                             |      |
| <b>Validated</b>           | NO                        |                             |      |
| <b>Character Set</b>       | ANSI                      |                             |      |



Most SAS procedures and subsystems check the order of the data as it is processed unless the Validated sort flag is set on the file.



Using the Sorted flag without the Validated sort flag improves the speed of index creation.

### Example: Using the SORTEDBY= Data Set Option

Suppose you want to create a sorted SAS data set from an external file that contains invoice information. The external file is already sorted by invoice number.

You can use the SORTEDBY= data set option to sort the data by the value of Invoice.

```
data company.transactions (sortedby=invoice);
  infile extdata;
  input Invoice 1-4 Item $6-20 Amount comma 6.;
run;
```

When the **Company.Transactions** data set is created, the sort information is stored with it. PROC SORT checks the sort information before it sorts a data set so that data is not re-sorted unnecessarily. If you

attempt to re-sort the data, the log indicates that the data set is already sorted and that no additional sorting occurred.

```
proc sort data=company.transactions;
  by invoice;
run;
```

#### SAS Log

```
667 proc sort data=work.transactions;
668 by invoice;
669 run;

NOTE: Input data set is already sorted, no sorting done.
NOTE: PROCEDURE SORT used (Total process time):
      real time      0.07 seconds
      cpu time      0.03 seconds
```



You can specify `SORTEDBY=_NULL_` to remove the Sorted flag. The Sorted flag is also removed if you change or add any values of the variables by which the data set is sorted.

The `SORTEDBY=` data set option

**Q.5.**

sorts the input data and sets the Sorted flag on the data set.

sorts the input data but does not set the Sorted flag on the data set.

does not sort the input data but sets the Sorted flag on the data set.

## Using a Threaded Sort

[Threaded processing](#) takes advantage of multiple CPUs by executing multiple threads in parallel (parallel processing). Threaded procedures are completed in less real time than if each task were handled sequentially, although the CPU time is generally increased.

Beginning with SAS 9, the `SORT` procedure can take advantage of threaded processing. A thread is a single, independent flow of control through a program or within a process.

Threaded sorting is enabled or disabled by using the SAS system option `THREADS | NOTHREADS` or the `THREADS | NOTHREADS` procedure option.

**General form, `SORT` procedure with the `THREADS | NOTHREADS` option:**

**`PROC SORT SAS-data-set-name THREADS | NOTHREADS;`**

where

- `SAS-data-set-name` is a valid SAS data set name
- `THREADS` enables threaded sorting
- `NOTHREADS` disables threaded sorting.



The `THREAD | NOTHREADS` procedure option overrides the value of the SAS system option `THREADS | NOTHREADS`. For information about the `THREADS | NOTHREADS` system option, see the SAS documentation.

When a threaded sort is used, the observations in the input data set are divided into equal temporary subsets, based on the number of processors that are allocated to the SORT procedure. Each subset is then sorted on a different processor. The sorted subsets are then interleaved to re-create the sorted version of the input data set.

### Using the CPUCOUNT= System Option

The performance of threaded sorting is affected by the value of the CPUCOUNT= system option. CPUCOUNT= specifies the number of processors that thread-enabled applications should assume will be available for concurrent processing. SAS uses this information to determine how many threads to start, not to restrict the number of CPUs that will be used.

#### General form, CPUCOUNT= system option:

**CPUCOUNT= *n* | ACTUAL;**

where

- *n* is a number from 1 to 1024 that indicates how many CPUs SAS will assume are available for use by thread-enabled applications
- **ACTUAL** causes SAS to detect how many CPUs are available for a specific session.



Setting CPUCOUNT= to a number greater than the actual number of available CPUs might result in reduced overall performance.



For more information about the CPUCOUNT= system option and other options that are relevant to SAS threading technology, see the SAS documentation.

When a thread-enabled procedure is submitted, CPU time

**Q.6.** generally increases.

generally decreases.

usually remains the same.

## Calculating and Allocating Sort Resources

### Sort Space Requirements

When data is sorted, SAS requires enough space in the data library for two copies of the data file that is being sorted as well as additional workspace.

In releases prior to SAS 9, the workspace required for an uncompressed data file is approximately three to four times the size of the data file. Beginning with SAS 9, the workspace required for an uncompressed data file is approximately twice the size of the data file. The workspace can be allocated in memory and/or on disk as a utility file, depending on the sort utility and on the options chosen.

You can use the following formula to calculate the amount of workspace that the SORT procedure requires:

### Formula for calculating the amount of workspace needed to sort a SAS data set:

bytes required = (*key-variable-length* + *observation-length*) \* *number-of-observations* \* 4  
where

- *key-variable-length* is the length of all key variables added together
- *observation-length* is the maximum observation length
- *number-of-observations* is the number of observations.



The multiplier 4 applies only to utility files used in releases prior to SAS 9, when PROC SORT needs to use disk space in order to sort the data. For in-memory sorting and sorting with SAS 9 and later, the multiplier is 2 or less.

### Example

Suppose you want to submit the following program under SAS 9:

```
proc sort data=company.customers;
  by customer_group customer_lastname;
run;
```

You can use the CONTENTS procedure or the DATASETS procedure to obtain the information that is required for the calculation.

Partial PROC CONTENTS Output

|                     |                                |                      |       |
|---------------------|--------------------------------|----------------------|-------|
| Data Set Name       | COMPANY.CUSTOMERS              | Observations         | 89954 |
| Member Type         | DATA                           | Variables            | 11    |
| Engine              | V9                             | Indexes              | 0     |
| Created             | 13:36 Thursday, August 7, 2003 | Observation Length   | 200   |
| Last Modified       | 10:38 Friday, August 8, 2003   | Deleted Observations | 0     |
| Protection          |                                | Compressed           | NO    |
| Data Set Type       |                                | Sorted               | NO    |
| Label               |                                |                      |       |
| Data Representation | WINDOWS                        |                      |       |
| Encoding            | wlatin1 Western (Windows)      |                      |       |

Alphabetic List of Variables and Attributes

| #  | Variable           | Type | Len | Format     | Label               |
|----|--------------------|------|-----|------------|---------------------|
| 11 | Customer_Age       | Num  | 3   |            | Customer Age        |
| 8  | Customer_Age_Group | Char | 12  |            | Customer Age Group  |
| 7  | Customer_BirthDate | Num  | 4   | DATE9.     | Customer Birth Date |
| 2  | Customer_Country   | Char | 2   | \$COUNTRY. | Customer Country    |
| 5  | Customer_FirstName | Char | 20  |            | Customer First Name |
| 3  | Customer_Gender    | Char | 1   | \$GENDER.  | Customer Gender     |
| 10 | Customer_Group     | Char | 40  |            | Customer Group Name |
| 1  | Customer_ID        | Num  | 8   | 12.        | Customer ID         |
| 6  | Customer_LastName  | Char | 30  |            | Customer Last Name  |

| Alphabetic List of Variables and Attributes |               |      |     |        |                    |
|---------------------------------------------|---------------|------|-----|--------|--------------------|
| #                                           | Variable      | Type | Len | Format | Label              |
| 4                                           | Customer_Name | Char | 40  |        | Customer Name      |
| 9                                           | Customer_Type | Char | 40  |        | Customer Type Name |

In this case, the amount of workspace needed to sort the data set is 48,575,160 bytes.

$$48,575,160 \text{ bytes} = (70+200)*89954*2$$



The SORT procedure is very I/O intensive. If the file that you are sorting is located in the **Work** library, all of the I/O for the procedure takes place on the file system in which the **Work** library is stored because, by default, the utility files that the SORT procedure creates are created in the **Work** library.

Beginning with SAS 9, you can use the UTILLOC= system option to specify one or more file systems in which utility files can be stored. For information about the UTILLOC= option, see the SAS documentation.

### Using the SORTSIZE= Option

The SORTSIZE= system option or procedure option specifies how much memory is available to the SORT procedure. Specifying the SORTSIZE= option in the PROC SORT statement temporarily overrides the SAS system option SORTSIZE=.

#### General form, SORTSIZE= option:

**SORTSIZE=***memory-specification*;

where *memory-specification* specifies the maximum amount of memory that is available to PROC SORT. Valid values for *memory-specification* are as follows:

- MAX specifies that all available memory can be used
- *n* specifies the amount of memory in bytes, where *n* is a real number
- *nK* specifies the amount of memory in kilobytes, where *n* is a real number
- *nM* specifies the amount of memory in megabytes, where *n* is a real number
- *nG* specifies the amount of memory in gigabytes, where *n* is a real number.



The default value of the SORTSIZE= option depends on your operating environment. See the SAS documentation for your operating environment for more information.

Generally, the value of SORTSIZE= should be less than the physical memory that is available to your process.

If the required workspace is less than or equal to the value specified in the SORTSIZE= system option or procedure option, then the entire sort can take place in memory, which reduces processing time.

If the actual required workspace is greater than the value specified in the SORTSIZE= system option or procedure option, then processing time is increased because the SORT procedure must

1. create temporary utility files in the **Work** directory or mainframe temporary area
2. request memory up to the value specified by SORTSIZE=
3. write a portion of the sorted data to a utility file.

This process is repeated until all of the data is sorted. The SORT procedure then interleaves the data in the utility files to create the final data set.

PROC SORT attempts to adapt to the constraint that is imposed by the SORTSIZE= option. Because PROC SORT uses memory as much as possible,

- a small SORTSIZE= value can increase CPU and I/O resource utilization
- a large SORTSIZE= value can decrease CPU and I/O resource utilization.

What happens when the workspace that is required for completing a sort is greater than the value specified in the SORTSIZE= option?

- Q.7.** Processing time is increased.  
Processing time is decreased.  
Processing time remains the same.

## Handling Large Data Sets

### Dividing a Large Data Set

A data set is too large to sort when there is insufficient room in the data library for a second copy of the data set or when there is insufficient disk space for three to four temporary copies of the data set.

One approach to this situation is to divide the large data set into smaller data sets. The smaller data sets can then be sorted and combined to re-create the large data set. This approach is similar to the process that is used in a threaded sort.

Techniques for dividing and sorting a large data set include

- using PROC SORT with the OUT= statement option and the FIRSTOBS= and OBS= data set options
- using PROC SORT with a WHERE statement
- using subsetting with IF-THEN/ELSE or SELECT-WHEN logic to create multiple output data sets, then sorting the output data sets.

Techniques that can be used to rebuild a large data from smaller, sorted data sets set include

- concatenating the smaller data sets with a SET statement
- interleaving the smaller data sets with SET and BY statements
- appending the smaller data sets with the APPEND procedure.

### Comparative Example: Dividing and Sorting a Large Data Set 1

Suppose you want to sort the SAS data set **Retail.Order\_fact** by the value of Order\_Date. The data set is too large to sort using a single SORT procedure. You could accomplish this task by

1. Segmenting by Observation
2. Subsetting Using an IF Statement with the YEAR Function
3. Subsetting Using an IF Statement with a Date Constant
4. Subsetting Using a WHERE Statement with the YEAR Function
5. Subsetting Using a WHERE Statement with a Date Constant.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for dividing and sorting a large data set.

## Programming Techniques

### 1. Segmenting by Observation

This program segments the data set **Orion.Order\_fact** into three smaller data sets by observation number. The three smaller data sets, **Work.One**, **Work.Two**, and **Work.Three**, are sorted by the value of **Order\_Date**. The large data set is then re-created by interleaving the three smaller, sorted data sets.

```
proc sort data=retail.order_fact
  (firstobs = 1 obs = 1500000)
  out=work.one;
  by order_date;
run;

proc sort data=retail.order_fact
  (firstobs = 1500001 obs = 3000000)
  out=work.two;
  by order_date;
run;

proc sort data=retail.order_fact
  (firstobs = 3000001)
  out=work.three;
  by order_date;
run;

data work.orders;
  set work.one work.two work.three;
  by order_date;
run;
```

### 2. Subsetting Using an IF Statement with the YEAR Function

This program segments the data set **Orion.Order\_fact** into three smaller data sets by using a subsetting IF statement and the **YEAR** function. The three smaller data sets, **Work.One**, **Work.Two**, and **Work.Three**, are then sorted by the value of **Order\_Date**. The large data set is then re-created by concatenating the three smaller, sorted data sets. Interleaving is not required because the smaller data sets do not overlap each other on the sort key **Order\_Date**.

```
data work.one work.two work.three;
set retail.order_fact;
year=year(order_date);
if year in (1998,1999)
  then output work.one;
else if year in (2000,2001)
  then output work.two;
else output work.three;
run;

proc sort data=work.one;
by order_date;
run;

proc sort data=work.two;
by order_date;
run;

proc sort data=work.three;
by order_date;
run;
```

```

by order_date;
run;

data work.orders;
  set work.one work.two work.three;
run;

```

### 3. Subsetting Using an IF Statement with a Date Constant

This program segments the data set **Orion.Order\_fact** into three smaller data sets by using a subsetting IF statement and a date constant. The three smaller data sets, **Work.One**, **Work.Two**, and **Work.Three**, are then sorted by the value of Order\_Date. The large data set is then re-created by concatenating the three smaller, sorted data sets. Interleaving is not required because the smaller data sets do not overlap each other on the sort key Order\_Date.

```

data work.one work.two work.three;
  set retail.order_fact;
  if order_date <= '31Dec1999'd then
    output work.one;
  else if '31dec1999'd < order_date < '01jan2002'd
    then output work.two;
  else output work.three;
run;

proc sort data=work.one;
  by order_date;
run;

proc sort data=work.two;
  by order_date;
run;

proc sort data=work.three;
  by order_date;
run;

data work.orders;
  set work.one work.two work.three;
run;

```

### 4. Subsetting Using a WHERE Statement with the YEAR Function

This program segments the data set **Orion.Order\_fact** into three smaller data sets by using a WHERE statement and the YEAR function. The three smaller data sets, **Work.One**, **Work.Two**, and **Work.Three**, are sorted by the value of Order\_Date. The large data set is then re-created by concatenating the three smaller, sorted data sets. Interleaving is not required because the smaller data sets do not overlap each other on the sort key Order\_Date.

```

proc sort data=retail.order_fact
  out=work.one;
  by order_date;
  where year(order_date) in (1998, 1999);
run;

proc sort data=retail.order_fact
  out=work.two;
  by order_date;
  where year(order_date) in (2000, 2001);
run;

proc sort data=retail.order_fact

```

```

out=work.three;
by order_date;
where year(order_date) in (2002);
run;

data work.orders;
  set work.one work.two work.three;
run;

```

## 5. Subsetting Using a WHERE Statement with a Date Constant

This program segments the data set **Orion.Order\_fact** into three smaller data sets by using a WHERE statement with a date constant. The three smaller data sets, **Work.One**, **Work.Two**, and **Work.Three**, are sorted by the value of Order\_Date. The large data set is then re-created by concatenating the three smaller, sorted data sets. Interleaving is not required because the smaller data sets do not overlap each other on the sort key Order\_Date.

```

proc sort data=retail.order_fact
  out=work.one;
  by order_date;
  where order_date le '31Dec1999'd;
run;

proc sort data=retail.order_fact
  out=work.two;
  by order_date;
  where order_date between '01jan2000'd and
    '31dec2001'd;
run;

proc sort data=retail.order_fact
  out=work.three;
  by order_date;
  where order_date ge '01jan2002'd;
run;

data work.orders;
  set work.one work.two work.three;
run;

```

## General Recommendations

- Use a DATA step rather than PROC APPEND to re-create a large data set from smaller subsets.
- Use a constant rather than a SAS function because calling a function repeatedly increases CPU usage.
- Use a subsetting IF with either a constant or a function rather than a WHERE statement with a function.

## Comparative Example: Dividing and Sorting a Large Data Set 2

Like the programs shown on the previous page, each of the following programs illustrates a method for dividing the large data set **Retail.Order\_fact** into smaller data sets for sorting. However, in this example, the smaller data sets **Work.One**, **Work.Two**, and **Work.Three** are combined using the APPEND procedure rather than a DATA step in programs 2, 3, 4, and 5.

1. Segmenting by Observation
2. Subsetting Using an IF Statement with the YEAR Function
3. Subsetting Using an IF Statement with a Date Constant

4. Subsetting Using a WHERE Statement with the YEAR Function
5. Subsetting Using a WHERE Statement with a Date Constant.

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for dividing and sorting a large data set.

## Programming Techniques

### **1. Segmenting by Observation**

This program segments the data set **Orion.Order\_fact** into three smaller data sets by observation number. The three smaller data sets, **Work.One**, **Work.Two**, and **Work.Three**, are sorted by the value of **Order\_Date**. PROC APPEND cannot be used to re-create a large data set that was segmented using FIRSTOBS= and OBS=. Therefore, the large data set is re-created by interleaving the three smaller, sorted data sets.

```

proc sort data=retail.order_fact
  (firstobs = 1 obs = 1500000)
  out=work.one;
  by order_date;
run;

proc sort data=retail.order_fact
  (firstobs = 1500001 obs = 3000000)
  out=work.two;
  by order_date;
run;

proc sort data=retail.order_fact
  (firstobs = 3000001)
  out=work.three;
  by order_date;
run;
data work.orders;
  set work.one work.two work.three;
  by order_date;
run;
```

### **2. Subsetting Using an IF Statement with the YEAR Function**

This program segments the data set **Orion.Order\_fact** into three smaller data sets by using a subsetting IF statement and the YEAR function. The three smaller data sets, **Work.One**, **Work.Two**, and **Work.Three**, are then sorted by the value of **Order\_Date**. The large data set is then re-created using the APPEND procedure.

```

data work.one work.two work.three;
set retail.order_fact;
year=year(order_date);
if year in (1998,1999)
  then output work.one;
else if year in (2000,2001)
  then output work.two;
else output work.three;
run;

proc sort data=work.one;
  by order_date;
run;
```

```

proc sort data=work.two;
  by order_date;
run;
proc sort data=work.three;
  by order_date;
run;

proc append base=work.orders data=work.one;
run;
proc append base=work.orders data=work.two;
run;
proc append base=work.orders data=work.three;
run;

```

### **3. Subsetting Using an IF Statement with a Date Constant**

This program segments the data set **Orion.Order\_fact** into three smaller data sets by using a subsetting IF statement and a date constant. The three smaller data sets, **Work.One**, **Work.Two**, and **Work.Three**, are then sorted by the value of Order\_Date. The large data set is then re-created using the APPEND procedure.

```

data work.one work.two work.three;
  set retail.order_fact;
  if order_date <= '31Dec1999'd then
    output work.one;
  else if '31dec1999'd < order_date < '01jan2002'd
    then output work.two;
  else output work.three;
run;

proc sort data=work.one;
  by order_date;
run;

proc sort data=work.two;
  by order_date;
run;

proc sort data=work.three;
  by order_date;
run;

proc append base=work.orders data=work.one;
run;
proc append base=work.orders data=work.two;
run;
proc append base=work.orders data=work.three;
run;

```

### **4. Subsetting Using a WHERE Statement with the YEAR Function**

This program segments the data set **Orion.Order\_fact** into three smaller data sets by using a WHERE statement and the YEAR function. The three smaller data sets, **Work.One**, **Work.Two**, and **Work.Three**, are sorted by the value of Order\_Date. The large data set is then re-created using the APPEND procedure.

```

proc sort data=retail.order_fact
  out=work.one;

```

```

by order_date;
where year(order_date) in (1998, 1999);
run;
proc sort data=retail.order_fact
  out=work.two;
  by order_date;
where year(order_date) in (2000, 2001);
run;
proc sort data=retail.order_fact
  out=work.three;
  by order_date;
where year(order_date) in (2002);
run;

proc append base=work.orders data=work.one;
run;
proc append base=work.orders data=work.two;
run;
proc append base=work.orders data=work.three;
run;

```

## 5. Subsetting Using a WHERE Statement with a Date Constant

This program segments the data set **Orion.Order\_fact** into three smaller data sets by using a WHERE statement with a date constant. The three smaller data sets, **Work.One**, **Work.Two**, and **Work.Three**, are sorted by the value of Order\_Date. The large data set is then re-created using the APPEND procedure.

```

proc sort data=retail.order_fact
  out=work.one;
  by order_date;
where order_date le '31Dec1999'd;
run;
proc sort data=retail.order_fact
  out=work.two;
  by order_date;
where order_date between '01jan2000'd and
  '31dec2001'd;
run;
proc sort data=retail.order_fact
  out=work.three;
  by order_date;
where order_date ge '01jan2002'd;
run;

proc append base=work.orders data=work.one;
run;
proc append base=work.orders data=work.two;
run;
proc append base=work.orders data=work.three;
run;

```

## General Recommendations

- Use a DATA step rather than PROC APPEND to re-create a large data set from smaller subsets.

## Using the TAGSORT Option

You can also use the **TAGSORT** option to sort a large data set. The TAGSORT option stores only the BY variables and the observation numbers in temporary files. The BY variables and the observation numbers are called **tags**. At the completion of the sorting process, PROC SORT uses the tags to retrieve records from the input data set in sorted order.

**General form, SORT procedure with the TAGSORT option:**

**PROC SORT DATA=SAS-data-set-name TAGSORT;**

where **SAS-data-set-name** is a valid SAS data set name.

When the total length of the BY variables is small compared to the record length, TAGSORT reduces temporary disk usage considerably because sorting just the BY variables means sorting much less data. However, processing time is usually higher than if a regular sort is used because TAGSORT increases CPU time and I/O usage in order to save memory and disk space. TAGSORT

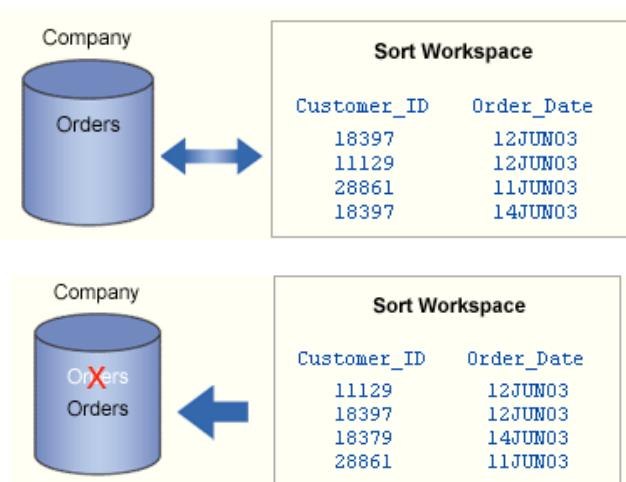
- uses significantly more CPU time and I/O than a regular sort if the data is extremely out of order with regard to the BY variables
- uses slightly more CPU time and I/O than a regular sort if the data is mostly in order with regard to the BY variables.

**Example**

In the following program, only the BY variables, Customer\_ID and Order\_Date, and the tags are stored in temporary files within the sort workspace. SAS then

- sorts the temporary files
- uses the tags to retrieve the observations from the original data set
- re-creates the sorted data set.

```
proc sort data=company.orders tagSort;  
  by customer_id order_date;  
run;
```



The TAGSORT option is not supported by the threaded sort.

Which technique generally requires the fewest resources for re-creating a large data set that has been divided for sorting?

- Q.8.**
- concatenating the smaller data sets with a SET statement
  - appending the smaller data sets with the APPEND procedure
  - merging the smaller data sets with MERGE and BY statements.

When the total length of the BY variables is small compared to the record length, using the TAGSORT option

- Q.9.**
- increases memory usage.
  - reduces processing time.
  - reduces temporary disk usage.

## Removing Duplicate Observations Efficiently

The SORT procedure can be used to remove duplicate observations when it is

- used with the NODUPKEY option
- used with the NODUPRECS option
- followed by FIRST. processing in the DATA step.

Generally, PROC SORT with the NODUPKEY option uses less I/O and CPU time than PROC SORT followed by a DATA step that uses FIRST. processing. Before viewing a comparative example, let's examine each of the techniques that are listed above.

### Using the NODUPKEY Option

The NODUPKEY option checks for and eliminates observations that have duplicate BY-variable values. If you specify this option, then PROC SORT compares **all BY-variable values** for each observation to those for the previous observation that was written to the output data set. If an exact match is found, then the observation is not written to the output data set.

#### General form, PROC SORT with the NODUPKEY option:

**PROC SORT DATA=SAS-data-set-name NODUPKEY;**

where SAS-data-set-name is a valid SAS data set name.

### Example

The SAS data set **Company.Reorder** contains two duplicated observations. Observation 9 is a duplicate of observation 1, and observation 7 is a duplicate of observation 2. The duplicate observations are removed when the data is sorted by the values of Product\_Line and Product\_Name and when the NODUPKEY option is used.

```
proc sort data=company.reorder nodupkey;
  by product_line product_name;
run;
```

SAS Data Set Company.Reorder

| Obs | Product_Line | Product_Name | Supplier_Name |
|-----|--------------|--------------|---------------|
|-----|--------------|--------------|---------------|

| Obs | Product_Line    | Product_Name                    | Supplier_Name             |
|-----|-----------------|---------------------------------|---------------------------|
| 1   | Children        | Ski Jacket w/Removable Fleece   | Scandinavian Clothing A/S |
| 2   | Children        | Kids Children's Fleece Hat      | 3Top Sports               |
| 3   | Clothes & Shoes | Watchit 120 Sterling/Reflective | Eclipse Inc               |
| 4   | Sports          | Sparkle Spray Blue              | CrystalClear Optics Inc   |
| 5   | Outdoors        | Money Purse, Black              | Top Sports                |
| 6   | Sports          | Mayday Serious Down Jacket      | Mayday Inc                |
| 7   | Children        | Kids Children's Fleece Hat      | 3Top Sports               |
| 8   | Clothes & Shoes | Tyfoon Linen Pants              | Typhoon Clothing          |
| 9   | Children        | Ski Jacket w/Removable Fleece   | Scandinavian Clothing A/S |

SAS Data Set Company.Reorder, Before Removing Duplicate Observations

| Obs | Product_Line    | Product_Name                               | Supplier_Name             |
|-----|-----------------|--------------------------------------------|---------------------------|
| 1   | Children        | Kids Children's Fleece Hat                 | 3Top Sports               |
| 2   | Children        | Kids Children's Fleece Hat                 | 3Top Sports               |
| 3   | Children        | Ski Jacket w/Removable Fleece              | Scandinavian Clothing A/S |
| 4   | Children        | Ski Jacket w/Removable Fleece              | Scandinavian Clothing A/S |
| 5   | Clothes & Shoes | Tyfoon Linen Pants                         | Typhoon Clothing          |
| 6   | Clothes & Shoes | Watchit 120 Sterling/Reflective            | Eclipse Inc               |
| 7   | Outdoors        | Money Purse, Black                         | Top Sports                |
| 8   | Sports          | Mayday Serious Down Jacket                 | Mayday Inc                |
| 9   | Sports          | Sparkle Spray Blue CrystalClear Optics Inc |                           |

SAS Data Set Company.Reorder, Duplicate Observations Removed

| Obs | Product_Line    | Product_Name                               | Supplier_Name             |
|-----|-----------------|--------------------------------------------|---------------------------|
| 1   | Children        | Kids Children's Fleece Hat                 | 3Top Sports               |
| 2   | Children        | Ski Jacket w/Removable Fleece              | Scandinavian Clothing A/S |
| 3   | Clothes & Shoes | Tyfoon Linen Pants                         | Typhoon Clothing          |
| 4   | Clothes & Shoes | Watchit 120 Sterling/Reflective            | Eclipse Inc               |
| 5   | Outdoors        | Money Purse, Black                         | Top Sports                |
| 6   | Sports          | Mayday Serious Down Jacket                 | Mayday Inc                |
| 7   | Sports          | Sparkle Spray Blue CrystalClear Optics Inc |                           |

### Using the NODUPRECS Option

The NODUPRECS option also checks for and eliminates duplicate observations. However, unlike the NODUPKEY option, the NODUPRECS option compares **all of the variable values** for each observation to those for the previous observation that was written to the output data set. If an exact match is found, then the observation is not written to the output data set.

**General form, PROC SORT with the NODUPRECS option:**

**PROC SORT DATA=SAS-data-set-name NODUPRECS;**

where SAS-data-set-name is a valid SAS data set name.



NODUP is an alias for NODUPRECS.

Because NODUPRECS checks only consecutive observations, some nonconsecutive duplicate observations might remain in the output data set. You can remove all duplicates with this option by sorting on all variables.

**Example**

When **Company.Reorder** is sorted using the BY variable Product\_Line and the NODUPRECS option, the duplicate observation that contains the product name *Kids Children's Fleece Hat* is eliminated because it exactly matches the observation that was previously written to the output data set. The duplicate observation that contains the product name *Ski Jacket w/Removable Fleece* is retained because it does not exactly match the observation that was previously written to the output data set.

```
proc sort data=company.reorder noduprecs;
  by product_line;
run;
```

**SAS Data Set Company.Reorder**

| Obs | Product_Line    | Product_Name                    | Supplier_Name             |
|-----|-----------------|---------------------------------|---------------------------|
| 1   | Children        | Ski Jacket w/Removable Fleece   | Scandinavian Clothing A/S |
| 2   | Children        | Kids Children's Fleece Hat      | 3Top Sports               |
| 3   | Clothes & Shoes | Watchit 120 Sterling/Reflective | Eclipse Inc               |
| 4   | Sports          | Sparkle Spray Blue              | CrystalClear Optics Inc   |
| 5   | Outdoors        | Money Purse, Black              | Top Sports                |
| 6   | Sports          | Mayday Serious Down Jacket      | Mayday Inc                |
| 7   | Children        | Kids Children's Fleece Hat      | 3Top Sports               |
| 8   | Clothes & Shoes | Tyfoon Linen Pants              | Typhoon Clothing          |
| 9   | Children        | Ski Jacket w/Removable Fleece   | Scandinavian Clothing A/S |

**SAS Data Set Company.Reorder, Before Removing Duplicate Observation**

| Obs | Product_Line    | Product_Name                    | Supplier_Name             |
|-----|-----------------|---------------------------------|---------------------------|
| 1   | Children        | Ski Jacket w/Removable Fleece   | Scandinavian Clothing A/S |
| 2   | Children        | Kids Children's Fleece Hat      | 3Top Sports               |
| 3   | Children        | Kids Children's Fleece Hat      | 3Top Sports               |
| 4   | Children        | Ski Jacket w/Removable Fleece   | Scandinavian Clothing A/S |
| 5   | Clothes & Shoes | Watchit 120 Sterling/Reflective | Eclipse Inc               |
| 6   | Clothes & Shoes | Tyfoon Linen Pants              | Typhoon Clothing          |
| 7   | Outdoors        | Money Purse, Black              | Top Sports                |
| 8   | Sports          | Sparkle Spray Blue              | CrystalClear Optic Inc    |
| 9   | Sports          | Mayday Serious Down Jacket      | Mayday Inc                |

SAS Data Set Company.Reorder, Duplicate Observation Remaining

| Obs | Product_Line    | Product_Name                    | Supplier_Name             |
|-----|-----------------|---------------------------------|---------------------------|
| 1   | Children        | Ski Jacket w/Removable Fleece   | Scandinavian Clothing A/S |
| 2   | Children        | Kids Children's Fleece Hat      | 3Top Sports               |
| 3   | Children        | Ski Jacket w/Removable Fleece   | Scandinavian Clothing A/S |
| 4   | Clothes & Shoes | Watchit 120 Sterling/Reflective | Eclipse Inc               |
| 5   | Clothes & Shoes | Typhoon Linen Pants             | Typhoon Clothing          |
| 6   | Outdoors        | Money Purse, Black              | Top Sports                |
| 7   | Sports          | Sparkle Spray Blue              | CrystalClear Optic Inc    |
| 8   | Sports          | Mayday Serious Down Jacket      | Mayday Inc                |

Both duplicate observations are removed when **Company.Reorder** is sorted by both Product\_Line and Product\_Name and when the NODUPRECS option is used.

```
proc sort data=company.reorder noduprecs;
  by product_line product_name;
run;
```

SAS Data Set Company.Reorder, Both Duplicate Observations Removed

| Obs | Product_Line    | Product_Name                    | Supplier_Name             |
|-----|-----------------|---------------------------------|---------------------------|
| 1   | Children        | Kids Children's Fleece Hat      | 3Top Sports               |
| 2   | Children        | Ski Jacket w/Removable Fleece   | Scandinavian Clothing A/S |
| 3   | Clothes & Shoes | Typhoon Linen Pants             | Typhoon Clothing          |
| 4   | Clothes & Shoes | Watchit 120 Sterling/Reflective | Eclipse Inc               |
| 5   | Outdoors        | Money Purse, Black              | Top Sports                |
| 6   | Sports          | Mayday Serious Down Jacket      | Mayday Inc                |
| 7   | Sports          | Sparkle Spray Blue              | CrystalClear Optics Inc   |



The SORTDUP= system option controls how NODUPRECS processing works. Specifying SORTDUP=PHYSICAL removes duplicates based on all variables in the data set. This is the default. Specifying SORTDUP=LOGICAL removes duplicates based only on the variables that remain after the DROP= and KEEP= data set options are processed. See the SAS documentation for more information.

### Using the EQUALS | NOEQUALS Option

EQUALS | NOEQUALS is a SORT procedure option that helps to determine the order of observations in the output data set. When you use NODUPRECS or NODUPKEY to remove observations from the output data set, the choice of EQUALS or NOEQUALS can have an effect on which observations are removed.

EQUALS is the default. For observations that have identical BY-variable values, EQUALS maintains the order from the input data set in the output data set. NOEQUALS does not necessarily preserve this order in the output data set. NOEQUALS can save CPU time and memory resources.

### Example

The following program uses PROC SORT with the NODUPKEY option and the NOEQUALS option to create an output data set that contains only the first observation in each BY group. Notice that the output data set **Work.New** contains different observations when the EQUALS option is used.

```
proc sort data=company.products out=work.new
  nodupkey noequals;
  by product_line;
run;
```

**SAS Data Set Company.Products**

| Obs | Product_Line    | Product_Name                           | Supplier_Name             |
|-----|-----------------|----------------------------------------|---------------------------|
| 1   | Clothes & Shoes | Big Guy Men's Ringer T                 | Eclipse Inc               |
| 2   | Children        | Boy's and Girl's Ski Pants with Braces | Scandinavian Clothing A/S |
| 3   | Outdoors        | Cotton Moneybelt/Polyester 45x11       | Prime Sports Ltd          |
| 4   | Sports          | Cougar Shorts                          | SD Sporting Goods Inc     |
| 5   | Clothes & Shoes | Far Out Teambag S                      | 3Top Sports               |
| 6   | Children        | Kid Basic Tracking Suit                | Triple Sportswear Inc     |
| 7   | Sports          | Maxrun Ultra short Sprinter Tights     | Force Sports              |
| 8   | Clothes & Shoes | Wa.leather Street Shoes                | Fuller Trading Co.        |

**SAS Data Set Work.New, NOEQUALS Option Used**

| Obs | Product_Line    | Product_Name                       | Supplier_Name         |
|-----|-----------------|------------------------------------|-----------------------|
| 1   | Children        | Kid Basic Tracking Suit            | Triple Sportswear Inc |
| 2   | Clothes & Shoes | Far Out Teambag S                  | 3Top Sports           |
| 3   | Outdoors        | Cotton Moneybelt/Polyester 45x11   | Prime Sports Ltd      |
| 4   | Sports          | Maxrun Ultra short Sprinter Tights | Force Sports          |

**SAS Data Set Work.New, EQUALS Option Used**

| Obs | Product_Line    | Product_Name                           | Supplier_Name             |
|-----|-----------------|----------------------------------------|---------------------------|
| 1   | Children        | Boy's and Girl's Ski Pants with Braces | Scandinavian Clothing A/S |
| 2   | Clothes & Shoes | Big Guy Men's Ringer T                 | Eclipse Inc               |
| 3   | Outdoors        | Cotton Moneybelt/Polyester 45x11       | Prime Sports Ltd          |
| 4   | Sports          | Cougar Shorts                          | SD Sporting Goods Inc     |



The EQUALS | NOEQUALS option is supported by the threaded sort. However, I/O performance might be reduced when you use the EQUALS option because partitioned data sets will be processed as if they are non-partitioned data sets.



The order of observations within BY groups that are returned by the threaded sort might not be consistent between runs. Therefore, using the NOEQUALS option can produce inconsistent results in your output data sets.

## Using FIRST. LAST. Processing in the DATA Step

FIRST. LAST. processing in the DATA step can also be used to remove duplicate observations from a SAS data set.

In the data set **Company.Onorder**, the fourth observation contains a duplicate value for Product\_Name. The following program removes the observation that contains the duplicate value by first sorting the input data set, **Company.Onorder**, by the value of Product\_Name. The DATA step then selects only the first observation in the BY group.

```
proc sort data=company.onorder
  out=work.sorted;
  by product_name;
run;
data work.onorder2;
  set work.sorted;
  by product_name;
  if first.product_name;
run;
```

SAS Data Set Company.Onorder

| Obs | Product_Line    | Product_Name                           | Quantity |
|-----|-----------------|----------------------------------------|----------|
| 1   | Clothes & Shoes | Big Guy Men's Ringer T                 | 70       |
| 2   | Children        | Boy's and Girl's Ski Pants with Braces | 55       |
| 3   | Outdoors        | Cotton Moneybelt/Polyester 45x11       | 20       |
| 4   | Sports          | Big Guy Men's Ringer T                 | 70       |
| 5   | Sports          | Cougar Shorts                          | 40       |
| 6   | Clothes & Shoes | Far Out Teambag S                      | 32       |
| 7   | Children        | Kid's Basic Tracking Suit              | 20       |
| 8   | Sports          | Maxrun Ultra short Sprinter Tights     | 25       |
| 9   | Clothes & Shoes | Wa.leather Street Shoes                | 30       |

SAS Data Set Work.Onorder2

| Obs | Product_Line    | Product_Name                           | Quantity |
|-----|-----------------|----------------------------------------|----------|
| 1   | Clothes & Shoes | Big Guy Men's Ringer T                 | 70       |
| 2   | Children        | Boy's and Girl's Ski Pants with Braces | 55       |
| 3   | Outdoors        | Cotton Moneybelt/Polyester 45x11       | 20       |
| 4   | Sports          | Cougar Shorts                          | 40       |
| 5   | Clothes & Shoes | Far Out Teambag S                      | 32       |
| 6   | Children        | Kid's Basic Tracking Suit              | 20       |
| 7   | Sports          | Maxrun Ultra short Sprinter Tights     | 25       |
| 8   | Clothes & Shoes | Wa.leather Street Shoes                | 30       |



### Comparative Example: Removing Duplicate Observations Efficiently

Suppose you want to remove observations from the data set **Retail.Order\_fact** in which the value of Order\_Date is duplicated. Among the techniques you could use are

1. The NODUPKEY Option and the EQUALS Option

2. The NODUPKEY Option and the NOEQUALS Option
3. PROC SORT and a DATA Step with BY-Group and FIRST. Processing

The following sample programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for eliminating duplicates.

## Programming Techniques

### 1. The NODUPKEY Option and the EQUALS Option

This program uses the NODUPKEY option and the EQUALS option to check for and eliminate observations that have duplicate BY-variable values. For observations that have identical BY-variable values, the EQUALS option maintains the order from the input data set in the output data set.

```
proc sort data=retail.order_fact
    out=work.sorted
    nodupkey equals;
    by order_date;
run;
```

### 2. The NODUPKEY Option and the NOEQUALS Option

This program uses the NODUPKEY option and the NOEQUALS option to check for and eliminate observations that have duplicate BY-variable values. For observations that have identical BY-variable values, the NOEQUALS option does not necessarily maintain the order from the input data set in the output data set.

```
proc sort data=retail.order_fact
    out=work.sorted
    nodupkey noequals;
    by order_date;
run;
```

### 3. PROC SORT and a DATA Step with BY-Group and FIRST. Processing

In this program, the input data set is first sorted using the SORT procedure. Duplicate observations are then removed using BY-group and FIRST. processing.

```
proc sort data=retail.order_fact
    out=work.sorted;
    by order_date;
run;
data work.sorted;
    set work.sorted;
    by order_date;
    if first.order_date;
run;
```

## General Recommendations

- To remove duplicate observations from a SAS data set, use PROC SORT with the NODUPKEY option rather than a PROC SORT step followed by a DATA step that uses FIRST. processing.
- Be careful not to confuse NODUPKEY with NODUPRECS.

### The NODUPKEY option

- Q10.** The NODUPKEY option  
 compares all BY-variable values for each observation to those for the previous  
observation that was written to the output data set

**Q.11.**

- observation that was written to the output data set.  
compares all BY-variable values for each observation to the following observation in the input data set.  
compares all of the variable values for each observation to those for the previous observation that was written to the output data set.

When used with the NODUPKEY option or the NODUPRECS option, the NOEQUALS option always maintains the order from the input data set in the output data set. is the default. can save CPU time and memory resources.

## Additional Features

### Selecting a Host Sort Utility

Host sort utilities are third-party sort packages that are available in some operating environments. In some cases, using a host sort utility with PROC SORT might be more efficient than using the SAS sort utility with PROC SORT.

The following table lists the host sort utilities that might be available at your site. SAS uses the values that are set for the SORTPGM=, SORTCUT=, SORTCUTP= and SORTNAME= system options to determine which sort to use.

| Operating Environment | Host Sort Utilities          |
|-----------------------|------------------------------|
| z/OS                  | Dfsort (default)<br>Syncsort |
| UNIX                  | Cosort<br>Syncsort (default) |
| Windows               | Syncsort                     |



Ask your system administrator whether a host sort utility is available at your site. For more information about host sort utilities, see the SAS documentation for your operating environment.

### Using the SORTPGM= System Option

The value specified in the SORTPGM= system option tells SAS whether to use the SAS sort, to use the host sort, or to determine which sort utility is best for the data set.

#### General form, SORTPGM= system option:

**OPTIONS SORTPGM= BEST | HOST | SAS;**

where

- BEST specifies that SAS chooses the sort utility. This is the default.
- HOST specifies that the host sort utility is always used
- SAS specifies that the SAS sort utility is always used.

### Using the SORTCUTP= System Option

The SORTCUTP= system option specifies the **number of bytes** above which the host sort utility is used instead of the SAS sort utility.

#### General form, SORTCUTP= system option:

**OPTIONS SORTCUTP=n | nK | nM | nG | MIN | MAX | hexX;**

where

- $n$  |  $nK$  |  $nM$  |  $nG$  | specifies the value in bytes, kilobytes, megabytes, or gigabytes, respectively
- MIN specifies the minimum value
- MAX specifies the maximum value
- $hexX$  specifies the value as a hexadecimal number of bytes.



To determine the minimum and maximum values for SORTCUTP=, see the SAS documentation for your operating environment.

The following table lists the default values for SORTCUTP= in the z/OS, UNIX, and Windows operating environments.

| Operating Environment | Default SORTCUTP= Value | Default Behavior                             |
|-----------------------|-------------------------|----------------------------------------------|
| z/OS                  | 4M                      | SAS sort is used until this value is reached |
| UNIX                  | 0                       | SAS sort is always used                      |
| Windows               | 0                       | SAS sort is always used                      |

#### Using the SORTCUT= System Option

Beginning with SAS 9, the SORTCUT= system option can be used to specify the **number of observations** above which the host sort utility is used instead of the SAS sort utility.



The SORTCUT= system option is not available in the z/OS operating environment.

#### General form, SORTCUT= system option:

**OPTIONS SORTCUT=n | nK | nM | nG | MIN | MAX | hexX;**

where

- $n$  |  $nK$  |  $nM$  |  $nG$  | specifies the number of observations.
- MIN specifies 0 observations
- MAX specifies the maximum number of observations
- $hexX$  specifies the number of observations in hexadecimal notation.



To determine the maximum value for SORTCUT=, see the SAS documentation for your operating environment.

The default value of the SORTCUT= system option is 0.

#### Using the SORTNAME= System Option

The SORTNAME= option specifies the host sort utility that will be used if the value of SORTPGM= is *BEST* or *HOST*.



The SORTNAME= system option is not available in the Windows operating environment.

**General form, SORTNAME= option:**

**OPTIONS SORTNAME=*host-sort-utility name*;**

where *host-sort-utility name* is the name of a valid sort host utility.

**Example**

When you specify SORTPGM=BEST, SAS uses the value of the SORTCUT= and SORTCUTP= options to determine whether to use the host sort or the SAS sort. If you specify values for both the SORTCUT= and SORTCUTP= options, and if either condition is true, SAS chooses the host sort.

In the program below, if the size of the SAS data set **Company.Orders** is larger than 10,000 bytes, the host sort utility, Syncsort, will be used instead of the SAS sort utility.

```
options sortpgm=best sortcutf=10000
      sortname=syncsort;
proc sort data=company.orders out=company.deliveries;
  by delivery_date;
run;
```

**Chapter Summary**

**Avoiding Unnecessary Sorts**

When BY-group processing with an index is used, the data can be sequenced by different variables without having to repeat the SORT procedure if multiple indexes are used. Because indexes are updated automatically, there is no need to re-sort a data set when observations are modified or added. However, BY-group processing with an index is less efficient than reading a sorted data set sequentially, and storage space is required for the index.

You can also use the NOTSORTED option with a BY statement to create ordered or grouped reports without sorting the data. The NOTSORTED option specifies that observations that have the same BY value are grouped together but are not necessarily sorted in alphabetical or numeric order. The NOTSORTED option works best when observations that have the same BY value are stored together.

The NOTSORTED option can be used with FIRST. and LAST., which are temporary automatic variables in the PDV that identify the first and last observations in each BY group. These temporary variables are available for DATA step programming but are not added to the output data set.

The GROUPFORMAT option is useful when you have defined formats for grouped data. The GROUPFORMAT option uses the formatted values of a variable, instead of the internal values to determine where a BY group begins and ends, and how FIRST. and LAST. are computed. When the GROUPFORMAT option is used, the data set must be sorted by the GROUPFORMAT variable or grouped by the formatted values of the GROUPFORMAT variable.

You can use a CLASS statement to specify the variables whose values define the subgroup combinations for an analysis by a SAS procedure. Unlike the BY statement, when the CLASS statement is used with Base SAS procedures, it does not require the data to be presorted using the BY-variable values or that you have an index based on the BY variables. If the data cannot be sorted, the CLASS statement is more efficient than the BY statement in terms of CPU time, memory, and I/O usage.

If you are working with input data that is already sorted, you can specify how the data is ordered by using the SORTEDBY= data set option. Although the SORTEDBY= option does not sort a data set, it sets the Sorted flag on the data set.

Review the related comparative examples:

- Using BY-Group Processing with an Index to Avoid a Sort
- Using a BY or CLASS Statement to Avoid a Sort.

### **Using a Threaded Sort**

Beginning with SAS 9, the SORT procedure can take advantage of threaded processing. Threaded jobs are completed in substantially less real time than if each task is handled sequentially. However, the CPU time for threaded jobs is generally increased

Threaded sorting is enabled or disabled by using the THREADS | NOTREADS SAS system option or procedure option. The procedure option overrides the value of the system option.

When a threaded sort is used, the observations in the input data set are divided into equal temporary subsets, based on how many processors are allocated to the SORT procedure. Each subset is then sorted on a different processor. The sorted subsets are then interleaved to recreate the sorted version of the input data set.

The performance of a threaded sort is affected by the value of the CPUCOUNT= system option. CPUCOUNT= specifies the number of processors that thread-enabled applications should assume will be available for concurrent processing. SAS uses this information to determine how many threads to start, not to restrict the number of CPUs that will be used.

### **Calculating and Allocating Sort Resources**

When data is sorted, SAS requires enough space in the data library for two copies of the data file that is being sorted, as well as additional workspace.

In releases prior to SAS 9, the required workspace is approximately three to four times the size of the data file. Beginning with SAS 9, the required workspace is approximately twice the size of the data file. The workspace can be allocated in memory and/or on disk as a utility file, depending on which sort utility and options are specified.

The SORTSIZE= option specifies how much memory is available to the SORT procedure. Generally, the value of SORTSIZE= should be less than the physical memory that is available to your process. If the required workspace is less than or equal to the value specified in the SORTSIZE= system option or procedure option, then the entire sort can take place in memory, which reduces processing time.

### **Handling Large Data Sets**

A data set is too large to sort when there is insufficient room in the data library for a second copy of the data set or when there is insufficient disk space for three to four temporary copies of the data set.

One approach to this situation is to divide the large data set into smaller subsets. The subsets can then be sorted and combined to re-create the large data set.

You can also use the TAGSORT option to sort a large data set. The TAGSORT option stores only the BY variables and the observation numbers in temporary files. The BY variables and the observation numbers

are called tags. At the completion of the sorting process, PROC SORT uses the tags to retrieve records from the input data set in sorted order.

When the total length of the BY variables is small compared to the record length, TAGSORT reduces temporary disk usage considerably because sorting just the BY variables means sorting much less data. However, processing time might be much higher because the TAGSORT option increases CPU and I/O usage in order to save memory and disk space.

Review the related comparative examples:

- Dividing and Sorting a Large Data Set 1
- Dividing and Sorting a Large Data Set 2.

### **Removing Duplicate Observations Efficiently**

The NODUPKEY option checks for and eliminates observations that have duplicate BY-variable values. If you specify this option, then PROC SORT compares all BY-variable values for each observation to those for the previous observation that was written to the output data set. If an exact match is found, then the observation is not written to the output data set.

The NODUPRECS option checks for and eliminates duplicate observations. However, unlike the NODUPKEY option, the NODUPRECS option compares all of the variable values for each observation to those for the previous observation that was written to the output data set. If an exact match is found, then the observation is not written to the output data set.

EQUALS | NOEQUALS is a procedure option that helps to determine the order of observations in the output data set. When you use NODUPRECS or NODUPKEY to remove observations from the output data set, the choice of EQUALS or NOEQUALS can have an effect on which observations are removed.

EQUALS is the default. For observations that have identical BY-variable values, EQUALS maintains the order from the input data set in the output data set. NOEQUALS does not necessarily preserve this order in the output data set. NOEQUALS can save CPU time and memory resources.

FIRST. LAST. processing in the DATA step can also be used to remove duplicate observations in a SAS data set.

Review the related comparative example:

- Removing Duplicate Observations Efficiently.

### **Additional Features**

Depending on your operating environment, you might be able to use additional sorting options, called host sort utilities. Host sort utilities are third-party sort packages. In some cases, using a host sort utility might be more efficient than using the SAS sort utility with PROC SORT.

SAS uses the values that are set for the SORTPGM=, SORTCUTP=, SORTCUT= and SORTNAME= system options to determine which sort utility to use.

## Chapter Quiz

Select the best answer for each question.

- When the following program is submitted, what is the value of FIRST.Product\_Line for the third observation in the data set **Work.Products**?

```
data new.products;
  set work.products
  by product_line notsorted;
run;
```

SAS Data Set Work.Products

| Obs | Product_Line    | Product_Name                    | Supplier_Name             |
|-----|-----------------|---------------------------------|---------------------------|
| 1   | Children        | Kids Children's Fleece Hat      | 3Top Sports               |
| 2   | Children        | Ski Jacket w/Removable Fleece   | Scandinavian Clothing A/S |
| 3   | Clothes & Shoes | Tyfoon Linen Pants              | Typhoon Clothing          |
| 4   | Clothes & Shoes | Watchit 120 Sterling/Reflective | Eclipse Inc               |
| 5   | Clothes & Shoes | Money Belt, Black               | Top Sports                |

- a. 1  
b. 3  
c. 0  
d. Clothes & Shoes
- Which option is used with the SORT procedure to store only the BY variables and the observation numbers in temporary files?
  - NOTSORTED
  - GROUPFORMAT
  - TAGSORT
  - SORTEDBY=
- Which of the following is **not** an advantage of BY-group processing with an index that is based on the BY variables?
  - The data can be sequenced without using the SORT procedure.
  - There is no need to re-sort a data set when observations are modified or added.
  - It is generally more efficient than reading a sorted data set sequentially.
  - The data can be sequenced by different variables if multiple indexes are used.
- Which SORT procedure option compares all of the variable values for each observation to those for the previous observation that was written to the output data set?
  - NODUPKEY
  - NODUPRECS
  - EQUALS
  - NOEQUALS

5. What happens if the workspace that is required for completing a sort is less than or equal to the value that is specified in the SORTSIZE= system option or procedure option?
  - a. CPU time is increased.
  - b. I/O is increased.
  - c. The entire sort can take place in memory.
  - d. A temporary utility file is created in the **Work** directory or in a mainframe temporary area.

# Chapter 12

## Querying Data Efficiently

### Introduction

SAS provides a variety of techniques for querying data that enable you to create the results that you want in different ways. In this lesson, you learn to select the most efficient query techniques from those listed below, based on comparisons of resource usage.

| Task                                                                                 | Techniques                                                                                                                                                                         |
|--------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| selecting a subset                                                                   | <ul style="list-style-type: none"><li>WHERE statement that references a data set that has been indexed</li></ul>                                                                   |
| creating a detail report                                                             | <ul style="list-style-type: none"><li>PRINT procedure</li><li>SQL procedure</li></ul>                                                                                              |
| creating a summary report for one class variable or a combination of class variables | <ul style="list-style-type: none"><li>MEANS procedure (or SUMMARY procedure)</li><li>TABULATE procedure</li><li>REPORT procedure</li><li>SQL procedure</li><li>DATA step</li></ul> |



This lesson does not cover the Scalable Performance Data Engine (SPD Engine), which is a SAS 9.1 technology for [threaded processing](#). For details about using the SPD Engine to improve performance, see the SAS documentation.



This lesson does not include guided practices. Also, the data sets that are used in examples throughout this lesson are not included in the sample data for this course. Therefore, the programs in this lesson will not run in your SAS session.

### Objectives

In this lesson, you learn to

- identify the costs and benefits of using an index
- identify the factors that affect whether SAS uses an index for WHERE processing
- determine whether SAS is likely to use an index to process a particular WHERE expression
- identify the main features of compound optimization
- identify the effect of indexing and order of data on WHERE processing
- print centile information for a data file
- identify the relative efficiency of the PRINT procedure and the SQL procedure for creating detail reports
- identify the relative efficiency of five tools for summarizing data for one categorical variable
- identify the relative efficiency of three ways of using the MEANS procedure to summarize data for selected combinations of categorical variables.

## Using an Index for Efficient WHERE Processing

When processing a WHERE expression, SAS determines which of the following access methods is likely to be most efficient:

### Sequential access

SAS Data Set

|          |           |   |    |
|----------|-----------|---|----|
| Anderson | 09JAN2000 | X | 34 |
| Baker    | 14OCT2001 | X | 54 |
| Davis    | 30MAR2000 | Y | 49 |
| Edwards  | 28JUN2002 | X | 52 |
| Smith    | 15JAN2000 | Y | 62 |
| Yates    | 04AUG2002 | X | 59 |

SAS searches through all observations sequentially (in the order in which they are stored in the data file).

### Direct access

SAS Data Set

|          |           |   |    |
|----------|-----------|---|----|
| Anderson | 09JAN2000 | X | 34 |
| Baker    | 14OCT2001 | X | 54 |
| Davis    | 30MAR2000 | Y | 49 |
| Edwards  | 28JUN2002 | X | 52 |
| Smith    | 15JAN2000 | Y | 62 |
| Yates    | 04AUG2002 | X | 59 |

SAS uses an **index** to access specific observations directly. Using an index to process a WHERE expression is referred to as **optimizing** the WHERE expression.

Using an index to process a WHERE expression improves performance in some situations but not in others. For example, it is more efficient to use an index to select a small subset than a large subset. In addition, an index conserves some resources at the expense of others.

By deciding whether to create an index, you also play a role in determining which access method SAS can use. When your program contains a WHERE expression, you should determine which access method is likely to be more efficient. If direct access is likely to be more efficient, you can make sure that an index is available by creating a new index or by maintaining an existing index.

To help you make a more effective decision about whether to create an index, this topic and the next few topics provide you with a closer look at the following:

- steps that SAS performs for sequential access and direct access
- benefits and costs of index usage
- steps that SAS performs to determine which access method is most efficient
- factors affecting resource usage for indexed access
- guidelines for deciding whether to create, use, and maintain an index.



You should already know how to create and maintain indexes by using the INDEX= data set option in the DATA statement, the DATASETS procedure, and the SQL procedure. To review these SAS elements, see the lessons **Creating and Managing Indexes Using PROC SQL** and **Creating Samples and Indexes**.



SAS can also use an index to process a BY statement. BY processing enables you to process observations in a specific order according to the values of one or more variables that are specified in a BY statement. Indexing a data file enables you to use a BY statement without sorting the data file. When you specify a BY statement, SAS checks the value of the Sorted flag. If the Sorted flag is set to NO, then SAS looks for an appropriate index. If an appropriate index

exists, the software automatically retrieves the observations from the data file in indexed order. Using an index to process a BY statement might not always be more efficient than simply sorting the data file. Therefore, using an index for a BY statement is generally for convenience, not for performance.

## Accessing Data Sequentially

When accessing observations sequentially, SAS must search through all observations in the order in which they are stored in the data file.

### Example

Suppose you want to create a new data set, **Company.D02jul2000**, that contains a subset of observations from the data set **Company.Dates**. The following DATA step uses a WHERE statement to select all observations in which the value of Date\_ID is *02JUL2000*:

```
data company.d02jul2000;
  set company.dates;
  where date_id='02JUL2000'd;
run;
```

The data set **Company.Dates** does **not** have an index defined on the variable Date\_ID, so SAS must use **sequential access** to process the WHERE statement.

The following steps shows what happens when SAS uses sequential access to process a WHERE statement in a query.

1. The following DATA step creates a new data set, **Company.D02jul2000**, that contains all observations from the data set **Company.Dates** in which the value of Date\_ID is *02JUL2000*:

```
data company.d02jul2000;
  set company.dates;
  where date_id='02JUL2000'd;
run;
```

2. SAS loads **one** page of the data set **Company.Dates** into each input page buffer that has been allocated.
3. SAS then reads the observations in the input page buffer(s) sequentially until it finds an observation that meets the WHERE criterion (a qualified observation).
4. SAS loads the qualified observation into the program data vector.
5. SAS copies the observation in the program data vector into an output buffer.
6. SAS continues to read observations sequentially and to load qualified observations until an output buffer is filled.
7. When an output buffer is filled, SAS writes the data in the output buffer to disk as a page of the data set **Company.D02jul2000**.

## Accessing Data Directly

When using an index for WHERE processing, SAS goes straight to each observation that contains the value without having to read every observation in the data set.

### Example

Suppose you have defined an index on the variable Date\_ID in the **Company.Dates** data set that was shown in the previous example. This time, when you submit the following DATA step, SAS **uses the index** to process the WHERE statement:

```
data company.d02jul2000;
  set company.dates;
  where date_id='02JUL2000'd;
run;
```

The process of retrieving data via an index (direct access) is more complicated than sequentially processing data, so direct access requires more CPU time per observation retrieved than sequential access. However, for a small subset, using an index can decrease the number of pages that SAS has to load into input buffers, which reduces the number of I/O operations.

The following steps shows what happens when SAS uses direct access to process a WHERE statement in a query.

1. The following DATA step creates a new data set, **Company.D02jul2000**, that contains all observations from the data set **Company.Dates** in which the value of Date\_ID is 02JUL2000:

```
data company.d02jul2000;
  set company.dates;
  where date_id='02JUL2000'd;
run;
```

2. The index is stored on disk and is loaded into a buffer. SAS performs a binary search on the index and positions the index at the first entry that contains a qualified value.
3. SAS finds the page in the data set **Company.Dates** that contains the record identifier (RID) that is associated with the qualified value in the index. Then SAS loads that page into an input page buffer.
4. SAS directly accesses the observation that is specified by the record identifier, then loads the qualified observation into the program data vector.
5. SAS copies the observation in the program data vector into an output buffer.
6. For any additional record identifiers that are associated with the value in the index, SAS continues to load only the necessary data set pages into an input page buffer (if the needed page is not already in the buffer) and to load qualified observations.
7. When an output buffer is filled, SAS writes the data in the output buffer to disk as a page of the data set **Company.D02jul2000**.

What might happen when SAS uses an index instead of sequential access to process a WHERE expression?

**Q1.**

SAS might load fewer pages from the data set on disk into the input buffer.

SAS might load fewer observations into the program data vector.

SAS might load fewer pages from the program data vector into the output buffer.

## Benefits and Costs of Using an Index

As the preceding examples show, both benefits and costs are associated with using an index. Weighing these benefits and costs is an important part of deciding whether using an index is efficient.

The main **benefits** of using an index include the following:

- provides fast access to a small subset of observations
- returns values in sorted order
- can enforce uniqueness.

The main **costs** of using an index include the following:

- requires extra CPU cycles and I/O operations for creating and maintaining an index
- requires increased CPU time and I/O activity for reading the data
- requires extra disk space for storing the index file
- requires extra memory for loading index pages and extra code for using the index.



SAS requires additional buffers when an index file is used. When a data file is opened, SAS opens the index file, but not the indexes. Buffers are not required unless SAS uses an index, but SAS allocates the buffers to prepare for using the index. The number of levels of an index determines the number of buffers that are allocated. The maximum number of buffers is three for data files that are open for input; the maximum number is four for data that is open for update. These buffers can be used for other processing if they are not used for indexes.

## How SAS Selects an Access Method

When SAS processes a WHERE expression, it first determines whether to use direct access or sequential access by performing the following steps:

1. identifies available indexes
2. identifies conditions that can be optimized
3. estimates the number of observations that qualify
4. compares probable resource usage for both methods.

In the next few topics, each step of this process is explained in detail.

### Identifying Available Indexes

The first step for SAS is to **determine whether there are any existing indexes** that might be used to process the WHERE expression. Specifically, SAS checks the variable in each condition in the WHERE expression to determine whether the variable is a **key variable** in an index.



SAS can use either a simple index or a composite index to optimize a WHERE expression. To be considered for use in optimizing a single WHERE condition, one of the following requirements must be met:

- the variable in the WHERE condition is the **key variable** in a **simple** index
- the variable in the WHERE condition is the **first key variable** in a **composite** index.

SAS identifies all indexes that are defined on any variable in the WHERE expression. However, no matter how many indexes are available, **SAS can use only one index to process a WHERE expression**. So, if multiple indexes are available, SAS must choose between them.

When SAS looks for available indexes, there are three possible outcomes:

| If ...                                                                                                                         | Then ...                                                                                                                                                                                                                                    |
|--------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| there is <b>no index</b> defined on any variables in the WHERE expression                                                      | SAS <b>does not continue</b> with the decision process. SAS must use sequential access to process the WHERE expression.                                                                                                                     |
| there is <b>one available index</b> that is defined on one or more variables in the WHERE expression                           | SAS <b>continues</b> with the decision process and determines whether using the available index is more efficient than using sequential access.                                                                                             |
| there are <b>multiple available indexes</b> , each of which is defined on one or more of the variables in the WHERE expression | SAS <b>continues</b> with the decision process. SAS must choose between the available indexes in the next few steps. SAS tries to select the index that satisfies the most conditions and that selects the smallest subset of observations. |



If a program specifies both a WHERE expression and a BY statement, SAS looks for one index that satisfies conditions for both. If such an index is not found, the BY statement takes precedence so that SAS can ensure that the data is returned in sorted order. With a BY statement, SAS cannot use an index to optimize a WHERE expression if the optimization invalidates the BY order.

### Example: Identifying One Available Index

Suppose you submit a program that contains the following WHERE statement, and suppose that the data set has one index, as shown below:

| WHERE Statement                          | Available Index                                                                       |
|------------------------------------------|---------------------------------------------------------------------------------------|
| where <b>delivery_date='02jul2000'd;</b> | <ul style="list-style-type: none"><li>simple index defined on Delivery_Date</li></ul> |

This WHERE expression has one condition, and the variable in that condition (`Delivery_Date`) is the key variable in the simple index. If all other requirements for optimization are met in later steps, then SAS can use this index to optimize the WHERE expression.

Likewise, if the only available index is a composite index in which `Delivery_Date` is the first key variable, then SAS can use the index if all other requirements for optimization are met.

Even if a WHERE statement has multiple conditions, SAS can use either a simple index or a composite index to optimize just one of the conditions. For example, suppose your program contains a WHERE statement that has two conditions, and suppose that the data set has one index, as shown below:

| WHERE Statement                                                      | Available Index                                                                       |
|----------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| where <b>order_date='01jan2000'd and delivery_date='02jul2000'd;</b> | <ul style="list-style-type: none"><li>simple index defined on Delivery_Date</li></ul> |

Assuming that all other requirements for optimization are met, SAS can use this index to optimize the second condition in this WHERE expression.

### Example: Identifying Multiple Available Indexes

Suppose your program contains a WHERE statement with two conditions, and suppose that each condition references a key variable in a different index, as shown below:

| WHERE Statement                                                                 | Available Indexes                                                                                                                    |
|---------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| where <b>order_date='01jan2000'd</b> and<br><b>delivery_date='02jul2000'd';</b> | <ul style="list-style-type: none"> <li>simple index defined on Order_Date</li> <li>simple index defined on Delivery_Date.</li> </ul> |

Although two indexes are available, SAS can use only one index to optimize a WHERE statement. In a later step of the process, SAS will try to select the index that satisfies the most conditions and that selects the smallest subset of observations.

## Compound Optimization

SAS usually uses an index to process just one condition, no matter how many conditions and variables a WHERE expression contains. However, in a process called **compound optimization**, SAS can use a composite index to optimize multiple conditions on multiple variables, which are joined with a logical operator such as AND. Constructing your WHERE expression to take advantage of multiple key variables in a single index can greatly improve performance.

In order for compound optimization to occur, at least the first two key variables in the composite index must be used in the WHERE conditions. Later in this lesson, you will learn about other requirements that must be met in order for compound optimization to occur.

 The WHERE expression can also contain non-indexed variables, and the key variables and non-indexed variables can appear in any order in the expression.

### Example: Composite Index That Can Be Used to Optimize Multiple Conditions

Suppose your program contains a WHERE statement that has two conditions, and suppose that each condition references one of the first two key variables in a composite index:

| WHERE statement                                                                 | Available Index                                                                                                                                                                                                             |
|---------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| where <b>order_date='01jan2000'd</b> and<br><b>delivery_date='02jul2000'd';</b> | composite index defined on the following variables: <ul style="list-style-type: none"> <li>Order_Date (first key variable)</li> <li>Delivery_Date (second key variable)</li> <li>Product_ID (third key variable)</li> </ul> |

Because the two variables that are referenced in the WHERE expression are the first two key variables in the composite index, SAS can use the composite index for compound optimization if the WHERE conditions meet all other requirements for optimization.

### Example: Composite Index That Can Be Used to Optimize One Condition

The following WHERE statement also contains two conditions, and each condition references one of the variables in the composite index:

| WHERE statement                                                               | Available Index                                                                                                                                                                                                             |
|-------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| where <b>order_date='01jan2000'd</b> and<br><b>product_id='220101400106';</b> | composite index defined on the following variables: <ul style="list-style-type: none"> <li>Order_Date (first key variable)</li> <li>Delivery_Date (second key variable)</li> <li>Product_ID (third key variable)</li> </ul> |

As in the previous WHERE statement, Order\_Date is the first key variable in the index. However, in this situation, the composite index can be used to optimize only the first condition. The second condition references the third key variable, Product\_ID, but the WHERE expression does not reference the second key variable, Delivery\_Date. Without a reference to both the first and second key variables, compound optimization cannot occur.

### Example: Composite Index That Cannot Be Used for Optimizing

Now suppose your program contains a WHERE statement that references only the second and third key variables in the composite index, as shown below:

| WHERE statement                                                     | Available Index                                                                                                                                                                                                                      |
|---------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| where delivery_date='02jul2000'd' and<br>product_id='220101400106'; | composite index defined on the following variables:<br><ul style="list-style-type: none"> <li>• Order_Date (first key variable)</li> <li>• Delivery_Date (second key variable)</li> <li>• Product_ID (third key variable)</li> </ul> |

In this situation, SAS **cannot** use the index for optimization at all because the WHERE statement does not reference the first key variable.

Suppose you want to select a subset of observations in the data set **Company.Organization**. The data set has a composite index on the following variables:

- Job\_Title (the first key variable)
- Employee\_Hire\_Date (the second key variable).

For which of the following WHERE statements will SAS consider using the index?

**Q.2.**

- where employee\_hire\_date eq '01may2002'd;
- where job\_title eq 'Security Manager' and employee\_hire\_date eq '01may2002'd;
- where department eq 'Administration' and employee\_hire\_date eq '01may2002'd;
- all of the above

### Identifying Conditions That Can Be Optimized

In addition to containing key variables, WHERE conditions must meet other requirements in order to be candidates for optimization. SAS considers using an index only for WHERE conditions that contain **certain operators and functions**. Therefore, the next step for SAS is to consider the operators and functions in the conditions that contain key variables.

#### Requirements for Optimizing a Single WHERE Condition

SAS considers using an index for a WHERE condition that contains any of the following operators and functions:



For all of the following examples, assume that the data set has simple indexes on the variables Quarter, Date\_ID, and Region.

| Operator                                 | Example                                                    |
|------------------------------------------|------------------------------------------------------------|
| comparison operators and the IN operator | where quarter = '1998Q1';<br>where date_id < '03JUL2000'd; |

|                                                                                                                    |                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                 |
|--------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| comparison operators with NOT                                                                                      |                                                                                                                                                                                                                                                                                                                                                                                         | <code>where quarter in ('1998Q2','1998Q3');<br/>where quarter ne '1999Q1';<br/>where quarter not in ('1999Q1','1999Q4');</code> |
| comparison operators with the colon modifier                                                                       | <p> You can add a colon modifier (:) to any comparison operator to compare only a specified prefix of a character string.</p> <p> The colon modifier cannot be used with PROC SQL; use the LIKE operator instead.</p> | <code>where quarter =: '1998';</code>                                                                                           |
| CONTAINS operator                                                                                                  |                                                                                                                                                                                                                                                                                                                                                                                         | <code>where quarter contains 'Q4';</code>                                                                                       |
| fully-bounded range conditions that specify both an upper and lower limit, which includes the BETWEEN-AND operator |                                                                                                                                                                                                                                                                                                                                                                                         | <code>where '01Jan1999'd &lt; date_id &lt; '31Dec1999'd;<br/>where date_id between '01Jan1999'd<br/>and '31Dec1999'd</code>     |
| pattern-matching operator LIKE<br>IS NULL or IS MISSING operator                                                   |                                                                                                                                                                                                                                                                                                                                                                                         | <code>where quarter like '%Q%';<br/>where quarter is null;<br/>where quarter is missing;</code>                                 |

| Function                                                                                                                                                                                                                                                                                                                                             | Example                                                                                 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| TRIM function<br>SUBSTR function in the form of<br><b>WHERE SUBSTR(variable,position,length)=string;</b><br>with these conditions: <ul style="list-style-type: none"><li>• <i>position</i> = 1</li><li>• <i>length</i> is less than or equal to the length of <i>variable</i></li><li>• <i>length</i> is equal to the length of the string</li></ul> | <code>where trim(region) = 'Queensland';<br/>where substr(quarter,1,4) = '1998';</code> |



Most but not all of the requirements listed above also apply to compound optimization. Requirements for compound optimization are covered later in this topic.

## WHERE Conditions That Cannot Be Optimized

SAS does **not** use an index to process a WHERE condition that contains any of the elements listed below:



For all of the following examples, assume that the data set has simple indexes on the variables Date\_ID, Quarter, and Quantity.

| Element in WHERE Condition             | Example                                |
|----------------------------------------|----------------------------------------|
| any function other than TRIM or SUBSTR | <code>where weekday(date_id)=2;</code> |

|                                                                                    |                                |
|------------------------------------------------------------------------------------|--------------------------------|
| a SUBSTR function that searches a string beginning at any position after the first | where substr(quarter,6,1)='1'; |
| the sounds-like operator (=*)                                                      | where quarter=*'1900Q0';       |
| arithmetic operators                                                               | where quantity=quantity+1;     |
| a variable-to-variable condition                                                   | where quantity gt threshold;   |

|      |                                                                                                                                                                                                                                                                                                                                                                               |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Q.3. | <p>Which of the following WHERE statements contains conditions that <b>can</b> be optimized? (Assume that simple indexes are defined on the variables Lastname and Quantity.)</p> <p> <b>where lastname= '*'Smith';</b><br/> <b>where lowercase(lastname)= 'smith';</b><br/> <b>where quantity ne '1';</b><br/>         None of these statements can be optimized.       </p> |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### Requirements for Compound Optimization

Most of the same operators that are acceptable for optimizing a single condition are also acceptable for compound optimization. However, compound optimization has special requirements for the operators that appear in the WHERE expression:

- The WHERE conditions must be connected by using either the AND operator or, if all conditions refer to the same variable, the OR operator.
- At least one of the WHERE conditions that contains a key variable must contain the EQ or IN operator.

Also, SAS **cannot** perform compound optimization for WHERE conditions that include any of the following:

- the CONTAINS operator
- the pattern-matching operators LIKE and NOT LIKE
- the IS NULL and IS MISSING operators
- any functions.

### Example: Compound Optimization

Suppose your program contains the following WHERE statement, which selects all people whose name is John Smith. The WHERE statement contains two conditions, each of which references a different variable:

```
where lastname eq 'Smith' and
      frstname eq 'John';
```

Suppose `Lastname` is the first key variable and `Frstname` is the second key variable in a compound index. This WHERE statement meets all requirements for compound optimization:

- The WHERE expression references at least the first two key variables in one composite index.
- The two WHERE conditions are connected by the AND operator.
- At least one of the conditions contains the EQ operator.

If the two conditions in the WHERE statement are reversed, as shown below, the statement still meets all requirements for compound optimization. The order in which the key variables appear does not matter.

```
where frstname eq 'John' and  
      lastname eq 'Smith';
```

Now suppose that the conditions in the WHERE statement are joined by the operator OR instead of AND:

```
where frstname eq 'John' or  
      lastname eq 'Smith';
```

These conditions **cannot** be optimized because they are joined by OR but they do not reference the same variable.

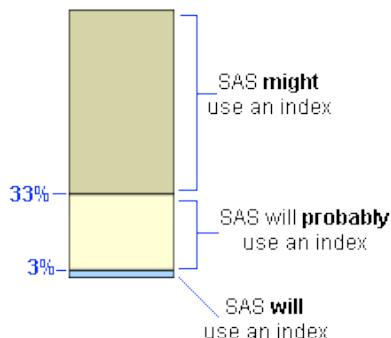
Suppose you are working with the data file **Company.Orders**, which contains the composite index **Shipped** on the variables Product\_ID and Delivery\_Date. Which of the following WHERE statements **cannot** be optimized?

**Q.4.**

```
where product_id = 210100100030 and delivery_date < '31dec2000'd;  
where product_id in (210100100030,210100100027) or delivery_date < '31dec2000'd;  
where product_id = 210100200000 and delivery_date < '31dec2000'd and quantity > 1;  
where quantity > 1 and delivery_date < '31dec2000'd and product_id = 210100100030;
```

## Estimating the Number of Observations

It is more efficient to use indexed access for a small subset and to use sequential access for a large subset. Therefore, after identifying any available indexes and evaluating the conditions in the WHERE expression, SAS estimates the number of observations that will be qualified by the index. Whether or not SAS uses an index depends on the percentage of observations that are qualified (the size of the subset relative to the size of the data set), as shown below:



- If the subset is **less than 3%** of the data set, direct access is almost certainly more efficient than sequential access, and SAS **will** use an index. In this situation, SAS does **not** go on to compare probable resource usage.
- If the subset is **between 3% and 33%** of the data set, direct access is likely to be more efficient than sequential access, and SAS will **probably** use an index.
- If the subset is **greater than 33%** of the data set, it is less likely that direct access is more efficient than sequential access, and SAS **might or might not** use an index.

When multiple indexes exist, SAS selects the one that appears to produce the **fewest** qualified observations (the **smallest subset**). SAS does this even when each index returns a subset that is less than 3% of the data set.

## **Printing Centile Information**

To help SAS estimate the number of observations that would be selected by a WHERE expression, each index stores 21 statistics called **cumulative percentiles**, or **centiles**. Centiles provide information about the distribution of values for the indexed variable.

Understanding the distribution of values in a data set can help you improve the efficiency of WHERE processing in your programs. You can print centile information for an indexed data file by specifying the CENTILES option in either of these places:

- the CONTENTS procedure
  - the CONTENTS statement in the DATASETS procedure.

**PROC CONTENTS** <options>;  
**RUN**;

**PROC DATASETS** <options>;  
    **CONTENTS** <options>;  
**QUIT**;

## Example

The following SAS program prints centile information for the data set **Company.Organization**:

```
proc contents data=company.organization centiles;  
run;
```

Partial output from this program is shown below. As indicated on the left, an index is defined on the variable Employee\_ID. The 21 centile values are listed on the right.

## Partial PROC CONTENTS Output

| Alphabetic List of Indexes and Attributes |       |               |             |                 |                        |                    |           |
|-------------------------------------------|-------|---------------|-------------|-----------------|------------------------|--------------------|-----------|
| #                                         | Index | Unique Option | Owned by IC | Update Centiles | Current Update Percent | # of Unique Values | Variables |
|                                           |       |               |             |                 |                        | 120729             |           |
|                                           |       |               |             |                 |                        | 120782             |           |
|                                           |       |               |             |                 |                        | 120834             |           |
|                                           |       |               |             |                 |                        | 120887             |           |
|                                           |       |               |             |                 |                        | 120939             |           |
|                                           |       |               |             |                 |                        | 120992             |           |
|                                           |       |               |             |                 |                        | 121044             |           |
|                                           |       |               |             |                 |                        | 121097             |           |
|                                           |       |               |             |                 |                        | 99999999           |           |

The 21 centile values consist of the following:

| Position in List | Value Shown in Output Above | Description                                                                                                                                                                       |
|------------------|-----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 (first)        | 120101                      | the <b>minimum value</b> of the indexed variable ( <b>0%</b> of values are lower than this value)                                                                                 |
| 2-20             | 120152 - 121097             | each value is greater than or equal to all other values in one of the 19 percentiles that range from the bottom <b>5%</b> to the bottom <b>95%</b> of values, in increments of 5% |
| 21 (last)        | 99999999                    | the <b>maximum value</b> of the indexed variable ( <b>100%</b> of values are lower than or equal to this value)                                                                   |



For information about updating and refreshing centiles for a data file, see the SAS documentation.

Suppose you have submitted the following program to select a subset of data from the large data set **Company.Customers**:

```
data_null_;
  set company.customers;
  where customer_country = 'MK';
run;
```

**Q5.**

SAS estimates that about 20% of the observations in the data set are qualified. Assuming that an index is defined on the variable that the WHERE expression references, how will SAS decide to process the WHERE expression?

SAS will **definitely** use an index.

SAS will **definitely not** use an index.

SAS will **probably** use an index.

## Comparing Probable Resource Usage

Once SAS estimates the number of qualified observations and selects the index that qualifies the fewest observations, SAS must then determine whether it is faster (more efficient) to satisfy the WHERE

expression by using the index or by reading all of the observations sequentially. Specifically, SAS predicts how many **I/O operations** will be required in order to satisfy the WHERE expression for each of the access methods. Then it compares the two resource costs.



Remember, if SAS estimates that a subset contains fewer than 3% of the observations in the data set, SAS does not need to estimate resource usage. In this situation, SAS will use the index to process the WHERE statement.

## How SAS Compares Resource Usage

To compare resource usage, SAS performs the following steps:

1. SAS predicts how many I/O operations will be required if it uses the index to satisfy the WHERE expression. To do so, SAS positions the index at the first entry that contains a qualified value. In a buffer management simulation that takes into account the current number of available buffers, the RIDs (record identifiers) on that index page are processed, indicating how many I/Os will be required in order to read the observations in the data file.
2. SAS calculates the I/O cost of a sequential pass of the entire data file.
3. SAS compares the two resource costs and determines which access method has a lower cost.



If comparing resource costs results in a tie, SAS chooses the index.

## Factors That Affect I/O

Several factors affect the number of I/O operations that are required for WHERE processing, including the following:

- subset size relative to data set size
- number of pages in the data file
- order of the data
- cost to uncompress a compressed file for a sequential read.

These factors are discussed in more detail below.

### Subset Size Relative to Data Set Size

As explained earlier in this lesson, SAS is more likely to use an index to access a small subset of observations. The process of retrieving data with an index is inherently more complicated than sequentially processing the data. This is why using an index requires more I/O operations and CPU time when a large subset is read.

For small subsets, however, the benefit of reading only a few observations outweighs the cost of the complex processing. The smaller the subset, the larger the performance gains. Remember that SAS will use an index if the subset is less than 3% of the data set, and SAS will probably use an index if the subset is between 3% and 33% of the data set.

### Number of Pages in the Data File

For a small data file, sequential processing is often just as efficient as index processing. If the data file's page count is less than three pages, then sequential access is faster even if the subset is less than 3% of the entire data set.



The amount of data that can be transferred to one buffer in a single I/O operation is referred to as page size. To see how many pages are in a data file, use either the CONTENTS procedure or the CONTENTS statement in the DATASETS procedure. For more information about reporting the page size for a data file, see the chapter **Controlling Memory Usage**.

### Order of the Data

The order of the data (sort order) affects the number of I/O operations as described below:

| Order of the Data                                                     | Effect on I/O Operations                                                                                                                                                                                                                                     |
|-----------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| observations are <b>randomly</b> distributed throughout the data file | The observations are located on a larger number of data file pages. An I/O operation is required each time that SAS loads a page. Therefore, the <b>more random</b> the data in the data file, the <b>more I/O operations</b> are needed to use the index.   |
| observations are <b>sorted</b> on the indexed variable(s)             | The data is ordered more like the index (in ascending value order), and the observations will be located on fewer data file pages. Therefore, the <b>less random</b> the data in the data file, the <b>fewer I/O operations</b> are needed to use the index. |



In general, sorting the data set by the key variable before indexing will result in greater efficiency. The more ordered the data file is with respect to the key variable, the more efficient the use of the index. If the data file has more than one index, then sorting the data by the most frequently used key variable is most efficient. Sorting the data set results in more efficient WHERE processing even when SAS does not use an index. To learn more about sorting and efficiency, see the chapter **Selecting Efficient Sorting Strategies**.

### Cost to Uncompress a Compressed File for a Sequential Read

When SAS reads a compressed data file, SAS automatically uncompresses the observations as they are read into the program data vector. This requires additional CPU resources, but fewer I/O operations are required because there are fewer data set pages. When performing a **sequential** read of a compressed data file, SAS must uncompress **all** observations in the file. However, when using **direct** access, SAS must uncompress only the **qualified** observations. Therefore, the resource cost of uncompressing observations is greater for a sequential read than for direct access.



Compressing a file is a process that reduces the number of bytes that are required for representing each observation. By default, a SAS data file is not compressed. For more information about compressing files, see the chapter **Controlling Data Storage Space**.

### Other Factors That Affect Resource Usage

**Data type** and **length** are two other factors that can affect index efficiency. **Numeric** key variables typically result in more CPU usage than character key variables, because numeric variables must be converted to collatable formats (formats that can be sequenced) when values are read into the index or retrieved from the index. **Character** values are already intrinsically collatable, so they do not have to be converted. A page cannot contain as many values if the values are longer. When the values are longer, the index structure is not as efficient, which might lead to more I/O and search time.

**Q.6.** To decide whether it is faster to use direct access or sequential access, which of the following does SAS compare?

the estimated amount of CPU time that is needed to use an index, and the cost of accessing the data sequentially  
the estimated number of I/O operations that are needed to use an index on a sorted data set, and the cost of using an index on an unsorted data set  
the estimated number of I/O operations that are needed to use an index, and the cost of accessing the data sequentially

## Deciding Whether to Create an Index

In previous topics, you learned how SAS determines whether sequential access or direct access is likely to be most efficient for WHERE processing. You also learned about a variety of factors that you can assess to determine which access method is most efficient. Once you have made your determination, you can use the following guidelines to decide whether it is efficient to create an index.

### Guidelines for Deciding Whether to Create an Index

- Minimize the number of indexes to reduce disk storage and update costs. Create indexes only on variables that are often used in queries or (when data cannot be sorted) in BY-group processing.
- Create an index when you intend to retrieve a small subset of observations from a large data file.
- Do not create an index if the data file's page count is less than three pages. It is faster to access the data sequentially.
- Create indexes on variables that are discriminating. Discriminating variables have many different values that precisely identify observations. A WHERE expression that subsets based on a discriminating variable results in a smaller subset than a WHERE expression that references a non-discriminating variable (a variable that has only a few values).
- To reduce the number of I/O operations that are performed when you create an index, first sort the data by the key variable. Then, to improve performance, maintain the data file in sorted order by the key variable.



If you choose not to use an index and the data set is large, it is still more efficient to sort the data set on the variable(s) that are specified in the WHERE statement.

- Consider how often your applications use an index. An index must be used often in order to compensate for the resources that are used in creating and maintaining it.
- Consider the cost of an index for a data file that is frequently changed.
- When you create an index to process a WHERE expression, do not try to create one index that is used to satisfy all queries.

Let's look at three sample queries to see how you can apply the guidelines that are listed on the previous page. These queries illustrate the effect of one factor—the size of the subset relative to the size of the data set—on the choice of an access method. For each query, you will learn:

- which access method SAS is likely to select
- whether you could improve performance by creating an index.

### Example: Selecting Subsets of Various Sizes from Data Sets of Various Sizes

Suppose you are working with the following two data sets, each of which contains information about a company's orders:

| Data Set Name | Pages | Observations |
|---------------|-------|--------------|
|---------------|-------|--------------|

|                             |         |            |
|-----------------------------|---------|------------|
| <b>Company.Orders_large</b> | 285,500 | 19,033,380 |
| <b>Company.Orders_small</b> | 2       | 140        |

You want to create queries to generate three subset detail reports, one for each of the following types of subsets:

- **small** subset from a **large** data set
- **large** subset from a **large** data set
- **small** subset from a **small** data set.

In all three queries, the WHERE expression specifies the variable Order\_Date. You know that this variable will be used frequently in queries for the company, and that it is a discriminating variable. According to the guidelines on the previous page, these are both criteria for creating an index on the variable. However, there is currently no index defined on this variable in either data set.

### Query 1: Small Subset from a Large Data Set

The first report that you want to generate shows all orders in **Company.Orders\_large** that were made on January 10, 1998. Your query is shown below, along with the subset size that you have estimated:

| Query                                                                                     | Subset Size                                                           |
|-------------------------------------------------------------------------------------------|-----------------------------------------------------------------------|
| <pre>data _null_;   set company.orders_large;   where order_date='10JAN1998'd; run;</pre> | <b>2232</b> observations (out of 19,033,380) = < .02% of the data set |

Because the subset is less than 3% of the entire data set, using an index on Order\_Date should be more efficient than using sequential access. SAS will use an index for WHERE processing, if an index is available. To improve performance, you should create an index on Order\_Date before running this program.

### Query 2: Large Subset from a Large Data Set

The second report shows all orders in **Company.Orders\_large** that were made before January 1, 1998. Your query and the estimated subset size are shown below:

| Query                                                                                        | Subset Size                                                                                   |
|----------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| <pre>data _null_;   set company.orders_large;   where order_date&lt;'01JAN2000'd; run;</pre> | <b>12,752,365</b> observations (out of 19,033,380) = approximately <b>67%</b> of the data set |

Because the subset is more than 33% of the entire data set, using the index is probably **less** efficient than using sequential access. SAS will probably not use the index for WHERE processing.

### Query 3: Small Subset from a Small Data Set

The third report shows all orders in the smaller data set **Company.Orders\_small** that were made on June 30, 1998. Your query and the estimated subset size are shown below:

| Query                                                                                     | Subset Size                                         |
|-------------------------------------------------------------------------------------------|-----------------------------------------------------|
| <pre>data _null_;   set company.orders_small;   where order_date='30JUN1998'd; run;</pre> | 2 observations (out of 140) = < 2 % of the data set |

Because the subset is less than 3% of the entire data set, SAS will use the index for WHERE processing. However, the data file's page count is less than three pages, so it is more efficient to use sequential access. In this situation, it is best not to create an index.

You have written a query that contains a WHERE expression that references one variable. In which of the following situations is it most likely **not** efficient to create and maintain an index on the referenced variable?

**Q7.**

The WHERE expression references the variable EmplID, which has a unique value for each observation.

The data set is very large and the subset is small (< 1%).

The WHERE expression references the variable LastName, which has many different values. The data set is sorted by LastName.

The WHERE expression references the variable Gender, which has two possible values.

### Using the Options **IDXWHERE=** and **IDXNAME=** to Control Index Usage

In most situations, it is best to let SAS determine whether or not to use an index for WHERE processing. However, sometimes you might want to control whether or not SAS uses an existing index. For example, if you know that your query will select a large subset and that indexed access will therefore not be efficient, you can tell SAS to ignore any index and to satisfy the conditions of the WHERE expression with a sequential search of the data set. Or, if your query will select a small subset and there are multiple available indexes, you can make sure that SAS uses a particular index to process your WHERE statement. Finally, you might want to force SAS to use (or not use) an index when you are benchmarking.

You should be familiar with the data set options **IDXWHERE=** and **IDXNAME=**, which you can use to control index usage:

| Option           | Action                                                                                                                                                                                                                                                                                                                      |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>IDXWHERE=</b> | specifies whether or not SAS should use an index to process the WHERE expression, no matter which access method SAS estimates is faster.<br><br> You cannot use IDXWHERE= to override the use of an index for processing a BY statement. |
| <b>IDXNAME=</b>  | causes SAS to use a specific index.                                                                                                                                                                                                                                                                                         |



You can use **either** **IDXWHERE=** or **IDXNAME=**, but not both at the same time, to control index usage.



For more information about the **IDXWHERE=** and **IDXNAME=** data set options, see the chapter **Creating and Managing Indexes Using PROC SQL**.

## Specifying MSGLEVEL=I to Determine Whether SAS Is Using an Index

To determine whether SAS is using an index to process a WHERE expression, you can specify *I* as the value of the MSGLEVEL= system option. Using MSGLEVEL=I causes SAS to display information about index usage in the SAS log.



To make the most efficient use of resources, use MSGLEVEL=I only for debugging and for verifying index usage.



For more information about the MSGLEVEL= system option, see the chapter **Creating Samples and Indexes** or the chapter **Creating and Managing Indexes Using PROC SQL**.

### Example: Using IDXWHERE=NO to Prevent Index Usage

Suppose you write the following query, which lists all employees who work in the Sales department of a company:

```
proc print data=company.organization;
  where department='Sales';
run;
```

Now suppose an index is defined on the variable Department in the data set **Company.Organization**. You know that Department has the value *Sales* in 65% of the observations, so it is **not** efficient for SAS to use an index for WHERE processing. To ensure that SAS does not use an index, specify IDXWHERE=NO after the data set name. At the beginning of the program, you can also add an OPTIONS statement that specifies MSGLEVEL=I to display a message about index usage in the SAS log. The revised program is shown below:

```
options msglevel=i;
proc print data=company.organization (idxwhere=no);
  where department='Sales';
run;
```

When you run this program, the SAS log indicates that the index was **not** used for processing:

#### SAS Log

**INFO:** Data set option (IDXWHERE=NO) forced a sequential pass of the data rather than use of an index for where-clause processing.

## Comparing Procedures That Produce Detail Reports

When you want to use a query to produce a detail report, you can choose between the **PRINT procedure** and the **SQL procedure**:

| Procedure         | Description                                                                                                                                                                               |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>PROC PRINT</b> | <ul style="list-style-type: none"><li>• produces data listings quickly</li><li>• can supply titles, footnotes, and column sums</li></ul>                                                  |
| <b>PROC SQL</b>   | <ul style="list-style-type: none"><li>• combines SQL and SAS features such as formats</li><li>• can manipulate data and create a SAS data set in the same step that creates the</li></ul> |

|  |                                                                                                                                                                               |
|--|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <ul style="list-style-type: none"> <li>report</li> <li>• can produce column and row statistics</li> <li>• does not offer as much control over output as PROC PRINT</li> </ul> |
|--|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

To perform a particular task, a single-purpose tool like PROC PRINT generally uses fewer computer resources than a multi-purpose tool like PROC SQL. However, PROC SQL often requires fewer and shorter statements to achieve the results that you want.

To illustrate the differences in resource usage between PROC PRINT and PROC SQL, let's look at some sample queries.

### Example: Using PROC PRINT and PROC SQL to Create Detail Reports

Suppose you are working with the data set **Company.Products** and that you want to generate four types of detail reports:

- simple detail report
- subset detail report
- sorted detail report
- sorted subset detail report.

For the first three reports, the PROC PRINT program is likely to use fewer resources than the PROC SQL program. For the last report, the resource usage for the two programs is likely to be about the same.

#### Report 1: Simple Detail Report

The simple detail report lists the product ID, product name, and supplier name for all products. The PROC PRINT program and PROC SQL program for producing this report are shown below:

| PROC PRINT                                                                                         | PROC SQL                                                                                               |
|----------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <pre>proc print data=company.products;   var product_id product_name     supplier_name; run;</pre> | <pre>proc sql;   select product_id product_name     supplier_name   from company.products; quit;</pre> |

In this situation, the PROC PRINT program is likely to use fewer CPU and memory resources than the PROC SQL program. The I/O resource usage should be approximately the same.

#### Report 2: Subset Detail Report

The subset detail report lists the product ID, product name, and supplier name for all products that come from Sweden (SE). The PROC PRINT program and PROC SQL program for producing this report are shown below:

| PROC PRINT                                                                                                                   | PROC SQL                                                                                                                    |
|------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <pre>proc print data=company.products;   var product_id product_name     supplier_name;   where supplier_country='SE';</pre> | <pre>proc sql;   select product_id product_name     supplier_name   from company.products   where supplier_name='SE';</pre> |

```
run;
```

```
      where supplier_country='SE';  
      quit;
```

Both steps use WHERE processing to subset the data. In this situation, the PROC PRINT program is likely to use fewer CPU and memory resources than the PROC SQL program. The I/O resource usage should be approximately the same.

### Report 3: Sorted Detail Report

The sorted detail report lists the product ID, product name, and supplier name for all products, with observations sorted by the supplier country. The PROC PRINT program and PROC SQL program for producing this report are shown below:

| PROC PRINT                                                                                                                                                                                       | PROC SQL                                                                                                                                               |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>proc sort data=company.products<br/>  out=product;<br/>  by supplier_country;<br/>run;<br/>proc print data=product;<br/>  var product_id product_name<br/>    supplier_name;<br/>run;</pre> | <pre>proc sql;<br/>  select product_id product_name<br/>    supplier_name<br/>  from company.products<br/>  order by supplier_country;<br/>quit;</pre> |

To sort the data, a PROC SORT step has been added to the PROC PRINT program, and an ORDER BY clause has been added to the PROC SQL program. In this situation, the PROC PRINT program is likely to use fewer CPU and memory resources than the PROC SQL program. The I/O resource usage should be approximately the same.

### Report 4: Sorted Subset Detail Report

The sorted subset detail report lists the product ID, product name, and supplier name for all products that come from Sweden (SE), with observations sorted by the supplier name. The PROC PRINT program and PROC SQL program for producing this report are shown below:

| PROC PRINT                                                                                                                                                                                                                                                                                                   | PROC SQL                                                                                                                                                                              |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>proc sort data=company.products<br/>  (keep=Product_ID Product_Name<br/>   Supplier_Name Supplier_Country )<br/>  out=product;<br/>  where supplier_country='SE';<br/>  by supplier_name;<br/>run;<br/>proc print data=product;<br/>  var product_id product_name<br/>    supplier_name;<br/>run;</pre> | <pre>proc sql;<br/>  select product_id product_name<br/>    supplier_name<br/>  from company.products<br/>  where supplier_country='SE'<br/>  order by supplier_name;<br/>quit;</pre> |

To sort the data, a PROC SORT step has been added to the PROC PRINT program. The PROC SORT step uses the KEEP= option to subset the observations, which improve efficiency. The PROC SQL step uses an ORDER BY clause for sorting and a WHERE clause for subsetting. In this situation, the CPU and memory usage for the PROC PRINT program and the PROC SQL program are about the same.

You are writing a query that generates a simple detail report. Which procedure is likely to use less CPU time and memory?

**Q.8.**

PROC PRINT

PROC SQL

Neither tool; they have the same resource usage.

## Comparing Tools for Summarizing Data

SAS provides a variety of tools for summarizing data. These summarization tools generate similar but not identical output, and they vary in efficiency. This topic discusses the relative efficiency of the following summarization tools:



Throughout this topic, all references to the MEANS procedure apply also to the SUMMARY procedure.

| Tool                                        | Description                                                                                                                                                                                                                                                    |
|---------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>MEANS procedure or SUMMARY procedure</b> | <ul style="list-style-type: none"><li>computes descriptive statistics for numeric variables</li><li>can produce a printed report and create an output data set</li></ul>                                                                                       |
| <b>TABULATE procedure</b>                   | <ul style="list-style-type: none"><li>produces descriptive statistics in a tabular format</li><li>can produce multidimensional tables with descriptive statistics</li><li>can also create an output data set.</li></ul>                                        |
| <b>REPORT procedure</b>                     | <ul style="list-style-type: none"><li>combines features of the PRINT, MEANS, and TABULATE procedures with features of the DATA step in a single report-writing tool that can produce a variety of reports</li><li>can also create an output data set</li></ul> |
| <b>SQL procedure</b>                        | <ul style="list-style-type: none"><li>computes descriptive statistics for one or more SAS data sets or DBMS tables</li><li>can produce a printed report or create a SAS data set</li></ul>                                                                     |
| <b>DATA step</b>                            | <ul style="list-style-type: none"><li>can produce a printed report</li><li>can also create an output data set</li></ul>                                                                                                                                        |



You can also use the FREQ procedure and the UNIVARIATE procedure to generate summary data and create summary reports, but these procedures are not covered in this lesson. For more information about any of these summarization tools, see the SAS documentation.

You can use these tools to summarize data at the following levels:

| Level of Summarization | Tools            |
|------------------------|------------------|
| entire data set        | any of the above |
| one class variable     | any of the above |

|                                             |                                                                                                                                                                                     |
|---------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                             |  To group the data, PROC SQL uses the GROUP BY statement, and the DATA step uses the BY statement. |
| one or more combinations of class variables | <ul style="list-style-type: none"> <li>• PROC MEANS (or PROC SUMMARY)</li> <li>• PROC TABULATE</li> </ul>                                                                           |

### Comparing Resource Usage across Summarization Tools

When summarizing data for one or more class variables, the tools in each of the following groups are similar in resource usage:

- **PROC MEANS (or PROC SUMMARY), PROC REPORT, and PROC TABULATE**
- **PROC SQL and the DATA step with PROC SORT.**

However, the relative efficiency of the two groups of tools varies according to the shape of the data, as shown below:

| Shape of the Data                                                                                                                                                                        | Most Efficient Tools                                                                                                                                                                                                                                                                                          |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| a relatively <b>small number</b> of distinct combinations of values of the class variable—the summarized data contains <b>10% or fewer</b> of the observations in the original data set  | <ul style="list-style-type: none"> <li>• <b>PROC MEANS (or PROC SUMMARY)</b></li> <li>• <b>PROC REPORT</b></li> <li>• <b>PROC TABULATE</b></li> </ul>                                                                                                                                                         |
| a relatively <b>large number</b> of distinct combinations of values of the class variable—the summarized data contains <b>more than 10%</b> of the observations in the original data set | <ul style="list-style-type: none"> <li>• <b>PROC SQL</b></li> <li>• <b>DATA step and PROC SORT</b></li> </ul>  Using a GROUP BY statement in PROC SQL is similar in CPU usage to a PROC SORT step followed by a DATA step. |

### Comparative Example: Displaying Summary Statistics for One Class Variable

Suppose you want to summarize the data set **Retail.Orders** by calculating the average quantity of products sold for each order type (each value of the class variable **Order\_Type**). You can use the following techniques to produce the summary report:

1. PROC MEANS
2. PROC REPORT
3. PROC SORT and a DATA step
4. PROC SQL
5. PROC TABULATE.

The following programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for summarizing data for one class variable.

### Programming Techniques

## 1. PROC MEANS

This PROC MEANS step creates an output report that displays the mean quantity of products sold (the analysis variable Quantity) for each order type (the class variable Order\_Type).

```
proc means data=retail.orders  
  (keep=order_type quantity)  
  mean maxdec=2;  
  class order_type;  
  var quantity;  
  run;
```

## 2. PROC REPORT

This PROC REPORT step creates an output report that displays the mean quantity of products sold (the analysis variable Quantity) for each order type (the class variable Order\_Type).

```
proc report data=retail.orders  
  (keep=order_type quantity);  
  column order_type quantity;  
  define order_type / group width=13  
    'Order Type';  
  define quantity / mean format=5.2  
    'Average Quantity'  
    width=8;  
  run;
```

## 3. PROC SORT and a DATA Step

This program uses a PROC SORT step and a DATA step to create an output report. The PROC SORT step specifies the variables to be included in the report, sorts the observations by the values of the variable Order\_Type, and generates the temporary output data set **Orders**. The DATA step calculates the mean quantity of products sold (the analysis variable Quantity) for each order type (the class variable Order\_Type) and displays these values in the temporary output data set.

```
proc sort data=retail.orders  
  (keep=order_type quantity)  
  out=orders;  
  by order_type;  
run;  
data _null_;  
  set orders;  
  by order_type;  
  format average_order 5.2;  
  if first.order_type then do;  
    num=0;  
    sum=0;  
  end;  
  num+1;  
  sum+quantity;  
  if last.order_type then do;  
    average_order=sum / num;  
    file print ods=  
      (var=(order_type average_order));  
    put _ods_;  
  end;  
run;
```

## 4. PROC SQL

This PROC SQL step creates an output report that displays the mean quantity of products sold (the

analysis variable Quantity) for each order type (the class variable Order\_Type).

```
proc sql;
  select order_type,
    avg(quantity) label='Average Order'
      format=5.2
  from retail.orders
  group by order_type;
quit;
```

## 5. PROC TABULATE

This PROC TABULATE step creates an output report that displays the mean quantity of products sold (the analysis variable Quantity) for each order type (the class variable Order\_Type).

```
proc tabulate data=retail.orders
  (keep=order_type quantity)
  format=comma8.2;
  class order_type;
  var quantity;
  table order_type, quantity*mean;
run;
```

### General Recommendations

- When summarizing data for one class variable, assess the shape of the data to determine which summarization tools are most efficient to use:
- If there is a relatively **small** number of distinct combinations of values of the class variable (the summarized data contains 10% or fewer of the observations in the original data set), use one of the following tools: PROC MEANS (or PROC SUMMARY), PROC REPORT, or PROC TABULATE.
- If there is a relatively **large** number of distinct combinations of values of the class variable (the summarized data contains more than 10% of the observations in the original data set), use one of the following tools: PROC SQL or the DATA step.

You are writing a program to summarize data for one class variable, and there is a relatively large number of distinct combinations of values of the class variable. In this situation, which of the following summarization tools is most efficient?

**Q.9.**

- DATA step
- MEANS procedure
- TABULATE procedure
- REPORT procedure

### Using PROC MEANS to Display Summary Statistics for Combinations of Class Variables

To produce summary statistics for **combinations of class variables**, you can use PROC MEANS in the following ways. These techniques differ in resource usage.

| Combinations of Class Variables        | Technique                    | Example                                                   |
|----------------------------------------|------------------------------|-----------------------------------------------------------|
| all possible combinations:<br><b>c</b> | <b>basic</b> PROC MEANS step | proc means data=lib.dataset mean;<br>class <b>a b c</b> ; |

|                                                                                          |                                                            |                                                                                                                                                                                                                          |
|------------------------------------------------------------------------------------------|------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>b</b><br><b>b * c</b><br><b>a</b><br><b>a * c</b><br><b>a * b</b><br><b>a * b * c</b> |                                                            | <pre>var salary; output out=summary1       mean=average; run;</pre>                                                                                                                                                      |
| <b>specific combinations:</b><br><b>a * b and a * c</b>                                  | <b>TYPES statement</b> in PROC MEANS                       | <pre>proc means data=lib.dataset mean; class a b c; var salary; <b>types a*b a*c;</b> output out=summary2       mean=average; run;</pre>                                                                                 |
| <b>specific combinations:</b><br><b>a * b and a * c</b>                                  | <b>NWAY option</b> in multiple PROC MEANS steps            | <pre>proc means data=lib.dataset nway; class a b; var salary; output out=summary3a       mean=average; run; proc means data=lib.dataset nway; class a c; var salary; output out=summary3b       mean=average; run;</pre> |
| <b>specific combinations:</b><br><b>a * b and a * c</b>                                  | <b>WHERE= option</b> in the OUTPUT statement in PROC MEANS | <pre>proc means data=lib.dataset; class a b c; var salary; output out=summary4       (<b>where=(_type_in(5,3))</b>)       n=employees       mean=average; run;</pre>                                                     |

### Comparing Resource Usage across Three Techniques for Using PROC MEANS

The three techniques for summarizing data for **specific combinations** of class variables (all but the basic PROC MEANS step) differ in resource usage as follows:

- The **TYPES statement in a PROC MEANS step** uses the fewest resources.
- A program that contains the **NWAY option in multiple PROC MEANS steps** uses the most resources because SAS must read the data set separately for each PROC MEANS step.
- The **WHERE= data set option in a PROC MEANS step** uses more resources than the TYPES statement in PROC MEANS because SAS must calculate all possible combinations of class variables before subsetting. However, the WHERE= data set option in PROC MEANS uses fewer resources than the NWAY option in multiple PROC MEANS steps.

Let's take a closer look at how to use a basic PROC MEANS step and the three other techniques that are listed above.

### Using a Basic PROC MEANS Step to Combine All Class Variables

By default, PROC MEANS (or PROC SUMMARY) creates the following:

- an output **report** that groups data and displays summary statistics for the combination of **all the class variables**
- an output **data set** that groups data and displays summary statistics for **all possible combinations** of the  $n$  class variables (from 1-way to  $n$ -way), as well as for the entire data set.

### **Example: Displaying Summary Statistics for All Combinations of the Class Variables**

Suppose you want to calculate average employee salaries and to group results for the combination of the three class variables Employee\_Country, Department, and Employee\_Gender.

The following PROC MEANS program creates both a report and a SAS data set:

```
proc means data=company.organization mean;
  class employee_country department
    employee_gender;
  var salary;
  output out=summary mean=average;
run;
```

The report groups data and displays summary statistics for the combination of the three class variables.  
The output report is shown Page 100

The output data set groups data and displays summary statistics for both of the following:

- all possible combinations (1-way, 2-way, and 3-way) of the three class variables:
  - Employee\_Gender
  - Department
  - Department and Employee\_Gender
  - Employee\_Country
  - Employee\_Country and Employee\_Gender
  - Employee\_Country and Department
  - Employee\_Country and Department and Employee\_Gender
- the entire data set.

The output data set is shown Page 100

### **Understanding Types**

Each combination of class variables that is used to calculate and group statistics for PROC MEANS is called a **type**.

For example, the following basic PROC MEANS step specifies the three class variables a, b, and c:

```
proc means data=lib.dataset mean;
  class a b c;
  var salary;
  output out=summary1
    mean=average;
run;
```

This PROC MEANS step generates seven possible types (combinations of the three variables):

| Variables Combined | Dimension |
|--------------------|-----------|
| c                  | 1-way     |

|           |       |
|-----------|-------|
| b         | 1-way |
| b * c     | 2-way |
| a         | 1-way |
| a * c     | 2-way |
| a * b     | 2-way |
| a * b * c | 3-way |

SAS uses the **\_TYPE\_ variable** to specify the combination of class variables that PROC MEANS uses to calculate the statistics for each observation in the output data set. The **\_TYPE\_ variable** has a unique value for each combination. SAS always combines the class variables in a particular sequence, based on the order in which they are listed in the CLASS statement, in order to assign the **\_TYPE\_ values**. For example, for each of the seven types (seven possible combinations of three class variables) shown above, SAS assigns a value to **\_TYPE\_** as follows:

| <b>_TYPE_ Value</b> | <b>Description of Combination</b>                                   | <b>Variables Combined</b> | <b>Dimension</b> |
|---------------------|---------------------------------------------------------------------|---------------------------|------------------|
| 1                   | <b>rightmost variable only</b>                                      | c                         | 1-way            |
| 2                   | <b>middle variable only</b>                                         | b                         | 1-way            |
| 3                   | <b>rightmost variable and middle variable</b>                       | b * c                     | 2-way            |
| 4                   | <b>leftmost variable</b>                                            | a                         | 1-way            |
| 5                   | <b>leftmost variable and rightmost variable</b>                     | a * c                     | 2-way            |
| 6                   | <b>leftmost variable and middle variable</b>                        | a * b                     | 2-way            |
| 7                   | <b>rightmost variable and middle variable and leftmost variable</b> | a * b * c                 | 3-way            |

As the number of class variables increases, so does the number of types. However, the **highest \_TYPE\_ value** (7, in this example) always indicates the combination of **all** class variables.

SAS includes the **\_TYPE\_ variable** in the output data set that is generated by PROC MEANS. In the output from the basic PROC MEANS step that was shown on the previous page, you can see that the observations are listed in order of increasing values of the **\_TYPE\_ variable**: Page 100

Note that the first observation in the output data set has a **\_TYPE\_ value** of 0, which indicates that the statistics are generated for the **entire data set**.



SAS calculates the **\_TYPE\_ variable** even if no output data set is requested.

By default, the output **data set** that is generated by PROC MEANS contains a separate observation for each unique combination of values of the class variables for each type. Each unique combination of values within a type is called a **level** of that type. In the output data set linked above, for example, there are 17 levels for type 2 (17 observations that have a **\_TYPE\_ value** of 2).

The output **report** that is generated by the basic PROC MEANS step contains only the observations that represent a combination of **all** of the class variables (the observations for which **\_TYPE\_=7**). The **\_TYPE\_ variable** is not displayed in the report.

## Using the TYPES Statement in PROC MEANS to Combine Class Variables

You can use the TYPES statement in PROC MEANS to specify which combinations of the class variables are used for grouping data and for calculating statistics. The CLASS statement is required in order to use the TYPES statement.

#### General form, TYPES statement:

**TYPES** request(s);

where *request(s)* specifies the combination(s) of class variables that PROC MEANS uses to create the types. A request is composed of one of the following:

- one class variable name
- several class variable names separated by asterisks
- ( ) to request the combination of all variables (\_TYPE\_=0).

To request combinations of class variables more concisely, you can use a grouping syntax by placing parentheses around several variables and joining other variables or variable combinations. The following examples of TYPES statements illustrate the use of grouping syntax:

| Example with Grouping Syntax | Equivalent Example without Grouping Syntax |
|------------------------------|--------------------------------------------|
| types a*(b c);               | types a*b a*c;                             |
| types (a b)*(c d);           | types a*c a*d b*c b*d;                     |
| types (a b c)*d;             | types a*d b*d c*d;                         |
| types () a*(b c);            | types a*b*c a*b a*c;                       |

#### Example: Using the TYPES Statement in PROC MEANS

Suppose you want to calculate average employee salaries, as in the previous example. This time, you want to group results for the two combinations of class variables shown below:

- Employee\_Country and Department
- Employee\_Country and Employee\_Gender.

To do this, you can add a TYPES statement to the PROC MEANS step:

```
proc means data=company.organization mean;
  class employee_country department
    employee_gender;
  var salary;
  types employee_country*department
    employee_country*employee_gender;
  output out=summary mean=average;
run;
```

This PROC MEANS step generates both an output report and an output data set. The report, shown below, has a separate table for each of the two combinations that were specified in the TYPES statement: Page100

The output data set summarizes and reports data for only the combinations (types) that are specified in the TYPES statement. A partial view of the output data set is shown below: Page100

#### Using the NWAY Option in PROC MEANS to Combine Class Variables

Another way to specify a combination of class variables is to use the NWAY option in PROC MEANS:

**General form, NWAY option in the PROC MEANS statement:**

**PROC MEANS NWAY;**

where **NWAY** specifies that the output will contain statistics for the combination of all specified class variables (the observations that have the highest \_TYPE\_ value).

The NWAY option enables you to generate summary statistics for **one** particular combination of class variables—**all** of the class variables—in a single PROC MEANS step. Therefore, to generate statistics for several different combinations of class variables, you can specify a separate PROC MEANS step that contains the NWAY option for **each** combination.

**Example: Using the NWAY Option in Multiple PROC MEANS Steps**

Suppose you want to calculate average employee salaries and to group results for the following combinations of class variables:

- Employee\_Country and Department
- Employee\_Country and Employee\_Gender.

You can use two PROC MEANS steps, each containing the NWAY option, as shown below. The first PROC MEANS step generates statistics for the first combination of class variables, and the second PROC MEANS steps generates statistics for the second combination of class variables.

```
proc means data=company.organization nway;
  class employee_country department;
  var salary;
  output out=summary1
    n=employees
    mean=average;
run;
proc means data=company.organization nway;
  class employee_country employee_gender;
  var salary;
  output out=summary2
    n=employees
    mean=average;
run;
```

When processing this program, SAS must read the data set once for each PROC MEANS step.

This program generates an output report and two output data sets. The report, shown in part below, has a separate table for each PROC MEANS step: Page100

Each output data set summarizes and reports data for one of the types that are specified in the TYPES statement. A partial view of each output data set is shown below: Page100

**Using the WHERE= Option in PROC MEANS to Combine Class Variables**

Yet another way to specify a combination of class variables is to use the **WHERE= data set option** in the OUTPUT statement:

**General form, WHERE= data set option in a basic OUTPUT statement:**

**OUTPUT <OUT=SAS-data-set> <output-statistic-specification(s)> (WHERE=(where-expression-1 <logical-operator where-expression-n>));**

where

- SAS-data-set specifies the new output data set as a 1-level or 2-level name.
- output-statistic-specification(s) specifies the statistic(s) to store in the output data set and names one or more variables that contain the statistics.
- where-expression is an arithmetic or logical expression that consists of a sequence of operators, operands, and SAS functions. An operand is a variable, a SAS function, or a constant. An operator is a symbol that requests a comparison, logical operation, or arithmetic calculation. The expression must be enclosed in parentheses.
- logical-operator can be AND, AND NOT, OR, or OR NOT.

When you use the WHERE= option in the OUTPUT statement, SAS must calculate all possible combinations of class variables, and subsetting does not occur until the results are written to output.

**Example: Using the WHERE= Option in PROC MEANS**

Suppose you want to calculate average employee salaries and to group results for two 2-way combinations of the three class variables Employee\_Country, Department, and Employee\_Gender. All possible combinations of these variables are listed below, and the two combinations that you want are highlighted:

| _TYPE_Value | Variables Combined                              | Dimension    |
|-------------|-------------------------------------------------|--------------|
| 1           | Employee_Gender                                 | 1-way        |
| 2           | Department                                      | 1-way        |
| 3           | Department * Employee_Gender                    | 2-way        |
| 4           | Employee_Country                                | 1-way        |
| 5           | <b>Employee_Country * Employee_Gender</b>       | <b>2-way</b> |
| 6           | <b>Employee_Country * Department</b>            | <b>2-way</b> |
| 7           | Employee_Country * Department * Employee_Gender | 3-way        |

To specify the types by \_TYPE\_ value, you can use the WHERE= option in the OUTPUT statement as shown below:

```
proc means data=company.organization;
  class employee_country department
    employee_gender;
  var salary;
  output out=summary
    (where=(_type_in (5,6)))
    n=employees
    mean=average;
run;
```

A partial view of the output report is shown below: Page100

A partial view of the output data set **Work.Summary** is shown below: Page100

Next, let's compare the resources that are used by these summarization techniques:

- the TYPES statement in PROC MEANS
- the NWAY option in multiple PROC MEANS steps
- the WHERE= option in PROC MEANS.

### Comparative Example: Displaying Summary Statistics for Combinations of Class Variables

Suppose you want to summarize the data set **Retail.Organization** by calculating average employee salaries for two 3-way combinations of four class variables:

- Employee\_Country, Department, and Employee\_Gender
- Department, Section, and Employee\_Gender.

You can use the following techniques to produce an output report and one or more output data sets:

1. TYPES Statement in PROC MEANS
2. NWAY Option in Two PROC MEANS Steps
3. WHERE= Option in PROC MEANS

The following programs show each of these techniques. You can use these samples as models for creating benchmark programs in your own environment. Your results might vary depending on the structure of your data, your operating environment, and the resources that are available at your site. You can also view general recommendations for summarizing data for combinations of class variables.

#### Programming Techniques

##### **1. TYPES Statement in PROC MEANS**

This program calculates the average employee salary for two 3-way combinations of the class variables Employee\_Country, Department, Employee\_Gender, and Section. The TYPES statement specifies the two combinations. The program generates an output report and an output data set named **Summary**.

```
proc means data=retail.organization mean;
  class employee_country department
    employee_gender section;
  var salary;
  types employee_country*department*employee_gender
    department*section*employee_gender;
  output out=summary
    n=employees
    mean=average;
run;
```

##### **2. NWAY Option in Two PROC MEANS Steps**

Each of the two PROC MEANS steps in this program calculates the average employee salary for a combination of three of the four class variables Employee\_Country, Department, Employee\_Gender, and Section. In each step, the NWAY option specifies that all three variables that are specified in the CLASS statement should be combined. The program generates an output report and two output data sets named **Summary1** and **Summary2**.

```
proc means data=retail.organization nway;
  class employee_country department
    employee_gender;
```

```

var salary;
output out=summary1
      n=employees
      mean=average;
run;

proc means data=retail.organization nway;
  class department section
    employee_gender;
  var salary;
  output out=summary2
      n=employees
      mean=average;
run;

```

### 3. WHERE= Option in PROC MEANS

This program calculates the average employee salary for two 3-way combinations of the class variables Employee\_Country, Department, Employee\_Gender, and Section. The WHERE= data set option in the OUTPUT statement specifies the two combinations by their \_TYPE\_ values. The program generates an output report and an output data set named **Summary3**.

```

proc means data=retail.organization;
  class employee_country department
    employee_gender section;
  var salary;
  output out=summary3 (where=(_type_in(7,14)))
      n=employees
      mean=average;
run;

```

## General Recommendations

- To summarize data for particular combinations of class variables, use the TYPES statement in PROC MEANS.

Suppose you have written the following program to summarize data for three combinations of class variables:

```

proc means data=company.orders mean;
  class cust_id employee_id order_type
    region;
  var retail_price;
  types cust_id*employee_id*order_type
    employee_id*order_type*region
    cust_id*order_type;
  output out=sum_orders mean=average;
run;

```

**Q10.**

Which of the following statements about this program is true?

This program generates an output report and an output data set, and it prints the output data set.

You could generate similar results less efficiently by using the NWAY option in three PROC MEANS steps.

## Additional Features

The WAYS statement in PROC MEANS provides yet another way to display summary statistics for combinations of class variables. In the WAYS statement, you specify one or more integers that define the number of class variables to combine in order to form all the unique combinations of class variables.

For example, the following program uses the WAYS statement to create summary statistics for the following combinations of the three class variables Employee\_Country, Department, and Employee\_Gender:

- each individual variable (all 1-way combinations)
- all 2-way combinations (Employee\_Country and Department, Employee\_Country and Employee\_Gender, and Employee\_Gender and Department).

```
proc means data=company.organization mean;
  class employee_country department
    employee_gender;
  var salary;
  ways 1 2;
  output out=summary
    mean=average;
run;
```

The WAYS statement can be used instead of or in addition to the TYPES statement.



For more information about the WAYS statement, see the SAS documentation.

## **Chapter Summary**

### **Using an Index for Efficient WHERE Processing**

When processing a WHERE expression, SAS determines whether it is more efficient to access observations in a data set sequentially, by searching through all observations, or directly, by using an index to access specific observations. Using an index to process a WHERE expression might improve performance and is referred to as optimizing the WHERE expression. By deciding whether to create an index, you play a role in determining which access method SAS can use.

In order to decide whether to use an index, you must evaluate the benefits and costs of using an index.

SAS performs a series of steps to determine whether to process a WHERE expression by using an index or by reading all the observations in the data file sequentially.

#### **Identifying Available Indexes**

First, SAS determines whether there are any existing indexes that might be used to process the WHERE expression. Specifically, SAS checks the variable in each condition in the WHERE expression to determine whether the variable is a key variable in either a simple index or a composite index. No matter how many indexes are available, SAS can use only one index to process a WHERE expression. So, if multiple indexes are available, SAS must choose between them.

It is most common for SAS to use an index to process just one condition in a WHERE expression. However, in a process called compound optimization, SAS can use a composite index to optimize multiple conditions on multiple variables, which are joined with a logical operator such as AND.

#### **Identifying Conditions That Can Be Optimized**

Second, SAS looks for operators and functions that can be optimized in the WHERE conditions that contain key variables. There are also certain operators and functions that cannot be optimized. For compound optimization, WHERE conditions must meet slightly different criteria in order to be candidates for optimization.

#### **Estimating the Number of Observations**

Third, SAS estimates how many observations will be qualified by the index. When multiple indexes exist, SAS selects the one that appears to produce the fewest qualified observations (the smallest subset). Whether or not SAS uses an index depends on the percentage of observations that are qualified (the size of the subset relative to the size of the data set). It is more efficient to use indexed access for a small subset and sequential access for a large subset. If SAS estimates that the number of qualified observations is less than 3% of the data file, SAS automatically uses the index and does not go on to compare probable resource usage.

To help SAS estimate how many observations would be selected by a WHERE expression, each index stores 21 statistics called cumulative percentiles, or centiles. Centiles provide information about the distribution of values for the indexed variable.

#### **Comparing Probable Resource Usage**

Fourth, SAS decides whether it is faster (cheaper) to satisfy the WHERE expression by using the index or by reading all of the observations sequentially. To make the decision, SAS predicts how many I/O

operations will be required in order to satisfy the WHERE expression for each of the access methods, and then compares the two resource costs.

Several factors affect the number of I/O operations that are required for WHERE processing, including the following:

- subset size relative to data set size
- number of pages in the data file
- order of the data
- cost to uncompress a compressed file for a sequential read.

Data type and length are two other factors that affect index efficiency.

### **Deciding Whether to Create an Index**

When you use a WHERE expression to select a subset, you can use specific guidelines to decide whether it is efficient to create an index. Depending on factors such as the size of the subset relative to the size of the data set, you might or might not choose to create an index.

In most situations, it is best to let SAS determine whether or not to use an index for WHERE processing. However, sometimes you might want to control whether or not SAS uses an existing index. You can use either of the data set options IDXWHERE= or IDXNAME=, but not both at the same time, to control index usage. You can specify MSGLEVEL=I to tell SAS to display information about index usage in the SAS log.

### **Comparing Procedures That Produce Detail Reports**

When you want to use a query to produce a detail report, you can choose between the PRINT procedure and the SQL procedure. To perform a particular task, a single-purpose tool like PROC PRINT generally uses fewer computer resources than a multi-purpose tool like PROC SQL. However, PROC SQL often requires fewer and shorter statements to achieve the results that you want.

For detail reports, a PROC PRINT step often, but not always, uses fewer resources than a PROC SQL step:

- PROC PRINT is usually more efficient than PROC SQL for generating a simple detail report, a subset detail report, and a sorted detail report.
- PRINT PRINT and PROC SQL will likely have similar resource usage for generating a sorted subset detail report.

### **Comparing Tools for Summarizing Data**

SAS provides a variety of tools for summarizing data, including the MEANS procedure (or SUMMARY procedure), the TABULATE procedure, the REPORT procedure, the SQL procedure, and the DATA step.

If you are summarizing data for one class variable, the tools in each of the following groups are similar in resource usage:

- PROC MEANS (or PROC SUMMARY), PROC REPORT, and PROC TABULATE
- PROC SQL and the DATA step.

However, the relative efficiency of the two groups of tools varies according to the shape of the data.

You can use PROC MEANS in a variety of ways to produce summary statistics for combinations of class variables. Each combination of class variables is called a type.

To summarize data for **all** combinations of class variables, you can use a basic PROC MEANS step (or PROC SUMMARY step). To produce summary statistics for **specific** combinations of class variables, you can use PROC MEANS in the following ways :

- the TYPES statement in a PROC MEANS step
- the NWAY option in multiple PROC MEANS steps
- the WHERE= option in a PROC MEANS step.

These three techniques vary in efficiency; the TYPES statement in PROC MEANS is the most efficient.

You can also use the WAYS statement in PROC MEANS to produce summary statistics for specific combinations of class variables.

Review the related comparative examples:

- Displaying Summary Statistics for One Class Variable
- Displaying Summary Statistics for Combinations of Class Variables.

## Chapter Quiz

Select the best answer for each question

1. Why can using an index reduce the number of I/O operations that are required for accessing a small subset?
  - a. Using an index requires larger input buffers, which can hold more pages.
  - b. The index does not have to be loaded into an input buffer.
  - c. The number of observations that SAS has to load into the program data vector (PDV) is decreased.
  - d. The number of pages that SAS has to load into input buffers is decreased.
2. You want to select a subset of observations in the data set **Company.Products**, and you have defined a simple index on the variable Rating. SAS **cannot** use the index to process which of the following WHERE statements?
  - a. `where rating is missing;`
  - b. `where rating=int(rating);`
  - c. `where rating between 3.5 and 7.5;`
  - d. `where rating=5.5;`
3. In which of the following situations is sequential access likely to be more efficient than direct access for WHERE processing?
  - a. The subset contains over 75% of the observations in the data set.
  - b. The WHERE expression specifies both key variables in a single composite index.
  - c. The data is sorted on the key variable.
  - d. The data set is very large.
4. You want to summarize data for one class variable, and you are trying to decide whether to use PROC MEANS (or PROC SUMMARY), PROC REPORT, PROC TABULATE, PROC SQL, or the DATA step with PROC SORT. Which of the following statements about the efficiency of these summarization tools is **not** true?
  - a. PROC MEANS (or PROC SUMMARY), PROC REPORT, and PROC TABULATE have similar resource usage.
  - b. The efficiency of all these tools is affected by the shape of the data.
  - c. The SQL procedure is always the least efficient because it is a general-purpose tool.
  - d. PROC SQL and the DATA step with PROC SORT have similar resource usage.
5. Which of the following techniques is most efficient for summarizing data for specific combinations of class variables?
  - a. the NWAY option in multiple PROC MEANS steps
  - b. the TYPES statement in a PROC MEANS step
  - c. the WHERE= option in a PROC MEANS step
  - d. a basic PROC MEANS step.

