SAS® 9.4 V9 LIBNAME Engine: Reference

SAS® Documentation
January 9, 2023

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2019. *SAS® 9.4 V9 LIBNAME Engine: Reference*. Cary, NC: SAS Institute Inc.

SAS® 9.4 V9 LIBNAME Engine: Reference

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

January 2023

9.4-P3:engsas7bdat

Contents

PART 4 Using V9 Engine Features

PART 1

Getting Started

1

Getting Started with the V9 Engine

About the V9 Engine

If you submit SAS procedures and DATA steps, or if you use SAS programs that were written by someone else, you probably use the V9 engine. Here are the essential concepts:

- If you do not specify an engine in a library assignment, and if the ENGINE system option is not set, then the default engine is used. In SAS®9, the shipped default Base SAS engine is the V9 engine. See "LIBNAME Statement: V9 Engine" on page 17.
- If you are unfamiliar with library assignment, try creating a V9 engine data set. For more information, see "SAS Libraries" in *SAS Programmer's Guide: Essentials*. At some customer sites, an administrator assigns SAS libraries.
- This document covers features that are supported by the V9 engine only or have significant details that are specific to the engine. See Chapter 7, "Introduction to Engine-Specific Features," on page 67. For information about other SAS engines, see "SAS Engines" in *SAS Programmer's Guide: Essentials*.

SAS Files and Member Types

The V9 engine recognizes the following member types in a SAS library. (Less-common member types are omitted here.) Other engines can also create some of these member types.

Table 1.1 *Most Common V9 Library Member Types*

SAS File	Member Type
data set	DATA
DATA step view, PROC SQL view	VIEW
catalog	CATALOG
stored, compiled DATA step program	PROGRAM
item store	ITEMSTOR

In output from the DATASETS procedure that lists the contents of a library, you might notice that additional files are listed below a data set. The additional files have the same name as the data set but a different member type. The files are attributes of the data set and are stored in a separate file. The member types in the following table are associated with a data set and cannot be specified in a MEMTYPE= option.

Table 1.2 *Data Set Attributes That Are Stored in Separate Files*

SAS File	Member Type
audit trail	AUDIT
extended attributes	XATTR
index	INDEX

For example, here is a portion of PROC DATASETS output. Notice that the `flowers` data set has an index.

Output 1.1 *PROC DATASETS Output Showing the Library Members*

#	Name	Member Type	File Size	Last Modified
1	CONTAINERS	VIEW	5KB	07/19/2018 08:33:19
2	FLOWERS	DATA	192KB	01/18/2019 17:24:24
	FLOWERS	INDEX	9KB	01/18/2019 17:24:24
3	FORMATS	CATALOG	17KB	07/03/2018 14:06:19
4	RESTOCK	DATA	128KB	11/09/2018 11:50:50

File extensions vary, depending on the operating environment:

- "File Extensions and Member Types in UNIX Environments" in *SAS Companion for UNIX Environments*
- "File Extensions for SAS Files" in *SAS Companion for Windows*
- "UFS Libraries" in *SAS Companion for z/OS*

External Data

Base SAS software can import and export some external text files without using a SAS/ACCESS engine. These files are usually referred to as raw data or delimited data.

In the DATA step, you can read and write external data by using the INFILE and FILE statements, respectively. The default delimiter is a blank. To specify a different delimiter (for example, a comma), use the DELIMITER= option. The DSD= option and other options might also be required. See "INFILE Statement" in *SAS DATA Step Statements: Reference* and "FILE Statement" in *SAS DATA Step Statements: Reference*.

You might be able to use the IMPORT and EXPORT procedures to read and write external data. A file with a .csv extension is imported automatically as a comma-separated file. To import other types of data files, specify the DBMS= option. If you specify DBMS=DLM, then the DELIMITER= option is required. See "IMPORT Procedure" in *Base SAS Procedures Guide* and "EXPORT Procedure" in *Base SAS Procedures Guide*.

Data Types

The V9 engine supports character data with a CHAR data type and numeric data with a DOUBLE data type. For full details, see "Data Types" in *SAS Programmer's Guide: Essentials*.

Table 1.3 V9 Engine Data Types

Data Type	Description
CHAR	Fixed-length character string (Base SAS character variable)
DOUBLE	Double-precision, floating-point number (Base SAS numeric variable)

TIP Within a DATA step, a variable is assumed to be numeric unless character is indicated with the dollar sign ($).

2

Quick Tutorials for the V9 Engine

Example: Create and Print a V9 Engine Data Set

Example

This example assigns a library, creates a SAS data set, and prints details about the data set. This example uses data from the http://support.sas.com/documentation/onlinedoc/viya/exampledatasets/cholesterol.csv file. Download the cholesterol.csv file to a directory that is accessible to SAS. Specify that path in the FILENAME statement.

```
libname myfiles v9 '/mydata/examples';        /*1*/
filename chol '/downloads/cholesterol.csv';   /*2*/

data myfiles.mypatientdata;                   /*3*/
```

```
   infile chol delimiter=',' firstobs=2;       /* 4 */
   input patient $ measurement cholesterol;    /* 5 */
run;                                           /* 6 */

proc print data=myfiles.mypatientdata;         /* 7 */
run;
```

1. The LIBNAME statement assigns the `myfiles` libref to an existing physical location for the data storage of a SAS library. The statement also specifies the V9 engine. The libref is used later as a nickname to reference the library, its physical location, and any other arguments of the LIBNAME statement.
2. The FILENAME statement assigns the `chol` fileref to the physical location of a comma-separated file, `cholesterol.csv`. The fileref is used later as a nickname to reference the file.
3. The DATA step creates a new data set named `mypatientdata` in the `myfiles` library.
4. In the DATA step, the INFILE statement specifies the `chol` fileref. The external file that is referenced by `chol` contains the data that is imported into the `mypatientdata` data set.

   The DELIMITER= option specifies that the external file is comma-delimited.

   The FIRSTOBS= option specifies to start the import at the second row, because the first row of the external file contains the variable names of the external data.
5. The INPUT statement specifies SAS variable names for the data set. The dollar sign ($) indicates that `patient` is a character variable. In SAS, variables are numeric by default.
6. The RUN statement ends the DATA step.
7. The PRINT procedure prints the `myfiles.mypatientdata` data set.

Output 2.1 *PROC PRINT Output of myfiles.mypatientdata*

Obs	patient	measurement	cholesterol
1	A2	1	150
2	A2	2	145
3	A2	3	148
4	B1	1	301
5	B1	2	280
6	B1	3	275
7	C1	1	212
8	C1	2	220
9	C1	3	240

The printed data set shows the expected behavior. However, as a best practice, always check the SAS log for errors, warnings, and notes. In this case, the log notes confirm that the processing was successful:

```
82         libname myfiles v9 '/mydata/examples';
NOTE: Libref MYFILES was successfully assigned as follows:
      Engine:        V9
      Physical Name: /mydata/examples
82       !
83         filename chol '/downloads/cholesterol.csv';
84
85         data myfiles.mypatientdata;
86            infile chol delimiter=',' firstobs=2;
87            input patient $ measurement cholesterol;
88         run;
NOTE: The infile CHOL is:
      Filename=/downloads/cholesterol.csv,
      Owner Name=userid,Group Name=mygroup,
      Access Permission=-rwxrwxrwx,
      Last Modified=19Nov2019:09:29:02,
      File Size (bytes)=121
NOTE: 9 records were read from the infile CHOL.
      The minimum record length was 8.
      The maximum record length was 8.
NOTE: The data set MYFILES.MYPATIENTDATA has 9 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.02 seconds

88       !
89
90         proc print data=myfiles.mypatientdata;
91         run;
NOTE: There were 9 observations read from the data set MYFILES.MYPATIENTDATA.
NOTE: The PROCEDURE PRINT printed page 1.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.04 seconds
      cpu time            0.04 seconds
```

Key Ideas

- If you do not specify an engine in a library assignment, and if the ENGINE system option is not set, then the default engine is used. In SAS®9, the shipped default Base SAS engine is the V9 engine. See "LIBNAME Statement: V9 Engine" on page 17.
- In the DATA step, you can read and write external data by using the INFILE and FILE statements, respectively. The default delimiter is a blank. To specify a different delimiter (for example, a comma), use the DELIMITER= option. The DSD= option and other options might also be required.
- You can also import CSV data by using the IMPORT procedure. See "Example: Read a Comma-Delimited File" in *SAS Programmer's Guide: Essentials*.

See Also

- "LIBNAME Statement: V9 Engine" on page 17
- "INFILE Statement" in *SAS DATA Step Statements: Reference*

- "INPUT Statement" in *SAS DATA Step Statements: Reference*
- "PRINT Procedure" in *Base SAS Procedures Guide*
- "SAS Libraries" in *SAS Programmer's Guide: Essentials*

Example: View Information about a V9 Engine Data Set

Example Code

This example prints information about the `myfiles.mypatientdata` data set. This data set is created in "Example: Create and Print a V9 Engine Data Set" on page 7.

```
libname myfiles v9 '/mydata/examples';
proc datasets library=myfiles;        /* 1 */
   contents data=mypatientdata;       /* 2 */
run;
quit;                                 /* 3 */
```

1. In the DATASETS procedure, the LIBRARY= option specifies the `myfiles` libref.
2. The CONTENTS statement requests information about the `mypatientdata` data set.
3. PROC DATASETS is an interactive procedure, which means that it supports RUN-group processing. For interactive procedures, always include a QUIT statement to stop the procedure.

The output from PROC DATASETS shows the attributes of the data set.

Output 2.2 *PROC DATASETS Data Set Information*

Data Set Name	MYFILES.MYPATIENTDATA	**Observations**	9
Member Type	DATA	**Variables**	3
Engine	V9	**Indexes**	0
Created	11/19/2019 13:20:11	**Observation Length**	24
Last Modified	11/19/2019 13:20:11	**Deleted Observations**	0
Protection		**Compressed**	NO
Data Set Type		**Sorted**	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64, LINUX_POWER_64		
Encoding	utf-8 Unicode (UTF-8)		

Output 2.3 *PROC DATASETS Engine/Host Dependent Information*

Engine/Host Dependent Information	
Data Set Page Size	65536
Number of Data Set Pages	1
First Data Page	1
Max Obs per Page	2714
Obs in First Data Page	9
Number of Data Set Repairs	0
Filename	/mydata/examples/mypatientdata.sas7bdat
Release Created	V.0305M0
Host Created	Linux
Inode Number	83863550
Access Permission	rw-r--r--
Owner Name	userid
File Size	128KB
File Size (bytes)	131072

Output 2.4 *PROC DATASETS List of Variables and Attributes*

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
3	cholesterol	Num	8
2	measurement	Num	8
1	patient	Char	8

Key Ideas

- Use the CONTENTS statement of PROC DATASET, or PROC CONTENTS, to list the attributes of a SAS data set.
- Many data set attributes are listed by default. Attributes include the data set name, member type, engine that created the file, number of observations (rows), number of variables (columns), and so on. The file name includes the physical location of the file and an extension that identifies the engine that created the table. For the V9 engine, the file extension of a data set is .sas7bdat. In the list of variables, the listed attributes include the name, data type, length, and any format or informat that is assigned to each variable.
- The V9 engine supports numeric data with a DOUBLE data type and character data with a CHAR data type.

See Also

- "DATASETS Procedure" in *Base SAS Procedures Guide*
- "Understanding the DATASETS Procedure" in *Step-by-Step Programming with Base SAS*

Example: Load SAS Data to CAS by Using the CASUTIL Procedure

Example Code

In this example, the CASUTIL procedure loads a SAS data set to SAS® Cloud Analytic Services (CAS). Your site must license and install SAS Viya to access this functionality.

To run this example, first create the `mypatientdata` data set in "Example: Create and Print a V9 Engine Data Set" on page 7.

```
cas casauto host="cloud.example.com" port=5570; /* 1 */
libname myfiles v9 '/mydata/examples';          /* 2 */

proc casutil;
   load data=myfiles.mypatientdata promote;     /* 3 */
   contents casdata='mypatientdata';            /* 4 */
run;
```

1. The CAS statement starts a CAS session and specifies `casauto` as the CAS session name. Replace the HOST= and PORT= values with your connection information.
2. The LIBNAME statement assigns the `myfiles` libref and V9 engine to the location of the `mypatientdata` data set.
3. In PROC CASUTIL, the LOAD statement with the DATA= option reads data from `myfiles.mypatientdata` and loads it into memory in CAS.

   The PROMOTE option loads the table with global scope.
4. The CONTENTS statement displays metadata about the table in CAS.

Output 2.5 *Contents of the SAS Data Set That Is Loaded to CAS*

Table Information for Caslib CASUSERHDFS(userid)

Table Name	Number of Rows	Number of Columns	Indexed Columns	NLS encoding	Created	Last Modified	Promoted Table	Repeated Table	View	Compressed
MYPATIENTDATA	9	3	0	utf-8	2019-11-19T10:32:20-05:00	2019-11-19T10:32:20-05:00	Yes	No	No	No

Detail Information for mypatientdata in Caslib CASUSERHDFS(userid).

Node	Number of Blocks	Active Blocks	Rows	Data size	Variable Data size	Blocks Mapped	Memory Mapped	Blocks Unmapped	Memory Unmapped	Blocks Allocated	Memory Allocated	Index Size	Compressed Size	Compression Ratio
ALL	18	9	9	216	0	9	1800	9	1800	0	0	0	0	0

Column Information for MYPATIENTDATA in Caslib CASUSERHDFS(userid)

Column	Type	Length	Formatted Length	Format Width	Format Decimal
patient	char	8	8	0	0
measurement	double	8	12	0	0
cholesterol	double	8	12	0	0

Key Ideas

- If you can access a SAS data set with the LIBNAME statement, you can use PROC CASUTIL to load the data into CAS.
- The LOAD DATA= form of the LOAD statement is the recommended syntax for programmers who want to use SAS language elements.

See Also

- "Example: Load a SAS Data Set to a CAS Server" in *SAS Programmer's Guide: Essentials*
- "LOAD Statement" in *SAS Cloud Analytic Services: User's Guide*
- *An Introduction to SAS Viya Programming*

PART 2

Syntax

3

LIBNAME Statement, V9 Engine

Dictionary

LIBNAME Statement: V9 Engine

Assigns or deassigns a libref (a shortcut name) to a SAS library, concatenates SAS libraries, or concatenates SAS catalogs.

Valid in: Anywhere

Category: Data Access

Restriction: When SAS is in a locked-down state, the LIBNAME statement is not available for files that are not in the lockdown path list. For more information, see "SAS Processing Restrictions for Servers in a Locked-Down State" in *SAS Programmer's Guide: Essentials*.

See: To clear one or all librefs (CLEAR) or to list the characteristics of a SAS library (LIST), see "LIBNAME Statement" in *SAS Global Statements: Reference*.

LIBNAME Statement under Windows, UNIX, and z/OS

Syntax

Form 1: **LIBNAME** *libref* V9 '*SAS-library*' <*options*><*host-options*>;

Form 2: **LIBNAME** *libref* V9 (*library-specification-1* <…*library-specification-n*>) <*options*>;

Syntax Description

libref

is a shortcut name for the aggregate storage location where your SAS files are stored. The libref must be a valid SAS name.

Length 1 to 8 bytes

Tip The SAS library assignment lasts only for the duration of the SAS session or until you change it or discontinue it with another LIBNAME statement.

See Other than length, a libref follows the same rules of syntax as any SAS name. For more information about SAS naming conventions, see "Rules for Most SAS Names" in *SAS Programmer's Guide: Essentials*

V9

specifies the V9 engine name. The V9 engine can process data that was created by the V9, V8, and V7 engines.

The shipped default Base SAS engine is BASE. In SAS®9 and SAS® Viya®, the BASE engine is an alias for the V9 engine. If you do not specify an engine name when you create a new library, and if you have not specified the ENGINE system option, then the V9 engine is automatically selected. If the library location already contains SAS files, then SAS might be able to assign the correct engine based on those files. For example, if the location contains V9 data sets only, then SAS assigns the V9 engine. However, if a library location contains a mix of different engine files, then SAS might not assign the engine you want. Therefore, specifying the engine is a best practice. For more information, see "Default Engine for a Mixed Library" on page 213.

'*SAS-library*'

is the physical name for the SAS library. The physical name is the name that is recognized by the operating environment. Enclose the physical name in single or double quotation marks.

Interaction You can set the DLCREATEDIR system option to create the directory for the SAS library that is specified in the LIBNAME statement if that directory does not exist. The default is NODLCREATEDIR.

Operating environment For details about specifying the physical names of files, see the SAS documentation for your operating environment.

library-specification

is two or more SAS libraries that are specified by physical names, previously assigned librefs, or a combination of the two. If you specify a physical name, enclose it in single or double quotation marks. Separate each specification with either a blank or a comma and enclose the entire list in parentheses. This form is used for library and catalog concatenation.

'*SAS-library*'

is the physical name of a SAS library, enclosed in quotation marks.

libref

is the name of a previously assigned libref.

Restriction When concatenating libraries, you cannot specify options that are specific to an engine or an operating environment.

See "Concatenating SAS Libraries or SAS Catalogs (Form 2)" on page 19

options
are the V9 engine LIBNAME statement options. See "LIBNAME Statement Options, V9 Engine" on page 23.

Restriction When concatenating libraries, you cannot specify options that are specific to an engine or an operating environment.

host-options
are specific to an operating environment. See the SAS documentation for your operating environment, listed below.

Restriction When concatenating libraries, you cannot specify options that are specific to an engine or an operating environment.

See "LIBNAME Statement: Windows" in *SAS Companion for Windows*

"LIBNAME Statement: UNIX" in *SAS Companion for UNIX Environments*

"LIBNAME Statement: z/OS" in *SAS Companion for z/OS*

Details

Assigning a SAS Library (*Form 1*)

The SAS library assignment lasts only for the duration of the SAS session or until you change the libref or discontinue it with another LIBNAME statement. The simplest form of the LIBNAME statement specifies only a libref and the physical name of a SAS library. If you omit the engine name, you invoke the default engine. For information about specifying engines, see the V9 engine description on page 18. See also "Example 1: Assigning and Using a Libref" on page 21.

Concatenating SAS Libraries or SAS Catalogs (*Form 2*)

When you logically concatenate two or more SAS libraries, you can reference them all with one libref. In the LIBNAME statement, specify each library with its physical pathname or its previously assigned libref. If you specify a physical pathname, enclose it in single or double quotation marks. You can use a combination of librefs and physical pathnames. Separate each specification with either a blank or a comma and enclose the entire list in parentheses. For details about the behavior, see "Library Concatenation" in *SAS Programmer's Guide: Essentials*. See also "Example 2: Logically Concatenating SAS Libraries" on page 21.

When you logically concatenate two or more SAS libraries, you also concatenate SAS catalogs. For details about the behavior, see "Catalog Concatenation" on page 170. See also "Example 3: Implicitly Concatenating SAS Catalogs" on page 21.

Automatically Creating the Library Directory

You can set the DLCREATEDIR system option to automatically create a new subdirectory for the SAS library. For more information, see "DLCREATEDIR System Option" in *SAS System Options: Reference*.

z/OS Specifics: For more information, see "DLCREATEDIR System Option: z/OS" in *SAS Companion for z/OS*.

Metadata-Bound Libraries

The V9 LIBNAME engine can enforce permissions on a user and group basis to SAS data sets that are bound to secured table objects in the SAS Metadata Server. Metadata-bound libraries provide enhanced protection for SAS data sets and SAS views. A connection to the metadata server is required in order to access metadata-bound data.

For more information, see *SAS Guide to Metadata-Bound Libraries* and the AUTHLIB procedure in *Base SAS Procedures Guide*.

If you have questions or need assistance accessing your data, contact your local SAS Administrator.

Levels of Locking

By default, SAS provides the greatest possible level of concurrent access while guaranteeing the integrity of the data, with member-level locking. Member-level locking, which is at the SAS data set level, enables Read access to many sessions, statements, or procedures. This locking restricts all other access to the SAS data set when a session, statement, or procedure acquires Update or Output access.

In some cases, you might want to guarantee the integrity of your data by controlling the levels of Update access yourself. Use the CNTLLEV= data set option to control locking levels. For the V9 engine, the CNTLLEV= data set option enables locking at three levels. For more information, see:

- "Levels of Locking" in *SAS Programmer's Guide: Essentials*
- "CNTLLEV= Data Set Option" in *SAS Data Set Options: Reference*
- "LOCK Statement" in *SAS Global Statements: Reference*

Comparisons

- Use the LIBNAME statement to reference a SAS library. Use the FILENAME statement to reference an external file.
- You can use the CATNAME statement or the LIBNAME statement to concatenate SAS catalogs. The CATNAME statement enables you to specify the names of the catalogs that you want to concatenate. The LIBNAME statement concatenates all like-named catalogs in the specified SAS libraries.

Examples

Example 1: Assigning and Using a Libref

This example assigns the libref `sales` to an aggregate storage location that is specified in quotation marks as a physical pathname. The DATA step creates the `quarter1` data set and stores it in the `sales` library location, `c:\myfiles`. The PROC PRINT step references the data set by its two-level name, `sales.quarter1`.

```
libname sales 'c:\myfiles';
data sales.quarter1;
infile 'your-input-file';
input salesrep $20. +6 jansales febsales
   marsales;
run;
proc print data=sales.quarter1;
run;
```

Example 2: Logically Concatenating SAS Libraries

In this example, the `oldlib` library contains last year's files, and the `newlib` library contains this year's files. Both libraries contain V9 engine files, so the engine name is not necessary.

```
libname oldlib 'C:\old-library';
libname newlib 'C:\new-library';
```

The following code concatenates `oldlib` and `newlib` by specifying the two librefs:

```
libname allmine (oldlib newlib);
```

The following code adds a third library to the concatenation. This example shows that you can specify librefs and physical pathnames in the same concatenation:

```
libname allmine (oldlib newlib 'C:\other-library');
```

In the SAS log, the individual libraries are referred to as numbered levels:

```
NOTE: Libref ALLMINE was successfully assigned as follows:
      Levels:           3
      Engine(1):        V9
      Physical Name(1): C:\old-library
      Engine(2):        V9
      Physical Name(2): C:\new-library
      Engine(3):        V9
      Physical Name(3): C:\other-library
```

Example 3: Implicitly Concatenating SAS Catalogs

For this example, all three of the individual libraries in `allmine` contain a SAS catalog named `codes`.

```
libname allmine (oldlib newlib 'C:\other-library');
```

After the concatenation, you can use the specification `allmine.codes` to access the entries from all three catalogs. See "Catalog Concatenation" on page 170.

To logically concatenate SAS catalogs that have different names, use the CATNAME statement instead. See "CATNAME Statement" in *SAS Global Statements: Reference*.

Example 4: Permanently Storing Data Sets with One-Level Names

When you create or reference a data set with a one-level name, the default location is the Work library, and the data set is temporary. If you want the convenience of specifying a one-level name for permanent, not temporary, SAS files, then set the USER= system option. This example stores the data set `quarter1` permanently without using a LIBNAME statement to assign a libref to a storage location:

```
options user='SAS-library';
data quarter1;
infile 'your-input-file';
input salesrep $20. +6 jansales febsales
   marsales;
run;
proc print data=quarter1;
run;
```

See Also

- "SAS Libraries" in *SAS Programmer's Guide: Essentials*

Statements

- "LIBNAME Statement" in *SAS Global Statements: Reference*

System Options

- "DLCREATEDIR System Option" in *SAS System Options: Reference*

4

LIBNAME Statement Options, V9 Engine

Dictionary

ACCESS= LIBNAME Statement Option

Assigns an access attribute to the SAS library.

Operating environment: Some operating environments support LIBNAME statement options that have similar functions to the ACCESS= option. See the SAS documentation for your operating environment.

Syntax

ACCESS=READONLY | TEMP

Syntax Description

READONLY
assigns a read-only attribute to an entire SAS library. SAS does not allow you to open a data set in the library in order to update information or write new information.

TEMP
specifies that the SAS library be treated as a scratch library. That is, the system does not consume CPU cycles to ensure that the files in a Temp library do not become corrupted.

Tip Use ACCESS=TEMP to save resources only when the data is recoverable.

Details

Using ACCESS=READONLY prevents writing to the data source. If this option is omitted, tables can be read, updated, and created if you have the necessary data source privileges.

AUTHADMIN=, AUTHALTER=, AUTHPW=, AUTHREAD= LIBNAME Statement Options

Specify access or passwords for metadata-bound libraries.

See: *SAS Guide to Metadata-Bound Libraries*

Syntax

AUTHADMIN=YES | NO
AUTHALTER=*alter-password*
AUTHPW=*password*
AUTHREAD=*read-password*
AUTHWRITE=*write-password*

Syntax Description

AUTHADMIN= YES | NO
specifies whether an administrator can access a metadata-bound library for which corresponding metadata is corrupted, misconfigured, or missing.

Default NO

Restriction This LIBNAME option can be used only by administrators of metadata-bound libraries.

Interactions If the administrator specifies AUTHADMIN=YES in a LIBNAME statement and knows the password (or passwords) for the target data, the administrator can access that data by explicitly supplying the password (or passwords).

An administrator can choose to specify the AUTHPW= option in the LIBNAME statement as an additional method for making the metadata-bound library password available to later requests.

Note The use of AUTHADMIN=YES is intended for the administrator to correct misaligned location and metadata information. To ensure that the user who is issuing the LIBNAME statement has administrator rights to correct the misalignments, the user must have the same permissions that are needed to run the AUTHLIB procedure statements and must supply the metadata-bound data passwords when accessing the data sets.

Tip The AUTHLIB REPAIR statement is preproduction. It is recommended that you use AUTHADMIN=YES when performing any AUTHLIB REPAIR action.. As a best practice, do not use AUTHADMIN=YES in any other circumstance.

See "AUTHPW=password"

"Metadata-Bound Libraries" in *SAS Global Statements: Reference*

SAS Guide to Metadata-Bound Libraries

PROC AUTHLIB in *Base SAS Procedures Guide*

AUTHALTER=*alter-password*

Specifies an ALTER password to use only in data access requests where both of these conditions exist:

- AUTHADMIN=YES is specified in the LIBNAME statement that is referenced in the request.
- The correct password for the target metadata-bound data set or library is not otherwise available or is invalid.

Requirement The AUTHADMIN option must be set to YES for this option to have an effect.

Interaction You can use the AUTHALTER= option in the same way as the AUTHPW= option if all three of the passwords (ALTER, READ, and WRITE) are the same.

See *SAS Guide to Metadata-Bound Libraries*

AUTHPW=*password*

Specifies a password to use only in data access requests where both of these conditions exist:

- AUTHADMIN=YES is specified in the LIBNAME statement that is referenced in the request or is invalid.

- The correct password for the target metadata-bound library is not otherwise available.

Requirement The AUTHADMIN option must be set to YES for this option to have an effect. However, the use of AUTHAMDIN=YES does not require that you use AUTHPW. You are not required to specify metadata-bound library passwords in a LIBNAME statement.

Interactions If the metadata-bound library has two or three distinct passwords, you must specify each individual password with the AUTHALTER=, AUTHREAD=, and AUTHWRITE= options as appropriate instead of using the AUTHPW= option on its own.

You can use the AUTHALTER= option in the same way as the AUTHPW= option if all three of the passwords (ALTER, READ, and WRITE) are the same and you are in a SAS language context where ALTER= can be used.

Tip An error occurs if the AUTHPW password does not match the password that is within the referenced secured library object.

See *SAS Guide to Metadata-Bound Libraries*

AUTHREAD=*read-password*

Specifies a READ password to use only in data access requests where both of these conditions exist:

- AUTHADMIN=YES is specified in the LIBNAME statement that is referenced in the request.
- The correct password for the target metadata-bound library is not otherwise available or is invalid.

Requirement The AUTHADMIN option must be set to YES for this option to have an effect.

See *SAS Guide to Metadata-Bound Libraries*

AUTHWRITE=*write-password*

Assigns a WRITE password to a metadata-bound library that prevents users from writing to a library, unless they enter the password.

Requirement The AUTHADMIN option must be set to YES for this option to have an effect.

See *SAS Guide to Metadata-Bound Libraries*

COMPRESS= LIBNAME Statement Option

Controls the compression of observations in output SAS data sets for a SAS library.

Default: If the COMPRESS= data set option or LIBNAME statement option is not set, then the value of the COMPRESS= system option is used. The default value for the COMPRESS= system option is NO.

Interaction: For the COPY procedure, the default value CLONE uses the compression attribute from the input data set for the output data set instead of the value that is specified in the COMPRESS= option. For more information about CLONE and NOCLONE, see the COPY Statement in the DATASETS procedure. This interaction does not apply when using SAS/SHARE or SAS/CONNECT.

Example: "Examples: Compression " on page 159

Syntax

COMPRESS=NO | CHAR | BINARY

Syntax Description

NO
specifies that the observations in a newly created SAS data set be uncompressed (fixed-length records).

CHAR
specifies that the observations in a newly created SAS data set be compressed (varying-length records) by SAS using RLE (Run Length Encoding). RLE compresses observations by reducing repeated consecutive characters (including blanks) to two-byte or three-byte representations.

Alias YES

Tip Use this compression algorithm for character data.

BINARY
specifies that the observations in a newly created SAS data set be compressed (variable-length records) by SAS using RDC (Ross Data Compression). RDC combines run-length encoding and sliding-window compression to compress the file.

Tip This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric variables). Because the compression function operates on a single record at a time, the record length needs to be several hundred bytes or larger for effective compression.

Details

For information about compression and examples, see Chapter 12, "Compression," on page 157.

Comparisons

The LIBNAME statement option takes precedence over the system option.

The data set option takes precedence over the LIBNAME statement option and the system option.

See Also

Data Set Options:

- "COMPRESS= Data Set Option" in *SAS Data Set Options: Reference*

System Options:

- "COMPRESS= System Option" in *SAS System Options: Reference*

CVPBYTES=, CVPENGINE=, CVPFORMATWIDTH=, CVPMULTIPLIER= LIBNAME Statement Options

Reduce the risk of truncation when character variables are transcoded.

See: *SAS National Language Support (NLS): Reference Guide*

Example: "Example: Avoid Truncation When Copying a SAS Library" on page 232

Syntax

LIBNAME *libref* '*SAS-library*' <CVPBYTES=*bytes*> CVPENGINE=V9 <CVPFORMATWIDTH=YES | NO> <CVPMULTIPLIER=*multiplier*>;

Syntax Description

CVPBYTES=*bytes*

specifies the number of bytes by which to expand character variable lengths when processing a SAS data file that requires transcoding. The CVP engine expands the lengths so that character data truncation does not occur. The lengths for character variables are increased by adding the specified value to the current length. You can specify a value from 0 to 32,766.

For example, the following LIBNAME statement implicitly assigns the CVP engine by specifying the CVPBYTES= option:

```
libname expand 'SAS data-library' cvpbytes=5;
```

Character variable lengths are increased by adding 5 bytes. A character variable with a length of 10 is increased to 15, and a character variable with a length of 100 is increased to 105.

Default If you specify CVPBYTES=, SAS automatically uses the CVP engine to expand the character variable lengths according to your specification. If you explicitly assign the CVP engine but do not specify either CVPBYTES= or CVPMULTIPLIER=, then SAS uses CVPMULTIPLIER=1.5 to increase the lengths of the character variables.

Restrictions The CVP engine supports SAS data files, no SAS views, catalogs, item stores, and so on.

The CVP engine is available for input (read) processing only.

For library concatenation with mixed engines that include the CVP engine, only SAS data files are processed. For example, if you execute the COPY procedure, only SAS data files are copied.

Requirement The number of bytes that you specify must be large enough to accommodate any expansion. Otherwise, truncation occurs, which results in an error message in the SAS log.

Interaction You cannot specify both the CVPBYTES= option and the CVPMULTIPLIER= option. Specify only one of these options.

See "Avoiding Character Data Truncation By Using the CVP Engine" in *SAS National Language Support (NLS): Reference Guide*

CVPENGINE=V9

specifies the V9 engine to process a SAS data file that requires transcoding. The CVP engine expands the character variable lengths before transcoding so that character data truncation does not occur. Then the specified engine processes the actual file.

Alias CVPENG

Default SAS uses the default SAS engine.

CVPFORMATWIDTH=YES | NO

specifics whether to expand the character format width.

If CVPVARCHAR= is not specified, the new format width is determined by the CVPMULTIPILER= and CVPBYTES= options.

If CVPVARCHAR= is specified, the CVP engine automatically adjusts the format width to meet the maximum-byte length of a converted character variable. For example, in a UTF-8 session, the format width is multiplied by 4.

Alias CVPFMTW

Default YES

CVPMULTIPLIER=*multiplier*

specifies a multiplier value that expands character variable lengths when you are processing a SAS data file that requires transcoding. The CVP engine expands the lengths so that character data truncation does not occur. The lengths for character variables are increased by multiplying the current length by the specified value. You can specify a multiplier value from 1 to 5 or you can specify 0 and then the CVP engine determines the multiplier automatically.

For example, the following LIBNAME statement implicitly assigns the CVP engine by specifying the CVPMULTIPLIER= option:

```
libname expand 'SAS data-library' cvpmultiplier=2.5;
```

Character variable lengths are increased by multiplying the lengths by 2.5. A character variable with a length of 10 is increased to 25, and a character variable with a length of 100 is increased to 250.

Alias CVPMULT

Default If you specify the CVPMULTIPLIER= option, SAS automatically uses the CVP engine to expand the character variable lengths according to your specification. If you explicitly specify the CVP engine but do not specify either the CVPMULTIPLIER= option or the CVPBYTES= option, then SAS uses CVPMULTIPLIER=AUTO(0) to increase the lengths. AUTO(0) sets the value of the CVP engine based on the encoding of the SAS session and input data set.

Restrictions The CVP engine supports SAS data files, no SAS views, catalogs, item stores, and so on.

The CVP engine is available for input (read) processing only.

For library concatenation with mixed engines that include the CVP engine, only SAS data files are processed. For example, if you execute the COPY procedure, only SAS data files are copied.

Requirement The number of bytes that you specify must be large enough to accommodate any expansion. Otherwise, truncation occurs, which results in an error in the SAS log.

Interaction You cannot specify both the CVPMULTIPLIER= option and the CVPBYTES= option. Specify only one of these options.

See "Avoiding Character Data Truncation By Using the CVP Engine" in *SAS National Language Support (NLS): Reference Guide*

EXTENDOBSCOUNTER= LIBNAME Statement Option

Specifies whether to extend the maximum observation count in output SAS data sets for a SAS library.

Alias: EOC=

Defaults: For Windows and z/OS, the default is YES.
For UNIX, the default is not set.

Restrictions: Use with output data sets only.
Use with the V9 engine only.
If you copy a file, the extended observation count attribute is not inherited.

See: "Understanding the Observation Count" on page 44

Syntax

EXTENDOBSCOUNTER=YES | NO

Syntax Description

YES
specifies to use the default maximum observation count of 2^{63}–1 or approximately 9.2 quintillion observations in a newly created SAS data set.

Restriction A SAS data set that has the extended observation count attribute cannot be used by releases prior to SAS 9.3. To remove the extended observation count attribute, the file must be re-created. If you plan to use the file in SAS 9.2 or earlier releases, then set EXTENDOBSCOUNTER=NO when you create the file.

Interaction EXTENDOBSCOUNTER=YES is ignored if the data representation is a 64-bit UNIX operating environment.

NO
is a compatibility setting for a newly created SAS data set, to enable its use in releases prior to SAS 9.3.

Details

The behavior depends on your operating environment:

- Under UNIX environments, by default the EXTENDOBSCOUNTER= option is not set. The extended observation count feature is not necessary under 64-bit UNIX. However, if you specify the OUTREP= option, and the data representation is not a 64-bit UNIX operating environment, then SAS automatically sets EXTENDOBSCOUNTER=YES. SAS adds the extended observation count feature for compatibility with environments other than UNIX where it might be necessary. (Many customers specify OUTREP= when they create a table for use in a different environment. This practice can help you avoid the limitations of CEDA processing.)
- Under Windows and z/OS, by default EXTENDOBSCOUNTER=YES. Files are created with the enhanced file format and the extended observation count attribute.

Note that backward compatibility is an issue only if the file is used in SAS 9.2 or earlier releases:

- Under UNIX, if you specify OUTREP= and plan to use the file in SAS 9.2 or earlier releases, specify EXTENDOBSCOUNTER=NO. If you do not specify OUTREP=, then you do not need to specify EXTENDOBSCOUNTER=NO.
- Under Windows and z/OS, if you plan to use the file in SAS 9.2 or earlier releases, specify EXTENDOBSCOUNTER=NO.

See Also

Data Set Options:

- "EXTENDOBSCOUNTER= Data Set Option" in *SAS Data Set Options: Reference*

System Options:

- "EXTENDOBSCOUNTER= System Option" in *SAS System Options: Reference*

INENCODING=, OUTENCODING= LIBNAME Statement Options

Overrides and changes the encoding when reading or writing SAS data sets in the SAS library.

See: *SAS National Language Support (NLS): Reference Guide*

Syntax

INENCODING=ANY | ASCIIANY | EBDICANY | *encoding-value*

Syntax Description

ANY
specifies no transcoding between ASCII and EBCDIC encodings.

NOTE: ANY is a synonym for binary. Because the data is binary, the actual encoding is irrelevant.

ASCIIANY
specifies that no transcoding occurs, assuming that the mixed encodings are ASCII encodings.

EBCDICANY
specifies that no transcoding occurs, assuming that the mixed encodings are EBCDIC encodings.

encoding-value
specifies an encoding value. For a list of encoding values, see "Encoding Values in SAS Language Elements" in *SAS National Language Support (NLS): Reference Guide*.

Details

The INENCODING= option is used to read SAS data sets in the SAS library. The OUTENCODING= option is used to write SAS data sets in the SAS library.

The INENCODING= value or the OUTENCODING= value is written to the SAS log when you use the LIST argument.

INENCODING= and OUTENCODING= are most appropriate when using an existing library that contains mixed encodings. To read a library that contains mixed encodings, you can set INENCODING= to ASCIIANY or EBCDICANY. To write a separate data set, you can use OUTENCODING= to specify a specific encoding, which is applied to the data set when it is created.

Comparisons

- Session encoding is specified using the ENCODING= system option.
- You can specify the encoding for reading data sets in a SAS library by using the LIBNAME statement INENCODING= option for input files. If both the LIBNAME statement option and the ENCODING= data set option are specified, SAS uses the data set option.
- You can specify the encoding for writing data sets to a SAS library by using the LIBNAME statement OUTENCODING= option for output files. If both the LIBNAME statement option and the ENCODING= data set option are specified, SAS uses the data set option.
- For the COPY procedure, the default CLONE option uses the encoding attribute of the input data set instead of the encoding value specified on the OUTENCODING= option. For more information about CLONE and NOCLONE, see COPY Statement.

  Note: This interaction does not apply when using SAS/CONNECT or SAS/SHARE.

See Also

- "Overview: Encoding for NLS" in *SAS National Language Support (NLS): Reference Guide*

System Options:

- "ENCODING System Option: UNIX, Windows, and z/OS" in *SAS National Language Support (NLS): Reference Guide*
- "LOCALE System Option" in *SAS National Language Support (NLS): Reference Guide*

Data Set Options:

- "ENCODING= Data Set Option" in *SAS National Language Support (NLS): Reference Guide*

LIBRARYDEFINITION= LIBNAME Statement Option

Retrieves pre-defined engine LIBNAME information from the Data Sources microservice and performs a LIBNAME assignment.

Alias: LIBDEF=

Restriction: This option is available in SAS Viya 3.4 and later.

Requirement: LIBDEF= must be the first option specified after the libref.

Syntax

LIBRARYDEFINITION=*source-definition-URI*

Syntax Description

source-definition-URI

retrieves pre-defined engine LIBNAME information from the Data Sources microservice and performs a LIBNAME assignment using that information.

Here is an example:

```
libname x
   LIBDEF="/dataSources/providers/Compute/
      sourceDefinitions/a22d983d-c324-442c-a605-06758af9aa6d"
   access=readonly;
```

Note: The SERVICESBASEURL= system option must be set during the invocation of your SAS session. This option specifies the host and port for file service requests. The access method fails at assignment time if the SERVICESBASEURL= option is not specified. For more information, see "SERVICESBASEURL= System Option" in *SAS System Options: Reference*.

Note: The SAS_VIYA_TOKEN environment variable must be defined. For more information about defining this variable, see "SAS_VIYA_TOKEN Environment Variable" in *Encryption in SAS*.

OUTREP= LIBNAME Statement Option

Specifies the data representation for the SAS library, which is the form in which data is stored in a particular operating environment.

Interactions: By default, PROC COPY uses the data representation of the file from the source library. If, instead, you want to use the data representation of the current SAS session, specify the NOCLONE option. If you want to use a different data representation, specify the NOCLONE option and the OUTREP= option. When you use PROC COPY with SAS/SHARE or SAS/CONNECT, the default behavior is to use the data representation of the current SAS session. For more information about CLONE and NOCLONE, see the COPY Statement in the *Base SAS Procedures Guide*.

The COPY procedure (with NOCLONE) and the MIGRATE procedure can use the LIBNAME option OUTREP= for DATA, VIEW, ACCESS, MDDB, and DMDB member types. Otherwise, only DATA member types are affected by the OUTREP= LIBNAME option.

When you use the OUTREP= LIBNAME statement option, the default encoding is based on the operating environment that is represented by the OUTREP= value and the locale of the current SAS session. To assign a nondefault encoding such as UTF-8, you must also specify the OUTENCODING= LIBNAME statement option. For more information about locale and encoding, see *SAS National Language Support (NLS): Reference Guide*.

Transcoding could result in character data loss when encodings are incompatible. For more information, see *SAS National Language Support (NLS): Reference Guide*.

If you specify the OUTREP= option, and you plan to use the data set in an earlier release of SAS, you might also want to specify the EXTENDOBSCOUNTER= option. See "EXTENDOBSCOUNTER= LIBNAME Statement Option" on page 30.

Syntax

OUTREP=*format*

Syntax Description

format

specifies the data representation, which is the form in which data is stored in a particular operating environment. Different operating environments use different standards or conventions for storing data.

- Floating-point numbers can be represented in IEEE floating-point format or IBM floating-point format.
- Data alignment can be on a 1-byte, 4-byte, or 8-byte boundary, depending on data type requirements for the operating environment.
- Data type lengths can be 8 bits or more for a character data type, 16 bit, 32 bit, or 64 bit for an integer data type, 32 bit for a single-precision floating-point data type, and 64 bit for a double-precision floating-point data type.
- The ordering of bytes can be big Endian or little Endian.

By default, SAS creates a new SAS data set by using the data representation of the CPU that is running SAS. Specifying the OUTREP= option enables you to create a SAS data set with a different data representation. For example, in a UNIX environment, you can create a SAS data set that uses a Windows data representation. For more information about compatibility and data representation, see "Cross-Environment Data Access" in *SAS Programmer's Guide: Essentials*.

Values for OUTREP= are listed in this table.

Table 4.1 Data Representation Values for OUTREP= Option

OUTREP= Value	Alias[1]	Environment
ALPHA_VMS_64		OpenVMS Alpha
HP_IA64	HP_ITANIUM	HP-UX for the Itanium Processor Family Architecture
HP_UX_32	HP_UX	HP-UX for PA-RISC
HP_UX_64		HP-UX for PA-RISC, 64-bit
LINUX_32	LINUX	Linux for Intel architecture
LINUX_X86_64		Linux for x64
LINUX_POWER_64		Linux on the Power Architecture[2]
MIPS_ABI		MIPS ABI
MVS_32	MVS	31-bit SAS on z/OS
MVS_64_BFP		64-bit SAS on z/OS
RS_6000_AIX_32	RS_6000_AIX	AIX
RS_6000_AIX_64		AIX
SOLARIS_32	SOLARIS	Solaris for SPARC
SOLARIS_64		Solaris for SPARC
SOLARIS_X86_64		Solaris for x64
VMS_IA64		OpenVMS on HP Integrity
WINDOWS_32	WINDOWS	32-bit SAS on Microsoft Windows
WINDOWS_64		64-bit SAS on Microsoft Windows (for both Itanium-based systems and x64)

1 It is recommended that you use the current values. The aliases are available for compatibility only.

2 LINUX_POWER_64 is added in SAS Viya 3.5. It is not supported in SAS 9.

See Also

Data Set Options:

- "OUTREP= Data Set Option" in *SAS Data Set Options: Reference*

POINTOBS= LIBNAME Statement Option

Specifies whether SAS creates compressed data sets whose observations can be randomly accessed or sequentially accessed.

Default: YES

Syntax

POINTOBS=YES | NO

Syntax Description

YES
produces a compressed data set that might be randomly accessed by observation number. This is the default.

Tip Specifying POINTOBS=YES does not affect the efficiency of retrieving information from a data set. It does increase CPU usage by approximately 10% when creating a compressed data set and when updating or adding information to it.

NO
suppresses the ability to randomly access observations in a compressed data set by observation number.

Tip Specifying POINTOBS=NO is desirable for applications where the ability to point directly to an observation by number within a compressed data set is not important. If you do not need to access data by observation number, then you can improve performance by approximately 10% by specifying POINTOBS=NO when creating a compressed data set or when updating or adding observations to it.

Comparisons

The data set option takes precedence over the LIBNAME statement option.

See Also

Data Set Options:

- "POINTOBS= Data Set Option" in *SAS Data Set Options: Reference*

REPEMPTY= LIBNAME Statement Option

Controls replacement of a data set when the new one is empty.

Default: YES

Interaction: If REPLACE=NO, the REPEMPTY= option is ignored.

Syntax

REPEMPTY=YES | NO

Syntax Description

YES
specifies that a new, empty data set can replace an existing data set with the same name.

Tip To avoid overwriting existing data sets with new, empty ones that are created by mistake, set REPEMPTY=NO.

NO
specifies that a new, empty data set cannot replace an existing data set with the same name.

Details

The following example code shows a common syntax error that creates an empty data set. If `mylib.test` exists already and the defaults, REPLACE=YES and REPEMPTY=YES, are in effect, then the existing data set is replaced by a new, empty data set.

```
data mylib.test;
run;
```

To prevent this error, set REPEMPTY=NO.

```
WARNING: Data set MYLIB.TEST was not replaced because REPEMPTY=NO and the
replacement file is empty.
```

Comparisons

The data set option takes precedence over the LIBNAME statement option.

See Also

Data Set Options:

- "REPEMPTY= Data Set Option" in *SAS Data Set Options: Reference*

PART 3

Accessing V9 Engine Data

5

SAS Data Sets

Parts of a SAS Data Set

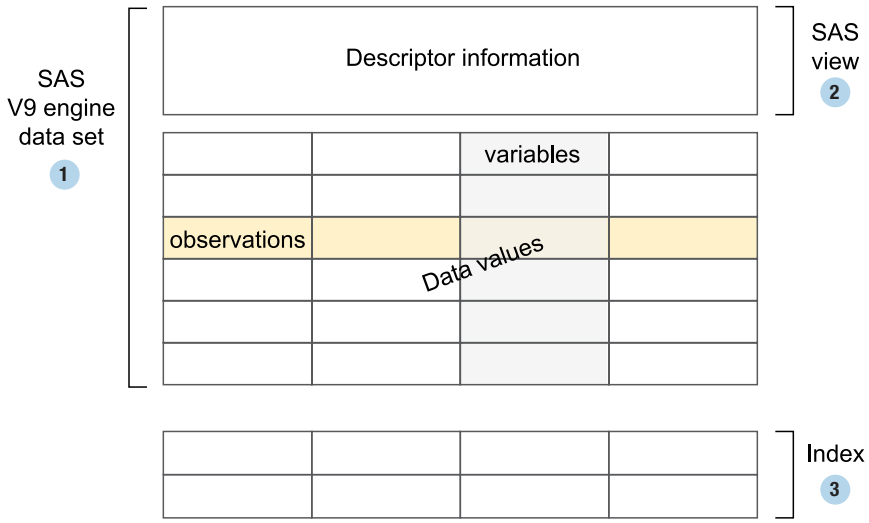
A SAS data set is a tabular collection of data (as variables and observations) whose contents are in a SAS file format. A SAS data set includes metadata that describes its attributes.

The metadata (also known as descriptor information) make the file self-documenting. SAS can obtain the attributes directly from the data set.

Descriptor information includes the number of observations, the observation length, the date on which the data set was last modified, and other facts. Descriptor information for individual variables includes attributes such as name, type, length, format, label, and whether the variable is indexed.

The following figure illustrates the logical components of a SAS data set.

Figure 5.1 *Logical Components of a SAS Data Set or View*



1. A V9 engine data set (member type DATA) contains descriptor information and data values.
2. In comparison, a SAS view (member type VIEW) contains descriptor information but not data. See Chapter 6, “SAS Views,” on page 49.
3. An index is a separate file that you can create for a SAS data set in order to provide direct access to specific observations. The index file has the same name as the data set and a member type of INDEX. (In some operating environments, the index is part of the data set, not a separate file.) See Chapter 8, “Indexes,” on page 69.

   Similarly, audit trails and extended attributes are other data set attributes that are stored as separate files. See Chapter 10, “Audit Trails,” on page 127 and Chapter 15, “Extended Attributes,” on page 197.

Understanding the Observation Count

Definition of Observation Count

The observation count includes both observations (rows) and deleted observations. The maximum number of observations that can be counted for a SAS data set is $2^{63}-1$ or approximately 9.2 quintillion observations. Exceeding that number is extremely unlikely for most users.

Backward Compatibility of the Extended Observation Count Attribute

Overview of the Extended Observation Count Attribute

In SAS 9.2 and earlier releases, the maximum observation count was much lower under some operating environments. In SAS 9.3, an extended observation count was offered to users with the option EXTENDOBSCOUNTER=YES. In SAS 9.4 and later releases, and in SAS Viya, the observation count is extended by default, and SAS has the same maximum observation count under all environments.

The behavior of EXTENDOBSCOUNTER= depends on your operating environment:

- Under UNIX environments, by default the EXTENDOBSCOUNTER= option is not set. The extended observation count feature is not necessary under 64-bit UNIX. However, if you specify the OUTREP= option, and the value is not a 64-bit UNIX operating environment, then SAS automatically sets EXTENDOBSCOUNTER=YES. SAS adds the extended observation count feature for compatibility with other environments. (Many customers specify OUTREP= when they create a data set for use in a different environment. This practice can help you avoid the limitations of CEDA processing.)
- Under Windows and z/OS, by default EXTENDOBSCOUNTER=YES. Files are created with the enhanced file format and the extended observation count attribute.

When to Use the EXTENDOBSCOUNTER=NO Option

The extended observation count attribute can make data sets unusable in SAS 9.2 and earlier releases, under certain circumstances. Here is an example of an error message that is produced when a feature is not supported in a previous release:

```
ERROR: File MYFILES.EXTEND.DATA not compatible with this SAS version.
```

The usage depends on your operating environment:

- Under UNIX, if you specify OUTREP= and plan to use the file in SAS 9.2 or earlier releases, specify EXTENDOBSCOUNTER=NO. If you do not specify OUTREP=, then you do not need to specify EXTENDOBSCOUNTER=NO.
- Under Windows or z/OS, if you plan to use the file in SAS 9.2 or earlier releases, specify EXTENDOBSCOUNTER=NO.

For more information, see the following:

- "EXTENDOBSCOUNTER= Data Set Option" in *SAS Data Set Options: Reference*
- "EXTENDOBSCOUNTER= LIBNAME Statement Option" on page 30
- "EXTENDOBSCOUNTER= System Option" in *SAS System Options: Reference*

Viewing the ExtendObsCounter Attribute in CONTENTS Output

When you run PROC CONTENTS or the CONTENTS statement of PROC DATASETS, you can see the **ExtendObsCounter** attribute in the output. If a SAS data set does not contain the extended observation count file attribute, the **ExtendObsCounter** field is not listed.

Output 5.1 *CONTENTS Procedure Output Showing ExtendObsCounter Attribute*

Engine/Host Dependent Information	
Data Set Page Size	65536
Number of Data Set Pages	1
First Data Page	1
Max Obs per Page	2039
Obs in First Data Page	2
Number of Data Set Repairs	0
ExtendObsCounter	YES
Filename	C:\My Documents\bigfile.sas7bdat
Release Created	9.0401M0
Host Created	X64_7PRO

Interactions with the Extended Observation Count

Note the following details:

- SAS functionality that copies files (such as the APPEND procedure, COPY procedure, MIGRATE procedure, and SET statement) does not copy the extended observation count attribute.
- In a SAS/SHARE client session, the EXTENDOBSCOUNTER= option in the LIBNAME statement is ignored if it is specified in combination with the SERVER= option.

Exceeding the Maximum Observation Count

SAS Processing When the Maximum Observation Count Is Reached

When a data set reaches the maximum observation count, continued SAS processing depends on whether the file has an index or an integrity constraint that uses an index.

- If the data set has an index or an integrity constraint that uses an index (unique key, primary key, and foreign key), an error message is issued.

  A data set is not damaged when an operation attempts to exceed the maximum observation count. However, you must take explicit action to continue processing the file. See "Recovering from an Exceeded Maximum Observation Count" on page 48.

- If the data set does not have an index or an integrity constraint that uses an index, sequential processing continues and additional observations are accepted. However, the file cannot store the observation count and does not maintain the observation numbers. Any operation that requires an observation number is not available. There are no messages to indicate that the file has reached or exceeded the maximum observation count.

  The following list describes some of the operations and features that are limited for a data set that exceeds the maximum observation count. This limited functionality is valid for a data set that does not have an index or an integrity constraint that uses an index. For a complete list, contact SAS Technical Support.

  - SAS procedures that return an observation count (such as the PRINT procedure or the CONTENTS procedure) return a missing value for the number of observations. A missing value is represented by a period (.).
  - SAS procedures that depend on the observation count (for example, the SORT procedure or the COMPARE procedure) can return unpredictable results.
  - Operations that update the observation count cannot be submitted. You cannot reset the observation count by deleting observations.
  - When you request to compress a file for which the observation count is no longer maintained, the compression percentage cannot be calculated.
  - You cannot create an index or an integrity constraint.

Recovering from an Exceeded Maximum Observation Count

Here are some ways to recover from an exceeded maximum observation count:

- You can delete the index or the integrity constraint and continue processing. However, because the file exceeds the maximum observation count, you have limited functionality. You can use the DATASETS procedure or the SQL procedure to delete indexes and integrity constraints. See the *Base SAS Procedures Guide* or the *SAS SQL Procedure User's Guide*.
- If the file was created in SAS 9.2 or earlier, the observation count might have been limited. Try re-creating the file in the current release to increase the maximum observation count.
- Deleted observations are included in the total observation count. If the data set contains deleted observations, try re-creating it. Use a method that does not retain deleted observations, such as the COPY procedure or the DATA step with a SET statement. (The MIGRATE procedure retains deleted observations.)

6

SAS Views

Definition of a SAS View

A *SAS view* is a virtual data set that extracts data values from other files in order to provide a customized and dynamic representation of the data.

- A view contains no data, but rather references data that is stored elsewhere.
- In most cases, you can use a SAS view as if it were a SAS data set. For example, you can reference a view in order to read data in DATA steps or PROC steps.
- The SAS member type is VIEW, and the file extension in most operating environments is .sas7bvew.

You can use SAS views as input under these conditions:

Here are the types of SAS views:

DATA step view
: is a stored DATA step program.

PROC SQL view
: is a stored query expression that is created in PROC SQL.

Comparing Views and Data Sets

The main difference between a SAS view and a SAS data set is where the data values are stored:

- A view contains metadata (descriptor information) and instructions for retrieving data, but a view does not store the data values.
- A data set contains both the metadata and the data values.

If you are deciding whether to create a data set or a view, consider the different features:

Table 6.1 *Differences between Views and Data Sets*

Feature	SAS View	SAS Data Set
Merge efficiency	One view can perform a multi-table join. With SAS/CONNECT, a view can join data sets that reside on different host computers.	Multiple DATA steps are required in order to merge data sets by common variables.
Disk space versus processing speed	A view does not store the underlying data, so processing speed could be affected.	Stores the full data for faster processing.
Data integrity	Data is dynamic. When you reference a view in a PROC step, the view executes and provides the data values as they currently exist in the underlying data.	Data is static.
Data preparation	Data is processed in its existing form during execution.	Variables can be sorted and indexed before they are used.
Separation of data from the data's consumers	A view can provide a prepackaged, custom perspective of the underlying data. The view's query can be modified without changing the data. For example, you could use existing data set variables to compute	A custom perspective could require a duplication of the data set. Modifying the data could require replacing the data set.

Feature	SAS View	SAS Data Set
	a new variable that does not need to be saved in the underlying data set.	
Updating data	Data that underlies a DATA step view is read-only. Data that underlies a PROC SQL view can be updated under specific conditions.	Data can be read and updated.
Storage on sequential-access devices, such as tape drives	Data cannot be stored on sequential-access devices because of their dynamic nature. Views must derive their information from data that is stored on random-access devices.	Data can be stored on sequential- or random-access devices.

The following data set features cannot be applied to views:

- indexes
- integrity constraints
- encryption
- compression
- audit trail
- generations

Comparing DATA Step Views and PROC SQL Views

If you are deciding whether to create a DATA step view or a PROC SQL view, consider the different features:

Table 6.2 *Differences between Views*

Feature	DATA Step View	PROC SQL View
DATA step programming	DATA step views support DATA step programming, including DO loops and IF-THEN/ELSE statements.	DATA step programming is not supported.
Updating the underlying data	Updating is not supported.	Updating is supported with some restrictions.

Feature	DATA Step View	PROC SQL View
Subsetting the data	The data is entirely loaded into memory before being subset by a WHERE clause.	The data can be subset by a WHERE clause before being loaded into memory. PROC SQL supports more types of WHERE clauses than are available in DATA step processing. A pass-through facility can send DBMS-specific SQL statements directly to a DBMS for execution. **Note:** SAS/ACCESS software is required in order to support SQL pass-through processing.
WHERE clause evaluation	The WHERE clause is evaluated by the DATA step view engine.	The WHERE clause might be evaluated by the PROC SQL view engine or by the underlying library's engine.
Data sources	SAS data sets DATA step views PROC SQL views DBMS data by using a SAS/ACCESS LIBNAME statement. raw data files (any file that can be read with an INPUT statement)	SAS data sets DATA step views PROC SQL views DBMS data by using a SAS/ACCESS LIBNAME statement (which can be embedded in the view) or SQL pass-through facility statements.

DATA Step Views

Definition of a DATA Step View

A DATA step view contains a stored DATA step program that can read from underlying data sources. Here are the commonly used underlying data sources:

- SAS data sets
- DATA step views
- PROC SQL views
- DBMS data by using a SAS/ACCESS LIBNAME statement
- raw data files (any file that can be read with an INPUT statement)

For several examples that demonstrate how to create and use DATA step views, see "Examples: SAS Views" on page 55.

Restrictions and Requirements

If you create a new view, and a view with the same name already exists in the library, then the old view is overwritten.

If you include global statements in a source program, SAS stores the DATA step view but not the global statements. Examples of global statements include the FILENAME, FOOTNOTE, LIBNAME, OPTIONS, and TITLE statements. When the view is referenced, execution can differ from usual DATA step processing.

When a view is created, the labels for the variables that it returns are also created. If the variable labels in the underlying data set are changed after the view is created, the labels in the view are not updated. Any procedure output shows the original labels. The view must be recompiled in order for the procedure output to reflect the new variable labels.

Performance Considerations

- DATA step code executes each time that you use a DATA step view, which might add considerable system overhead. In addition, you run the risk of having your data change between steps. However, this behavior also means that you read the most recent data, because you read the data when the view is executed.
- Depending on how many reads or passes on the data that are required, processing overhead increases in the following ways:
  - When one sequential pass is requested, no virtual data set is created. Compared to traditional methods of processing, making one pass improves performance by decreasing the number of input/output operations and elapsed time.
  - When random access or multiple passes are requested, the DATA step view must build a spill file. The spill file contains all generated observations so that subsequent passes can read the same data that was read by previous passes. You might be able to use the view SPILL= data set option to reduce the size of a spill file. See "SPILL= Data Set Option" in *SAS Data Set Options: Reference*.
- The VBUFSIZE= system option and the OBSBUF= data set option can be used to speed up execution time when processing DATA step views. For information about optimizing performance with SAS views, see "Setting VBUFSIZE= and OBSBUF= for SAS DATA Step Views" in *SAS Programmer's Guide: Essentials*.

  For more information about the VBUFSIZE= system option, see "VBUFSIZE= System Option" in *SAS System Options: Reference*. For more information about the OBSBUF data set option, see "OBSBUF= Data Set Option" in *SAS Data Set Options: Reference*.

Comparing DATA Step Views and Stored, Compiled DATA Step Programs

DATA step views differ from stored, compiled DATA step programs in the following ways:

- A DATA step view is implicitly executed when it is referenced as an input data set by another DATA or PROC step. Its main purpose is to provide data, one record at a time, to the invoking procedure or DATA step.
- A stored, compiled DATA step program is explicitly executed when it is specified by the PGM= option in a DATA statement. Its purpose is usually a more specific task, such as creating SAS data sets or originating a report. The DATA step program is compiled when it is stored, which reduces processing when it is executed.
- You can use the REDIRECT statement when you execute a stored, compiled DATA step. You cannot use this statement with DATA step views.

See also Chapter 14, “Stored, Compiled DATA Step Programs,” on page 187.

PROC SQL Views

Definition of a PROC SQL View

A PROC SQL view is a PROC SQL query expression that is named and stored for later use. When you use a PROC SQL view in a SAS program, the view derives its data from the data sets (or tables) or views that are listed in its FROM clause. The data that is accessed by the view is a subset or superset of the data in its underlying data sets or views.

A PROC SQL view can read or write data from the following data sources:

- SAS data sets
- DATA step views
- PROC SQL views
- DBMS data by using a SAS/ACCESS LIBNAME statement, which can be embedded in the view, or SQL pass-through facility statements

Creating and Using PROC SQL Views

See the following examples:

- "Example: Create a PROC SQL View" on page 56
- "Example: View a Subset of a DICTIONARY Table" on page 62
- "Example: Return a View's Source Code by Using the DESCRIBE Statement" on page 63

For complete documentation on how to create and use PROC SQL views, see "Creating and Using PROC SQL Views" in *SAS SQL Procedure User's Guide*.

Examples: SAS Views

Example: Create a DATA Step View

Example Code

This example creates a DATA step view from a `sashelp` data set.

```
libname myfiles v9 'c:\examples';
data myfiles.classsubset / view=myfiles.classsubset; /* 1 */
   set sashelp.class (keep=name age);                /* 2 */
   where age gt 14;                                  /* 3 */
run;
proc print data=myfiles.classsubset noobs;           /* 4 */
run;
```

1. The VIEW= option in the DATA statement creates a view named `myfiles.classsubset`.
2. Only the variables `name` and `age` are kept from the `sashelp.class` data set.
3. Only test subjects whose `age` is greater than 14 are included in the view. If the age requirement changes, you can modify this WHERE clause without affecting the underlying data set.
4. The PRINT procedure executes the view.

The PROC PRINT output shows test subjects who are over the age of 14.

Output 6.1 *MyFiles.ClassSubset DATA Step View*

Name	Age
Janet	15
Mary	15
Philip	16
Ronald	15
William	15

Key Ideas

- VIEW= is a DATA statement option, not a data set option.
- A view can be referenced like a data set in DATA steps or PROC steps.
- When you create a view, you store the query language and not the data. You can change the query without changing the stored data.
- A view is not executed when it is created. A view is executed when it is referenced by another DATA step or procedure.

See Also

- VIEW= option in the DATA statement
- Chapter 6, “SAS Views,” on page 49

Example: Create a PROC SQL View

Prerequisites

First, create or locate example data in your DBMS. The following example code creates the Teradata table named Grades. To use your DBMS, modify the LIBNAME statement. If you have not licensed a SAS/ACCESS interface to a DBMS, then modify the LIBNAME statement to create a SAS data set instead.

```
libname mytddata teradata server=mytera user=myid password=mypw;
data mytddata.grades;
   input student $ test1 test2 final;
   datalines;
Fred 66 80 70
Wilma 97 91 98
;
```

Example Code

This example uses the V9 engine to create a PROC SQL view that references the Teradata table.

```
libname myfiles v9 'c:\examples';          /* 1 */
proc sql;
   create view myfiles.highgrades as       /* 2 */
      select *                             /* 3 */
      from mytddata.grades                 /* 4 */
      where final gt 90;                   /* 5 */
quit;
proc print data=myfiles.highgrades noobs; /* 6 */
run;
```

1. The LIBNAME statement assigns the `myfiles` libref to a physical location and specifies the V9 engine.
2. The CREATE VIEW statement creates the view `myfiles.highgrades`.
3. All the columns are selected.
4. The columns are selected from the Teradata table `mytddata.grades`. The `mytddata` libref was assigned in Prerequisites on page 56.
5. The view includes rows where the value of the `final` variable is greater than 90.
6. PROC PRINT executes the view.

The PROC PRINT output shows that Wilma's `final` grade is greater than 90. Fred's `final` grade is not greater than 90, so Fred is not included in the output.

Output 6.2 *MyFiles.HighGrades PROC SQL View*

student	test1	test2	final
Wilma	97	91	98

Key Ideas

- A PROC SQL view can reference a table that is stored in a DBMS. You must license the appropriate SAS/ACCESS product.
- When a view's underlying data is stored in a DBMS, you need two LIBNAME statements. Use a SAS/ACCESS LIBNAME statement to reference the DBMS and a V9 engine LIBNAME statement to create the view.

See Also

- "CREATE VIEW Statement" in *SAS SQL Procedure User's Guide*
- "PROC SQL Views" on page 54
- "Comparing DATA Step Views and PROC SQL Views" on page 51

Example: Use a View to Merge Data and Calculate a Variable

Example Code

This example uses a DATA step view to merge multiple data sets without creating a new output data set. The combined data can be used in subsequent applications.

```
libname myfiles v9 'c:\examples';

data myfiles.sales / view=myfiles.sales;             /* 1 */
   merge sashelp.nvst1(rename=(amount=amount1))      /* 2 */
         sashelp.nvst2(rename=(amount=amount2))
         sashelp.nvst3(rename=(amount=amount3))
         sashelp.nvst4(rename=(amount=amount4));
      by date;                                       /* 3 */
      total = amount1 + amount2 + amount3 + amount4; /* 4 */
run;

proc print data=myfiles.sales;
run;
```

1. The DATA step defines a view named `sales` in the `myfiles` library.
2. The MERGE statement merges four data sets from the `sashelp` library. Each data set contains an `amount` variable. The RENAME= data set option ensures that the variables are distinct in the merged `sales` view.
3. The data is merged by the `date` variable.
4. The variable `total` does not exist in the underlying data sets. Its value is computed for each date.

The PROC PRINT output shows the merged data with the value of the computed `total` variable.

Output 6.3 *DATA Step View myfiles.sales*

Obs	DATE	amount1	amount2	amount3	amount4	total
1	01JAN1997	-30000	-60000	-20000	-40000	-150000
2	01JAN1998	7500	13755	5000	10000	36255
3	01JAN1999	7500	13755	5000	10000	36255
4	01JAN2000	7500	13755	5000	10000	36255
5	01JAN2001	7500	13755	5000	10000	36255
6	01JAN2002	7500	23755	5000	20000	56255

Key Ideas

- Using a view saves disk space because you do not duplicate data from the underlying data sets.
- A view can dynamically compute the value of a variable that does not exist in the underlying data set and does not need to be saved.

See Also

- "VIEW=view-name" in *SAS DATA Step Statements: Reference*
- "MERGE Statement" in *SAS DATA Step Statements: Reference*
- "BY Statement" in *SAS DATA Step Statements: Reference*
- For the syntax of calculations like `total = amount1 + amount2 + amount3 + amount4`, see topics about expressions and operators in *SAS Programmer's Guide: Essentials*

Example: Print the Current View of Changing Data

Prerequisites

First, create a text file that contains the following raw data. Name the file `pointslist.txt`.

```
frisbee    A01 009
gilhoolie  A01 085
merlbeest  A01 155
```

Example Code

A SAS view can respond dynamically to changes in external data. The following DATA step reads an external text file to create a view and a data set.

```
libname myfiles 'c:\examples';                      /* 1 */
filename student 'c:\examples\pointslist.txt';      /* 2 */

data myfiles.class(keep=name project points)
     myfiles.problems(keep=name project points code date)
     / view=myfiles.class;                          /* 3 */
   infile student;
   input name $ 1-10 project $ 12-14 points 16-18; /* 4 */

select;
   when (points>100 or points<30)                   /* 5 */
      do code=1;
         date=datetime();
         output myfiles.problems;
      end;
   otherwise                                        /* 6 */
      output myfiles.class;
end;
run;
```

1. The LIBNAME statement assigns the `myfiles` libref to the location for storing the data set and view.
2. The FILENAME statement assigns the `student` fileref to the location and name of the `pointslist.txt` file, which you created in Prerequisites on page 56.
3. The DATA step creates the `class` view and the `problems` data set. The view has three columns and the data set has five columns. The VIEW= option identifies `class` as a view.
4. The INFILE statement specifies the `student` fileref. The INPUT statement assigns column names to the raw data. The `name`, `project`, and `points` columns are used in the view and the data set.
5. The `code` and `date` columns are defined. These two columns are used in the data set only.

   When the `points` value is greater than 100 or less than 30, the observation is assigned an error code of `1` along with a SAS datetime. The information is stored in the data set.
6. The observations that do not have the specified errors are stored in the view.

If you check your file system at this point or run PROC DATASETS, you see the `myfiles.class` view but not the `myfiles.problems` data set. Because the DATA step view creates both the view and the data set, the data set is not created until the view is executed. In the following code, the first PROC PRINT executes the view and creates the data set.

```
proc print data=myfiles.class noobs;
run;
```

```
proc print data=myfiles.problems noobs;
   format date datetime18.;
run;
```

If you check your file system now, you can see files for both the view and the data set. Here is the output for the view and the data set:

Output 6.4 *DATA Step View myfiles.class*

name	project	points
gilhoolie	A01	85

Output 6.5 *Data Set myfiles.problems*

name	project	points	code	date
frisbee	A01	9	1	31OCT18:11:14:54
merlbeest	A01	155	1	31OCT18:11:14:54

Add Rows to the External File and Execute the DATA Step View Again

The data in the external file `pointslist.txt`, which is referenced by the fileref `student`, is updated frequently. Add these lines to the end of the `pointslist.txt` file that you created in Prerequisites on page 56:

```
xylotone   A02 100
zymeco     A02 010
```

Run the two PROC PRINT steps again. There is no need to run the DATA step again.

```
proc print data=myfiles.class noobs;
run;
proc print data=myfiles.problems noobs;
   format date datetime18.;
run;
```

If you check your file system or run PROC DATASETS, you can see that the `myfiles.problems` data set is newer than the `myfiles.class` view. The data set has one new observation, but the view has not changed because it does not contain data. Here is the output, which shows the added observations:

Output 6.6 *DATA Step View myfiles.class after External File Update*

name	project	points
gilhoolie	A01	85
xylotone	A02	100

Output 6.7 *Data Set myfiles.problems after External File Update*

name	project	points	code	date
frisbee	A01	9	1	31OCT18:11:33:27
merlbeest	A01	155	1	31OCT18:11:33:27
zymeco	A02	10	1	31OCT18:11:33:27

Key Ideas

- You can run a procedure or DATA step against a view at any time in order to access the latest data.
- If you specify additional data sets in the DATA statement, SAS creates these data sets when the view is executed in a subsequent DATA or PROC step.
- A DATA step view cannot contain global statements like FILENAME and LIBNAME. Place global statements outside the code that defines the view.

See Also

- "Comparing Views and Data Sets" on page 50
- "SELECT Statement" in *SAS DATA Step Statements: Reference*

Example: View a Subset of a DICTIONARY Table

Example Code

DICTIONARY tables are special read-only PROC SQL tables or views. Because the tables can be very large, this example shows how to create a subset.

```
proc sql;
   create view myview as                      /* 1 */
      select libname, memname, nobs, filesize /* 2 */
      from sashelp.vtable                     /* 3 */
      where memname contains "ZIP";           /* 4 */
   select * from myview;
quit;
```

1 PROC SQL creates a view named `myview`. No libref is specified, so the view is created in the Work library.

2 The `libname`, `memname`, `nobs`, and `filesize` variables are selected.

3 The data is selected from the `sashelp.vtable` view, which is a PROC SQL view of DICTIONARY.TABLES. This DICTIONARY table contains information about the data sets that are known in the current SAS session.

4 The WHERE statement specifies to search for data sets that contain `ZIP` in the name.

The PROC SQL output shows that one data set meets the conditions of the WHERE clause:

Output 6.8 *PROC SQL View myview Created from sashelp.vtable*

Library Name	Member Name	Number of Physical Observations	Size of File
SASHELP	ZIPCODE	41140	36274176

Key Ideas

- You can create a PROC SQL view with a WHERE clause to subset a large data set. SAS processes the WHERE clause before opening the data set that is referenced by the view.
- DICTIONARY tables and their PROC SQL views in Sashelp are automatic features of SAS. Use them to query information about data and the current SAS session.

See Also

- "SAS Dictionary Tables" in *SAS Programmer's Guide: Essentials*
- "Accessing SAS Information By Using DICTIONARY Tables" in *SAS SQL Procedure User's Guide*
- "PROC SQL Views" on page 54

Example: Return a View's Source Code by Using the DESCRIBE Statement

Example Code

This example submits the DESCRIBE statement in PROC SQL. First, create `myview` as shown in "Example: View a Subset of a DICTIONARY Table" on page 62.

```
proc sql;
```

```
   describe view myview;
quit;
```

The SAS log shows the view's definition in PROC SQL:

Example Code 6.1 *SAS Log Showing the View Definition in PROC SQL*

```
NOTE: SQL view WORK.MYVIEW is defined as:

        select libname, memname, nobs, filesize
          from SASHELP.VTABLE
         where memname contains 'ZIP';
```

This example submits the DESCRIBE statement in a DATA step. First, create `myfiles.classsubset` as shown in "Example: Create a DATA Step View" on page 55.

```
data view=myfiles.classsubset;
   describe;
run;
```

Example Code 6.2 *SAS Log Showing the View Definition in a DATA Step*

```
NOTE: DATA step view MYFILES.CLASSSUBSET is defined as:

data myfiles.classsubset / view=myfiles.classsubset ;
   set sashelp.class (keep=name age);
   where age gt 14;
run;
```

Key Ideas

- Use the DESCRIBE statement in PROC SQL for a PROC SQL view. Use the DESCRIBE statement in the DATA step for a DATA step view.
- Retrieving a view's source code can be helpful if you are not familiar with the data. In addition, you might want to submit the code with modifications to create another view of the data.

See Also

- "DESCRIBE Statement" in *SAS SQL Procedure User's Guide*
- "DESCRIBE Statement" in *SAS DATA Step Statements: Reference*

PART 4

Using V9 Engine Features

7

Introduction to Engine-Specific Features

What Are Engine-Specific Features?

This document covers features that are supported by the V9 engine only or have significant details that are specific to the engine. Most of Base SAS functionality can be used by multiple SAS engines. Those concepts and syntax are covered in a series of documents. For the list of documents, see the following topic, "Base SAS Usage and Syntax" on page 67.

Base SAS Usage and Syntax

See *SAS Programmer's Guide: Essentials* for examples and in-depth coverage of the features that are common to SAS engines.

For the most commonly used syntax, see these references:

- *Base SAS Procedures Guide*
- *SAS Data Set Options: Reference*
- *SAS Formats and Informats: Reference*
- *SAS DATA Step Statements: Reference*
- *SAS Global Statements: Reference*

- *SAS System Options: Reference*

For usage and syntax that is specific to the operating environment, see these references:

- *SAS Companion for UNIX Environments*
- *SAS Companion for Windows*
- *SAS Companion for z/OS*

8

Indexes

Definition of an Index

An index is an optional file that you can create for a SAS data set in order to provide direct access to specific observations. The index stores values in ascending order for a specific variable or variables and includes information about the location of those values within observations in the data set. In other words, an index enables SAS to locate an observation by value.

For example, suppose that you want the observation with the value of the `ssn` variable (Social Security number) equal to `123-45-6789`.

- Without an index, SAS accesses observations sequentially in the order in which they are stored in the data set. SAS reads each observation, looking for the `ssn` value of `123-45-6789` until all observations are read.
- With an index on the variable `ssn`, SAS accesses the observation directly. SAS meets the condition using the index and goes directly to the observation that contains the value, without having to read each observation.

Here are some essential concepts:

- You can create an index when you create a data set, or you can create an index for an existing data set.
- For each data set, you can create one or multiple indexes. For the V9 engine, in most operating environments, all the data set's indexes are stored in one file that has the same name as the data set. The member type is INDEX and the file extension in most operating environments is .sas7bndx. See "SAS Files and Member Types" on page 4.
- The data set can be compressed and encrypted.
- Once an index exists, SAS treats it as part of the data set. That is, if you add or delete observations or modify values, the index is automatically updated.

For the different ways to create or delete an index, see "Examples: Indexes" on page 93.

You can also create and delete indexes using other SAS utilities and products, such as SAS/CONNECT, SAS/IML, and SAS Component Language (SCL).

Benefits of an Index

In general, SAS can use an index to improve performance in the following situations, especially for a large data set:

- For WHERE processing, an index can provide faster and more efficient access to a subset of data. To process a WHERE expression, SAS by default decides whether to use an index or to read the data set sequentially.
- For BY processing, an index returns observations in the index order, which is in ascending value order. The SORT procedure is not required, even when the data set is not stored in the sorted order. (If you use PROC SORT to sort by the BY variable, then the index is not used.)
- For the SET and MODIFY statements, the KEY= option enables you to specify an index in a DATA step in order to retrieve particular observations in a data set.

In addition, an index can benefit other areas of SAS. In SCL, an index improves the performance of table lookup operations. For the SQL procedure, an index enables the software to process certain classes of queries more efficiently (for example, join queries). For SAS/IML, you can explicitly specify that an index be used for Read, Delete, List, or Append operations.

Even though an index can reduce the time required to locate a set of observations, there are costs associated with creating, storing, and maintaining the index. These costs are covered in "Deciding Whether to Create an Index" on page 74.

Note: An index is never used for the subsetting IF statement in a DATA step, or for the FIND and SEARCH commands in the FSEDIT procedure.

Types of Indexes

Overview of Types of Indexes

When you create an index, you designate which variable or variables to index. An indexed variable is called a key variable. You can create the following types of indexes:

- a simple index, which consists of the values of one variable
- a composite index, which consists of the values of more than one variable, with the values concatenated to form a single value

When you are deciding whether to create a simple index or a composite index, consider how you will access the data:

- If you usually access data for a single variable, a simple index is best.
- If you frequently access data for multiple variables, a composite index could be beneficial.
- If you frequently access data for multiple variables in the same WHERE expression, then a composite index can provide compound optimization.

When you create a simple or composite index, you can set the following conditions:

- limit an index (and its data set) to unique values
- exclude missing values from an index

The following topics show how the different types of indexes are used. For additional examples of the syntax, see "Examples: Indexes" on page 93.

Simple Index

The most common index is a simple index, which is an index of values for one key variable. The variable can be numeric or character. When you create a simple index, SAS uses the variable name as the index name.

The following example shows the DATASETS procedure statements that are used to create two simple indexes for variables `class` and `major` in the `college.survey` data set:

```
proc datasets library=college;
   modify survey;
      index create class;
      index create major;
run;
```

To process a WHERE expression using an index, SAS uses only one index. When the WHERE expression has multiple conditions using multiple key variables, SAS determines which condition qualifies the smallest subset. For example, suppose that `college.survey` contains the following data:

- 42,000 observations contain `class=12`
- 6,000 observations contain `major='Biology'`

The smaller subset is the index on the `major` variable. Therefore, SAS would select the index on `major` to process the following WHERE expression:

```
where class=12 and major='Biology';
```

Composite Index

A composite index is an index of two or more key variables with their values concatenated to form a single value. The variables can be numeric, character, or a combination. When you create a composite index, you must specify an index name.

The following example shows the DATASETS procedure statements that are used to create a composite index named `zipid` for the data set `college.maillist`. The composite index specifies two key variables: `zipcode` and `schoolid`.

```
proc datasets library=college;
   modify maillist;
      index create zipid=(zipcode schoolid);
run;
```

Often, only the first variable of a composite index is used. The following WHERE expression can use the `zipid` composite index because the variable `zipcode` is the first key variable in the index:

```
where zipcode = 78753;
```

You can take advantage of all key variables in a composite index by using compound optimization. The following WHERE expression can use the `zipid` composite index to satisfy all of the conditions with a single search of the index:

```
where zipcode = 78753 and schoolid = 55;
```

Unique Index

Often it is important to require that values for a variable be unique, such as Social Security numbers or employee numbers. When you create a simple or composite index, include the UNIQUE option. A unique index guarantees that values for the variable (or the combination of a composite group of variables) remain unique for every observation in the data set. If an update tries to add a duplicate value to that variable, the update is rejected.

The following example creates a simple index for the variable `idnum` and requires that all values for `idnum` be unique:

```
proc datasets library=college;
   modify student;
      index create idnum / unique;
run;
```

Index with No Missing Values

If a variable has a large number of missing values, you might want to keep them from using space in the index. Therefore, when you create an index, you can include the NOMISS option to specify that missing values are not maintained by the index.

The following example creates a simple index for the variable `status` and specifies that the index does not maintain missing values for the variable:

```
proc datasets library=college;
   modify student;
      index create status / nomiss;
run;
```

In contrast to the UNIQUE option, observations with missing values for the key variable can be added to the data set. However, the missing values are not added to the index.

If missing values are present in the data set, SAS does not use an index that was created with the NOMISS option to process the following code:

- a BY statement
- a WHERE expression that qualifies observations that contain missing values.

If there are no missing values, SAS considers using the index to process the BY statement or WHERE expression.

Deciding Whether to Create an Index

Introduction to Index Costs

An index exists to improve performance. However, an index conserves some resources at the expense of others. The following topics describe the costs associated with creating, using, and maintaining an index.

CPU Cost

CPU time is necessary to create an index. CPU time is also necessary to maintain the index. That is, for an indexed data set, when a value is added, deleted, or modified, it must also be added, deleted, or modified in the appropriate index or indexes.

Additional CPU time is necessary when SAS uses an index. Reading values with an index is a more complicated process than reading data sequentially. However, you also benefit from SAS reading only those observations that meet the conditions. CPU usage is especially high when a large number of observations meet the conditions.

To compare CPU usage with and without an index, for some operating environments, you can issue the STIMER or FULLSTIMER system options. These options write performance statistics to the SAS log. For documentation, see the list of operating environment companions in “Base SAS Usage and Syntax” on page 67.

I/O Cost

Using an index to read observations from a data set can increase the number of I/O (input/output) requests compared to reading the data set sequentially. For example, processing a BY statement with an index might increase I/O count, but you save in not having to issue the SORT procedure. For WHERE processing, SAS considers I/O count when deciding whether to use an index.

1. SAS does a binary search on the index file and positions the index to the first entry that contains a qualified value.
2. SAS uses the value's RID (identifier) to directly access the observation that contains the value. SAS transfers the observation between external storage to a buffer, which is the memory into which data is read or from which data is written. The data is transferred in pages, which is the amount of data (the number of observations) that can be transferred for one I/O request. (Each data set has a specified page size.)
3. SAS continues the process until the WHERE expression is satisfied. Each time SAS accesses an observation, the data set page that contains the observation must be read into memory if it is not already there. Therefore, if the observations are on multiple data set pages, an I/O operation is performed for each observation.

The result is that the more random the data, the more I/Os are required to use the index. If the data is ordered more like the index, which is in ascending value order, a smaller number of I/Os are required to access the data.

The number of buffers determines how many pages of data can simultaneously be in memory. In general, the larger the number of buffers, the smaller the number of I/Os that are required. For example, if the page size is 4096 bytes and one buffer is allocated, then one I/O transfers 4096 bytes of data (or one page). To reduce I/Os, you can increase the page size but you need a larger buffer. To reduce the buffer size, you can decrease the page size but you use more I/Os.

For information about data set characteristics like the data set page size and the number of data set pages, use the CONTENTS procedure (or use the CONTENTS statement in the DATASETS procedure). With this information, you can determine the data set page size and experiment with different sizes. Note that the information that is available from PROC CONTENTS depends on the operating environment.

The BUFSIZE= data set option (or system option) sets the permanent page size for a data set when it is created. The page size is the amount of data that can be transferred for an I/O operation to one buffer. The BUFNO= data set option (or system option) specifies how many buffers to allocate for a data set and for the overall system for a given execution of SAS. That is, BUFNO= is not stored as a data set attribute. See the following references:

- “BUFSIZE= System Option” in *SAS System Options: Reference*
- “BUFSIZE= Data Set Option” in *SAS Data Set Options: Reference*
- “BUFNO= System Option” in *SAS System Options: Reference*
- “BUFNO= Data Set Option” in *SAS Data Set Options: Reference*

Buffer Requirements

In addition to the resources that are used to create and maintain an index, SAS also requires additional memory for buffers when an index is used. Opening a data set opens the index file but none of the indexes. The buffers are not required unless SAS uses the index, but they must be allocated in preparation for the index that is being used.

The number of buffers that are allocated depends on the number of levels in the index tree and in the data set Open mode. If the data set is open for input, the

maximum number of buffers is three; for update, the maximum number is four. (Note that these buffers are available for other uses; they are not dedicated to indexes.)

The IBUFSIZE= system option specifies the page size on disk for an index file when it is created. The default setting causes SAS to use the minimum optimal page size for the operating environment. Typically, you do not need to specify an index page size. However, some situations could require a different page size. See "IBUFSIZE= System Option" in *SAS System Options: Reference*.

The IBUFNO= system option specifies an optional number of extra buffers to be allocated when navigating an index file. SAS automatically allocates a minimal number of buffers. Typically, you do not need to specify extra buffers. However, using IBUFNO= to specify extra buffers could improve execution time by limiting the number of input/output operations that are required for a particular index file. The improvement in execution time, however, comes at the expense of increased memory consumption. See "IBUFNO= System Option" in *SAS System Options: Reference*.

Disk Space Requirements

Additional disk space is required to store the index file. This file might show up as a separate file or appear to be part of the data set, depending on the operating environment.

For information about the index file size, use the CONTENTS procedure (or the CONTENTS statement in the DATASETS procedure). Note that the available information from PROC CONTENTS depends on the operating environment.

Guidelines for Creating Indexes

Data Set Considerations

- For a small data set, sequential processing is often just as efficient as index processing. Do not create an index if the data set page count is less than three pages. It would be faster to access the data sequentially. To see how many pages are in a data set, use the CONTENTS procedure (or use the CONTENTS statement in the DATASETS procedure). Note that the information that is available from PROC CONTENTS depends on the operating environment.
- Consider the cost of an index for a data set that is frequently updated. If a data set changes often, the overhead associated with updating the index can outweigh the processing advantages you gain from accessing the data with an index.
- Create an index when you intend to retrieve a small subset of observations from a large data set (for example, less than 25% of all observations). When this occurs, the cost of processing data set pages is lower than the overhead of

sequentially reading the entire data set. The smaller the subset, the larger the performance gains.

- To reduce the number of I/Os performed when you create an index, first sort the data by the key variable. Then to improve performance, maintain the data set in sorted order by the key variable. This technique reduces the I/Os by grouping like values together. That is, the more ordered the data set is with respect to the key variable, the more efficient the use of the index. If the data set has more than one index, sort the data by the most frequently used key variable.
- An index might not be necessary to optimize a WHERE expression if the data is sorted appropriately in order to satisfy the condition. To process a WHERE expression without an index, SAS first checks for the sort indicator that is stored with the file from a previous SORT procedure. If the sort indicator is appropriate, SAS stops reading the file once there are no more values that satisfy the WHERE expression. For example, consider a file that is sorted by `age`, without an index. To process the expression `where age le 25`, SAS stops reading observations after it finds an observation that is where `age` is greater than 25. Note that while SAS can determine when to stop reading observations, if there is no index, there is no indication where to begin. Without an index, SAS always begins with the first observation, which can require reading a lot of observations.

Index Use Considerations

- Keep the number of indexes per data set to a minimum in order to reduce disk storage and update costs.
- Consider how often your applications use an index. An index must be used often in order to make up for the resources that are used in creating and maintaining it.
- When you create an index to process a WHERE expression, do not try to create one index that can be used to satisfy all queries. If several variables are used in queries, those queries might be best satisfied with simple indexes on the most discriminating of those variables.

Choosing the Key Variables

The variable that is indexed is called the *key variable*.

- Choose a key variable that is used frequently in queries, and that selects a small subset from a large data set.
- Choose a key variable that most precisely identifies the observations that meet the conditions of the WHERE expression. That is, the variable should be *discriminating*, which means that the index should select the fewest possible observations. For example, a variable such as `firstname` would not be discriminating if a large percentage of the population has the same first name. A variable such as `lastname` could be a better choice, if few persons in the data share the same last name.
- When you create a composite index, the first key variable should be the most discriminating.

For example, consider a data set with variables `lastname` and `gender`.

- If many queries against the data set include `lastname`, then indexing `lastname` could be beneficial. Last names vary and are usually discriminating.
- Even if many queries include `gender`, an index on `gender` might not be a good choice. It is not a discriminating variable if half the population is male and half is female.
- If queries against the data set most often include both variables, as shown in the following WHERE expression, then creating a composite index on `lastname` and `gender` could improve performance.

```
where lastname='LeVoux' and gender='F';
```

Using an Index for WHERE Processing

How SAS Selects an Index for WHERE Processing

Overview of Index Selection

When you issue a WHERE expression, you use conditions to select observations for processing. Using an index to process a WHERE expression improves performance and is referred to as *optimizing* the WHERE expression.

To process a WHERE expression, by default SAS decides whether to use an index or read all the observations in the data set sequentially. To make this decision, SAS performs the following analysis:

1. Identifies the available index or indexes.
2. Estimates the number of observations that would be qualified. If multiple indexes are available, SAS selects the index that returns the smallest subset of observations.
3. Compares resource usage to decide whether it is more efficient to satisfy the WHERE expression by using the index or by reading all the observations sequentially.

Note: SAS considers several factors when deciding whether to use an index. Therefore, experimentation is the best way to determine the optimal performance. If you have a WHERE expression that is used repeatedly, compare the results using an index and without an index in order to determine which method provides the best performance. You can control index usage with the IDXWHERE= and IDXNAME= data set options. See “Controlling WHERE Processing Index Usage with Data Set Options” on page 84.

How SAS Identifies the Available Index or Indexes

The first step for SAS in deciding whether to use an index is to identify any key variables in the WHERE expression. (A key variable is an indexed variable.) Even though a WHERE expression can consist of multiple conditions that specify different variables, SAS uses only one index to process the WHERE expression. SAS selects the index that satisfies the most conditions and qualifies the fewest observations:

- Usually, SAS selects one condition. The variable that is specified in the condition has either a simple index or is the first key variable in a composite index. See "Writing a WHERE Expression for Optimization" on page 80.
- However, you can take advantage of multiple key variables in a composite index by constructing an appropriate WHERE expression. This practice is called *compound optimization*. See "Using Compound Optimization" on page 82.

How SAS Uses Centiles to Estimate the Number of Qualified Observations

After SAS identifies the index or indexes that can meet the conditions of the WHERE expression, SAS estimates the number of observations that will be qualified by an available index. When multiple indexes exist, SAS selects the one that seems to produce the fewest qualified observations.

SAS estimates the number of observations that will be qualified by using stored statistics called *cumulative percentiles* (or *centiles* for short). Centiles information represents the distribution of values in an index so that SAS does not have to assume a uniform distribution. To print centiles information for an indexed data set, include the CENTILES option in PROC CONTENTS (or in the CONTENTS statement in the DATASETS procedure).

SAS uses the centiles information in the following way:

- SAS uses an index if it estimates that the WHERE expression will select approximately one-third or less of the total number of observations in the data set.
- If the number of qualified observations is less than 3% of the data set (or if no observations are qualified), SAS automatically uses the index. In this case, SAS does not compare resource usage.
- If all the observations are qualified, by default SAS does not use the index unless the IDXNAME= or IDXWHERE= data set option is specified.

By default, SAS updates centiles information when 5% of values for the indexed variable have changed. You can specify a different percentage, or always, or never:

- When you create an index, you can specify the UPDATECENTILES= option in the INDEX CREATE statement in PROC DATASETS.

- To change the setting, specify the UPDATECENTILES= option in the INDEX CENTILES statement in PROC DATASETS.
- To update centiles information without changing the setting, specify the REFRESH option in the INDEX CENTILES statement in PROC DATASETS.

How SAS Compares Resource Usage

In some cases, the centiles information does not cause an automatic decision about whether to use the index or to read all of the observations sequentially. In these cases, SAS compares resource usage in the following way:

1. SAS predicts the number of I/Os that it would take to meet the conditions of the WHERE expression using the index. To accomplish the prediction, SAS positions the index to 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 (identifiers) on that index page are processed, indicating how many I/Os it takes to read the observations in the data set.

   If the observations are randomly distributed throughout the data set, the observations are located on multiple data set pages. This means that an I/O is needed for each page. Therefore, the more random the data in the data set, the more I/Os it takes to use the index. If the data in the data set is ordered more like the index, which is in ascending value order, a smaller number of I/Os are needed to use the index.
2. SAS calculates the I/O cost of a sequential pass of the entire data set.
3. SAS compares the two resource costs. The following factors affect the comparison:
   - size of the subset relative to the size of the data set
   - data set value order
   - data set page size
   - number of allocated buffers
   - cost to uncompress a compressed data set for a sequential read
4. SAS chooses the lowest resource cost. In the case of a tie, SAS chooses the index.

Writing a WHERE Expression for Optimization

SAS attempts to use an index for the following types of conditions:

Table 8.1 *WHERE Conditions That Can Be Optimized*

Condition	Valid for Compound Optimization	Examples

comparison operators, which include the EQ operator; directional comparisons like less than or greater than; and the IN operator	yes	`where empnum eq 3374;`
		`where empnum < 2000;`
		`where state in ('NC','TX');`

comparison operators with NOT	yes	`where empnum ^= 3374;` `where x not in (5,10);`
comparison operators with the colon modifier	yes	`where lastname gt: 'Sm';`
CONTAINS operator	no	`where lastname contains 'Sm';`
fully bounded range conditions specifying both an upper and lower limit, which includes the BETWEEN-AND operator	yes	`where 1 < x < 10;` `where empnum between 500 and 1000;`
pattern-matching operators LIKE and NOT LIKE	no	`where firstname like '%Rob_%'`
IS NULL or IS MISSING operator	no	`where name is null;` `where idnum is missing;`
TRIM function	no	`where trim(state)='Texas';`
SUBSTR (left of =) function in the form of: WHERE SUBSTR (*variable, position* <,*length*>)='*string*'; when the following conditions are met: *position* specifies a numeric constant for the beginning character position that is less than or equal to the *variable* length.	no	`where substr (month,4,5)='ember' and (city='Charleston' or city='Atlanta');`

Condition	Valid for Compound Optimization	Examples
length specifies a numeric constant for the length of *string*. The *length* plus *position* cannot be larger than the *variable* length plus 1.		

The following WHERE conditions cannot be optimized with an index:

- arithmetic operators
- a variable-to-variable condition
- the sounds-like operator
- any function other than the TRIM and SUBSTR function as listed in the preceding table

For examples of WHERE expressions that can use simple or composite indexes, see "Types of Indexes" on page 71.

For syntax, see the following resources:

- "WHERE Expressions" in *SAS Programmer's Guide: Essentials*
- "WHERE Statement" in *SAS DATA Step Statements: Reference*

Using Compound Optimization

Compound optimization is the process of optimizing multiple WHERE expression conditions by using one composite index. Compound optimization can greatly improve performance.

For example, suppose there is a composite index for `lastname` and `firstname`. If you execute the following WHERE expression, SAS uses the concatenated values for the first two variables, and then SAS further evaluates each qualified observation for the `empid` value:

```
where lastname eq 'Smith' and firstname eq 'John' and empid=3374;
```

For compound optimization to occur, all of the following conditions must be true.

- At least the first two key variables in the composite index must be used in valid WHERE expression conditions. For a list of conditions that are valid for compound optimization, see "Writing a WHERE Expression for Optimization" on page 80.
- At least one condition must use the EQ or IN operator. For example, you cannot have all range conditions.
- The conditions must be connected with the AND or the OR logical operator:
  - When conditions are connected with AND, the conditions can occur in any order. Here is an example:

    ```
    where lastname eq 'Smith' and firstname eq 'John';
    ```

- When conditions are connected with OR, the conditions must specify the same variable. Here is an example:

```
where firstname eq 'John' and
   (lastname eq 'Smith' or lastname eq 'Jones');
```

Note: SAS transforms the OR conditions that specify the same variable into a single condition that uses the IN operator. For the preceding WHERE expression, SAS converts the two OR conditions into `lastname in ('Smith','Jones')`. Then, SAS uses the composite index for the variables `firstname` and `lastname` in order to select the observations where `firstname` is `John` and `lastname` is `Smith` or `Jones`.

For the following examples, assume there is a composite index for variables `I`, `J`, and `CH`:

- The following WHERE expression conditions are compound optimized because every condition specifies a variable that is in the composite index, and each condition uses one of the supported operators. SAS positions the composite index to the first entry that meets all three conditions and retrieves only observations that meet all three conditions.

```
where I = 1 and J not in (3,4) and 'abc' < CH;
```

- For the following WHERE expression, the first two conditions are compound optimized. After retrieving a subset of observations that satisfy the first two conditions, SAS examines the subset and eliminates any observations that fail to match the third condition.

```
where I in (1,4) and J = 5 and K like '%c';
```

- This WHERE expression can be compound optimized for variables `I` and `J`. After retrieving observations that satisfy the second and third conditions, SAS examines the subset and eliminates those observations that do not satisfy the first condition.

```
where X < 5 and I = 1 and J = 2;
```

- The following WHERE expression can be compound optimized on `I` and `J`:

```
where X < Z and I = 1 and J = 2;
```

- The following WHERE expression cannot be compound optimized because neither `J` nor `CH` is the left-most variable in the composite index:

```
where J = 1 and CH = 'abc';
```

- The following WHERE expression cannot be optimized because the comparison condition on the variable `I` is variable-to-variable, which is not supported for index processing:

```
where I < K and J in (3,4) and CH = 'abc';
```

Compound optimization can occur for a NOMISS composite index if at least one condition does not qualify missing values. That is, compound optimization cannot occur on a NOMISS index, which is an index that does not maintain missing values, if every condition could result in a missing value. The following examples illustrate compound optimization with a NOMISS composite index for variables `I`, `J`, and `K`.

- The following WHERE expression can be compound optimized, because the condition `K = 1` cannot result in a missing value:

```
where I in (.,5) and J < 4 and K = 1;
```

- This WHERE expression cannot be compound optimized, because each condition could result in a missing value:

```
where I in (.,5) and J < 4 and K <= 1;
```

- The following WHERE expression cannot be compound optimized, because each condition could result in a missing value. The condition `J < 4` qualifies observations as `J = .`, and those observations are not represented in the NOMISS composite index:

```
where I = . and J < 4 and .A < K < .D;
```

Using an Index and WHERE Compared to a Subsetting IF Statement

A subsetting IF statement never uses an index. For better performance, consider replacing a subsetting IF statement with a WHERE statement together with an index. See "Subsetting IF Statement versus the WHERE Statement" in *SAS Programmer's Guide: Essentials*.

CAUTION
IF and WHERE statements are processed differently and might produce different results in DATA steps that use the SET, MERGE, or UPDATE statements. The WHERE statement selects observations before they are brought into the Program Data Vector (PDV). The subsetting IF statement selects observations after they are read into the PDV.

Controlling WHERE Processing Index Usage with Data Set Options

You can control index usage for WHERE processing with the IDXWHERE= and IDXNAME= data set options.

The IDXWHERE= data set option overrides the internal analysis by SAS about whether to use an index to meet the conditions of a WHERE expression. Here are the details:

- IDXWHERE=YES tells SAS to decide which index is the best for optimizing a WHERE expression. SAS disregards the possibility that a sequential search of the data set might be more resource-efficient.
- IDXWHERE=NO tells SAS to ignore all indexes and meet the conditions of a WHERE expression by sequentially searching the data set.
- Using an index to process a BY statement cannot be overridden with IDXWHERE=.

The following example tells SAS to decide which index is the best for optimizing the WHERE expression. SAS disregards the possibility that a sequential search of the data set might be more resource-efficient.

```
data mydata.empnew;
   set mydata.employee (idxwhere=yes);
   where empnum < 2000;
```

For details, see "IDXWHERE= Data Set Option" in *SAS Data Set Options: Reference*.

The IDXNAME= data set option directs SAS to use a specific index in order to meet the conditions of a WHERE expression.

By specifying IDXNAME=*index-name*, you are specifying the name of a simple or composite index for the data set.

The following example uses the IDXNAME= data set option to direct SAS to use a specific index to optimize the WHERE expression. SAS disregards the possibility that a sequential search of the data set might be more resource-efficient. SAS does not attempt to determine whether the specified index is the best one. (Note that the `empnum` index was not created with the NOMISS option.)

```
data mydata.empnew;
   set mydata.employee (idxname=empnum);
   where empnum < 2000;
```

For details, see "IDXNAME= Data Set Option" in *SAS Data Set Options: Reference*.

Note: IDXWHERE= and IDXNAME= are mutually exclusive. Using both options results in an error.

Displaying Index Usage Information in the SAS Log

This OPTIONS statement changes the MSGLEVEL= system option to display information about index usage in the SAS log :

```
options msglevel=i;
```

When you set MSGLEVEL=I, the following occurs:

- If an index is used, a message displays the name of the index.
- If an index is not used but one exists that could optimize at least one condition in the WHERE expression, messages provide suggestions about optimization. The suggested actions could influence SAS to use the index. For example, a message might suggest that you sort the data set into index order or specify more buffers.
- A message displays the IDXWHERE= or IDXNAME= data set option value if the setting can affect index processing.

Using an Index with SAS Views

A SAS view is a virtual data set that retrieves data values from other files. For more information, see Chapter 6, "SAS Views," on page 49.

You cannot create an index for a SAS view. However, if a SAS view references a SAS data set that has an index, then the view can use the index. That is, if the view definition includes a WHERE expression using a key variable, then SAS attempts to use the index. There are other ways to take advantage of a key variable when using a SAS view.

In this example, you create an SQL view named `stat` from data set `crime`, which has the key variable `state`. In addition, the view definition includes a WHERE expression:

```
proc sql;
   create view stat as
   select * from crime
   where murder > 7;
quit;
```

If you issue PROC SQL with an SQL WHERE clause that specifies the key variable `state`, then the SQL view can join the two conditions. The join enables SAS to use the index `state`:

```
proc sql;
   select * from stat where state > 42;
quit;
```

Using an Index for BY Processing

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. For the V9 engine, a data set must be sorted or indexed on the BY variable before you submit the BY statement. If the data is not already sorted, you can either use the SORT procedure or use an index.

In the following example code, the `myfile` data set is not sorted on the `lastname` variable. However, the `lastname` variable is indexed, so the BY statement uses the index to order the values by last names.

```
proc print data=myfile;
   by lastname;
run;
```

Using an index to process a BY statement might not always be more efficient than sorting the data set. This is usually the case when the data set has a large number of observations per page. A small observation length and a large page size can result in a large number of observations per page. For information about these data set characteristics, see the data set page size and maximum observations per page in CONTENTS procedure output (or the CONTENTS statement in the DATASETS procedure).

For details about BY processing, see “Grouping Data” in *SAS Programmer’s Guide: Essentials*.

A BY statement can use an index in the following situations:

- If the BY statement contains one variable, the variable must be the key variable for a simple index or be the first key variable in a composite index.

- If the BY statement contains more than one variable, then one of the following requirements must be met:
  - The first variable in the BY statement is the key variable for a simple index. The other variables in the BY statement are sorted in ascending order.
  - The first variable in the BY statement is the key variable for composite index. The other variables are used in the BY statement in the same order as they appear in the composite index, or they are sorted in ascending order.

If the data is not sorted or indexed properly, then SAS stops processing and an error is written in the log.

For example, if the variable `major` has a simple index, the following BY statement uses the index to order the values:

```
by major;
```

The following BY statement can also use the simple index on `major`, but the `state` values must be sorted:

```
by major state;
```

If a composite index named `zipid` exists, and it consists of the variables `zipcode` and `schoolid`, the following BY statements use the index:

```
by zipcode;
by zipcode schoolid;
```

The following BY statement can also use the complex index `zipid`, but the `name` values must be sorted:

```
by zipcode schoolid name;
```

The composite index `zipid` is not used for these BY statements:

```
by schoolid;
by schoolid zipcode;
```

In addition, a BY statement does not use an index in these situations:

- The BY statement includes the DESCENDING or NOTSORTED option.
- The index was created with the NOMISS option.
- The data set is physically stored in sorted order based on the variables specified in the BY statement.

Using an Index for Both WHERE and BY Processing

If both a WHERE expression and a BY statement are specified, SAS looks for one index that satisfies requirements for both. If one index meets the conditions for the BY statement, and another index meets the conditions for the WHERE statement, then SAS uses the index that satisfies the requirements for the BY statement. SAS cannot use an index to optimize a WHERE expression if the optimization would invalidate the BY order.

For example, the following statements could use an index on the variable `lastname` to optimize the WHERE expression. The order of the observations returned by the index does not conflict with the order required by the BY statement:

```
proc print;
  by lastname;
  where lastname >= 'Smith';
run;
```

However, the following statements cannot use an index on `lastname` to optimize the WHERE expression. The BY statement requires that the observations be returned in `empid` order:

```
proc print;
   by empid;
   where lastname = 'Smith';
run;
```

Specifying an Index with the KEY= Option for SET and MODIFY Statements

In the DATA step, the SET and MODIFY statements support the KEY= option, which enables you to specify an index. You cannot use the KEY= option together with the BY statement or the WHERE statement.

Using the _IORC_ automatic variable with the SYSRC autocall macro provides you with error-handling information. You can use _IORC_ to control the actions to take when a match is successful or not successful.

For more information, see the KEY= option in the following statements:

- "MODIFY Statement" in *SAS DATA Step Statements: Reference*
- "SET Statement" in *SAS DATA Step Statements: Reference*

Operations That Affect Indexes

Copying an Indexed Data Set

When you copy an indexed data set by using the COPY procedure (or the COPY statement of the DATASETS procedure), the procedure can re-create the index file for the new data set. Specify INDEX=YES, which is the default. Be aware that re-creating the index increases the processing time for the PROC COPY step.

If you copy from disk to disk, the index is re-created. If you copy from disk to tape, the index is not re-created on tape. However, after copying from disk to tape, if you then copy back from tape to disk, the index can be re-created. Note that if you move a data set with the MOVE option in PROC COPY, the index file is deleted from IN= library and re-created in OUT= library.

The CPORT procedure also has INDEX=YES|NO to specify whether to export indexes with indexed data sets. By default, PROC CPORT exports indexes with indexed data sets. The CIMPORT procedure, however, does not handle the index file at all, and indexes must be re-created.

Updating an Indexed Data Set

When values in an indexed data set are added, modified, or deleted, SAS automatically updates the index. The following activities affect an index as indicated:

Table 8.2 *Maintenance Tasks and Index Results*

Task	Result
delete a data set	index file is deleted
rename a data set	index file is renamed
rename key variable	simple index is renamed
delete key variable	simple index is deleted
add observation	index entries are added
delete observations	index entries are deleted and space is recovered for reuse
update observations	index entries are deleted and new ones are inserted

Note: Use SAS to perform additions, modifications, and deletions to your data sets. Using operating environment commands to perform these operations makes your files unusable.

Appending observations to an indexed data set requires additional processing. SAS automatically keeps the values in the index consistent with the values in the data set. To reduce the performance impact, SAS suspends index updates until all observations are added, and then updates the index with data from the newly added observations. See "APPEND Statement" in *Base SAS Procedures Guide*.

An index that is created without the UNIQUE option can result in multiple occurrences of the same value, which results in multiple RIDs for one value. For large data sets with many multiple occurrences of the same value, the list of RIDs for a given value might require several pages in the index file. Because the RIDs are stored in physical order, any new observation that is added to the data set with the

given value is stored at the end of the list of RIDs. Navigating through the index to find the end of the RID list can cause many I/O operations. SAS remembers the previous position in the index so that when inserting more occurrences of the same value, the end of the RID list is found quickly.

Sorting an Indexed Data Set

You can sort an indexed data set only if you direct the output of the SORT procedure to a new data set so that the original data set remains unchanged. However, the new data set is not automatically indexed.

Note: If you sort an indexed data set with the FORCE option, the index file is deleted.

Indexes and Integrity Constraints

Integrity constraints can also use indexes. When an integrity constraint that uses an index is created, if a suitable index already exists, it is used. Otherwise, a new index is created. When an index is created, it is marked as being “owned” by the creator, which can be either the user or an integrity constraint.

If either the user or an integrity constraint requests creation of an index that already exists and is owned by the other, the requestor is also marked as an "owner" of the index. If an index is owned by both, then a request by either to delete the index results in removing only the requestor as owner. The index is deleted only after both the integrity constraint and the user have requested the index's deletion. A note in the log indicates when an index cannot be deleted.

Indexes and CEDA Processing

When the encoding or data representation of a data set does not match that of the SAS session, Cross-Environment Data Access (CEDA) is used to access the data. However, indexes and integrity constraints cannot be accessed. Output from PROC CONTENTS (or the CONTENTS statement in PROC DATASETS) for such a data set does not show the presence of an index or integrity constraint. Therefore, WHERE optimization that uses an index is not supported.

Similarly, copying such a data set with PROC COPY (or the COPY statement in PROC DATASETS) results in an output data set that has no indexes or integrity constraints. The only exception is PROC MIGRATE, which supports the re-creation of indexes and integrity constraints in the SAS session’s encoding and data representation.

For information about CEDA, see “Cross-Environment Data Access” in *SAS Programmer’s Guide: Essentials*.

How the V9 Engine Creates and Uses an Index File

What Is an Index File?

The index file is a SAS file that has the same name as its associated data set, and that has a member type of INDEX. For the V9 engine, there is only one index file per data set. That is, all indexes for a data set are stored in a single file.

The index file might be a separate file, or be part of the data set, depending on the operating environment. The index file is stored in the same SAS library as its data set.

The index file consists of entries that are organized hierarchically and connected by pointers, all of which are maintained by SAS. The lowest level in the index file hierarchy consists of entries that represent each distinct value for an indexed variable, in ascending value order. Each entry contains this information:

- a distinct value
- one or more unique RIDs that indicate each observation containing the value. (Think of the RID as an internal observation number.)

How SAS Creates the Index File

When you request to create an index, SAS reads the data set, one observation at a time. SAS extracts values and RIDs for each key variable, and places them in the index file. That is, in an index file, each value is followed by one or more RIDs, which identify the observations in the data set that contains the value. (Multiple RIDs result from multiple occurrences of the same value.) For example, the following table represents index file entries for the variable LastName:

Table 8.3 *Index File Entries*

Value	Record Identifier
`Avery`	10
`Brown`	6, 22, 43
`Craig`	5, 50
`Dunn`	1

SAS ensures that the values that are placed in the index are successively the same or increasing. SAS determines whether the data is already sorted by the key variables in ascending order. It determines the data's sort status by checking the sort indicator in the data set, which is an attribute of the file that indicates how the data is sorted. The sort indicator is stored with the SAS data set descriptor information, and is set from a previous SORT procedure or SORTEDBY= data set option.

If the values in the sort indicator are in ascending order, SAS does not sort the values for the index file and avoids the resource cost. Note that SAS always validates that the data is sorted as indicated. If not, the index is not created. For example, if the sort indicator was set from a SORTEDBY= data set option, and the data is not sorted as indicated, an error occurs. A message is written to the SAS log stating that the index was not created because values are not sorted in ascending order.

If the values in the sort indicator are not in ascending order, SAS sorts the data that is included in the index file in ascending order. To sort the data, SAS follows this process:

1. SAS first attempts to sort the data using the thread-enabled sort. By dividing the sorting into separately executable processes, the time to sort the data can be reduced. To use the thread-enabled sort:
   - The index must be sufficiently large. SAS determines the index size.
   - The SAS system option CPUCOUNT= must be set to more than one processor.
   - The THREADS system option must be enabled.
   - Adequate memory must be available for the thread-enabled sort. (If not enough memory is available, SAS reduces the number of threads to one and begins the sort process again, which increases the time to create the index.)
2. If the thread-enabled sort cannot be done, SAS uses the unthreaded sort.

Note: To display messages regarding what type of sort is used, memory and resource information, and the status of the index being created, set the SAS system option MSGLEVEL=I.

```
options msglevel=i;
```

SAS automatically keeps the index file balanced as updates are made. A balanced index file ensures a uniform cost to access any index entry. All space that is occupied by deleted values is recovered and reused.

How SAS Uses the Index File

When an index is used to process a request, such as a WHERE expression, SAS performs a binary search on the index file. SAS positions the index to the first entry that contains a qualified value. SAS then uses the value's RID to read the observation that contains the value. If a value has more than one RID (such as the value `Brown` in the previous example), SAS reads the observation that is pointed to by the next RID in the list. The result is that SAS can quickly locate the observations that are associated with a value or range of values.

For example, an index can optimize the following WHERE expression:

```
where age > 20 and age < 35;
```

SAS positions the index to the index entry for the first value that is greater than 20, and uses the value's RID or RIDs to read the observation or observations. Then, SAS moves sequentially through the index entries reading observations until it reaches the index entry for the value that is equal to or greater than 35.

Examples: Indexes

Example: Create an Index by Using the DATA Step

Example Code

This series of examples creates and indexes the `mybaseball` data set in the Work library. The data is copied from `sashelp.baseball`. The data set and its indexes are created in the same DATA step.

In the first example, the INDEX= data set option creates a simple index on the `team` variable:

```
data mybaseball(index=(team));
   set sashelp.baseball;
run;
```

The following example creates a simple index on the `team` variable and a simple index on the `name` variable:

```
data mybaseball(index=(team name));
   set sashelp.baseball;
run;
```

The following example creates a composite index named `hitserror`. The first key variable is `nhits` and the second key variable is `nerror`:

```
data mybaseball(index=(hitserror=(nhits nerror)));
   set sashelp.baseball;
run;
```

The following example creates a simple index on the `name` variable that has unique values and no missing values:

```
data mybaseball(index=(name/unique/nomiss));
   set sashelp.baseball;
run;
```

The following example creates a composite index named `hitserror` with no missing values. The first key variable is `nhits` and the second key variable is `nerror`:

```
data mybaseball(index=(hitserror=(nhits nerror)/nomiss));
   set sashelp.baseball;
run;
```

The following example creates a simple index on the `name` variable that has unique values and no missing values, and a composite index named `hitserror` with no missing values. In the composite index `hitserror`, the first key variable is `nhits`, and the second key variable is `nerror`:

```
data mybaseball(index=(name/unique/nomiss hitserror=(nhits nerror)/
nomiss));
   set sashelp.baseball;
run;
```

Key Ideas

- To create indexes in a DATA step when you create the data set, use the INDEX= data set option.
- With the INDEX= data set option, you can specify the NOMISS or UNIQUE option after a slash. When you specify more than one data set option in the DATA statement, place a slash before each option.

See Also

- "INDEX= Data Set Option" in *SAS Data Set Options: Reference*
- "Types of Indexes" on page 71

Example: Create or Delete an Index by Using the DATASETS Procedure

Example Code

In this example, the DATASETS procedure deletes and creates indexes from an existing data set, `mybaseball`.

```
data mybaseball(index=(team name));                    /* 1 */
   set sashelp.baseball;
```

```
run;

proc datasets library=work;
   modify mybaseball;                                    /* 2 */
      index delete team name;                            /* 3 */
      index create name / unique nomiss;                 /* 4 */
      index create team nruns / nomiss;                  /* 5 */
      index create nrbi hitserror=(nhits nerror)/nomiss; /* 6 */
quit;
```

1 Create `mybaseball` in the Work library by copying `sashelp.baseball`.

2 The data set to be modified is `mybaseball`.

3 The INDEX DELETE statement deletes two indexes, `team` and `name`.

4 The INDEX CREATE statement creates a simple index on the `name` variable that has unique values and no missing values.

5 The INDEX CREATE statement creates a simple index on the `team` variable and a simple index on the `nruns` variable. Both of the indexes have no missing values.

6 The INDEX CREATE statement creates a simple index on the `nrbi` variable, and composite index named `hitserror`. In the composite index, the first key variable is `nhits`, and the second key variable is `nerror`. Both the `nrbi` index and the `hitserror` index have no missing values.

Key Ideas

- The INDEX CREATE and INDEX DELETE statements are applied to the data set that is specified in the MODIFY statement of PROC DATASETS.
- The INDEX CREATE statement supports the NOMISS and UNIQUE options after a slash. When you specify more than one option in the INDEX CREATE statement, use one slash before the first option. Do not place a slash before the second option. The specified options apply to all indexes that are created in the statement.
- If you delete and create indexes in the same MODIFY step, place the INDEX DELETE statement before the INDEX CREATE statement. This practice enables the space that was occupied by deleted indexes to be reused.

See Also

- “INDEX DELETE Statement” in *Base SAS Procedures Guide*
- “INDEX CREATE Statement” in *Base SAS Procedures Guide*
- “MODIFY Statement” in *Base SAS Procedures Guide*

Example: Create or Delete an Index by Using the SQL Procedure

Example Code

This example creates and indexes the `newbaseball` data set in the Work library. The data is copied from `sashelp.baseball`. The data set and its indexes can be created in the same SQL procedure step or in a separate step.

```
proc sql;
   create table newbaseball as
      select * from sashelp.baseball;
   create unique index name on newbaseball (name);        /* 1 */
proc sql;
   create index hitserror on newbaseball (nhits, nerror); /* 2 */
proc sql;
   drop index name from newbaseball;                      /* 3 */
quit;
```

1. The CREATE INDEX statement creates a simple index on the `name` variable that has unique values.
2. The CREATE INDEX statement creates a complex index named `hitserror`. In the composite index, the first key variable is `nhits`, and the second key variable is `nerror`.
3. The DROP INDEX statement deletes the `name` index.

Key Ideas

- For multiple variable names, use a comma to separate the names (which is an SQL convention) instead of blanks (which is a SAS convention).
- If you are creating a simple index, then the index name must be the same as variable name. If you are creating a composite index, then the index name cannot be the same as any variable in the table.
- The SQL procedure supports the UNIQUE option but not the NOMISS option.

See Also

- "CREATE INDEX Statement" in *SAS SQL Procedure User's Guide*
- "DROP Statement" in *SAS SQL Procedure User's Guide*

- "Creating an Index" in *SAS SQL Procedure User's Guide*

Example: View Index Information for a Data Set

Example Code

This example demonstrates how to find index information in PROC CONTENTS output.

```
data mybaseball(index=(name/unique/nomiss hitserror=(nhits nerror)/
nomiss));
   set sashelp.baseball;
run;
proc contents data=mybaseball;
run;
```

PROC CONTENTS reports the following information about the data set's indexes:

- number and names of indexes for a data set
- the names of key variables
- the options in effect for each key variable

Output 8.1 *Index Information in PROC CONTENTS Output*

Alphabetic List of Indexes and Attributes					
#	Index	Unique Option	NoMiss Option	# of Unique Values	Variables
1	Name	YES	YES	322	
2	hitserror		YES	298	nHits nError

The Engine/Host portion of the PROC CONTENTS output shows information that can be useful when deciding whether a data set is large enough to benefit from an index:

- data set page size and the number of data set pages
- index file page size and the number of index file pages

Output 8.2 *Engine/Host Portion of PROC CONTENTS Output*

Engine/Host Dependent Information	
Data Set Page Size	65536
Number of Data Set Pages	3
First Data Page	1
Max Obs per Page	303
Obs in First Data Page	286
Index File Page Size	4096
Number of Index File Pages	8
Number of Data Set Repairs	0
ExtendObsCounter	YES
Filename	C:\SAS Temporary Files_1234_\mybaseball.sas7bdat
Release Created	9.0401M6
Host Created	X64_10PRO
Owner Name	userid
File Size	256KB
File Size (bytes)	262144

Key Ideas

- The CONTENTS procedure (or the CONTENTS statement in PROC DATASETS) reports information about a data set's indexes.
- The Engine/Host portion of CONTENTS output varies by operating environment, and might not show all the information that is demonstrated in this example.
- Include the CENTILES option to display centiles information for indexed variables:

```
proc contents centiles data=mybaseball;
run;
```

See Also

- "Concepts: CONTENTS Procedure" in *Base SAS Procedures Guide*
- "CENTILES" in *Base SAS Procedures Guide*

Example: Recover a Damaged Index

Example Code

This example repairs a damaged data set, `mylib.mydata`, and re-creates its indexes.

```
proc datasets library=mylib;
   repair mydata;
run;
```

Key Ideas

- An index can become damaged for many of the same reasons that a data set or catalog can become damaged.
- Use the REPAIR statement in PROC DATASETS to repair a data set. The REPAIR statement attempts to re-create all indexes for the data set. The restored data set might not include the last several updates that occurred before the system failed.
- If indexes are destroyed by using the FORCE option in PROC SORT, you cannot use the REPAIR statement to re-create the indexes.

See Also

- "REPAIR Statement" in *Base SAS Procedures Guide*
- "SORT Procedure" in *Base SAS Procedures Guide*

9

Integrity Constraints

Definition of an Integrity Constraint

An integrity constraint is a data validation rule that restricts the data values that can be stored for a variable in a SAS data set. Integrity constraints help preserve the validity and consistency of the data.

Here are some essential concepts:

- SAS enforces the integrity constraints when the values associated with a variable are added, updated, or deleted.
- If you add an integrity constraint to an existing data set, SAS verifies that the existing data values conform to the constraint that is being added.
- SAS integrity constraints are supported for V9 engine data sets only. Integrity constraints are not supported for views.
- If you specify a name for a constraint, the name must be a valid SAS name. (See "Rules for Most SAS Names" in *SAS Programmer's Guide: Essentials*.) If you do not supply a name, a default name is generated.
- Primary key, foreign key, and unique integrity constraints are stored in an index. Therefore, the member type is INDEX and the file extension in most operating environments is .sas7bndx. See "SAS Files and Member Types" on page 4.
- If generation data sets are used, you must create the integrity constraints in each data set generation that includes protected variables.
- You can customize the behavior when observations fail a constraint and are rejected by SAS. To customize the log messages, use the MESSAGE= and MSGTYPE= options. To collect rejected observations in a separate data set, use an audit trail. See "Example: Manage the Rejected Observations from an Integrity Constraint" on page 114.
- If multiple variables are included in the specification for a primary key, foreign key, or a unique integrity constraint, a composite index is created. The integrity constraint enforces the combination of variable values. The relationship between SAS indexes and integrity constraints is described in "How Integrity Constraints Use Indexes" on page 105.
- Integrity constraints are either general or referential. Under some restrictions, one variable can be used in both types of constraints, which is called an overlapping constraint.

General Integrity Constraints

General integrity constraints enable you to restrict the values of variables within a single file. There are four types of general constraints:

check
: limits the data values of variables to a specific set, range, or list of values. Check constraints can also enforce a contingency from one variable to another variable in the same observation. Check constraints that are created in SAS 9.2 and later are not compatible with earlier releases of SAS.

not null
: requires that a variable contain a data value. Null (missing) values are not allowed.

unique
: requires that the specified variable or variables contain unique data values. A null data value is allowed but is limited to a single instance.

primary key
: requires that the specified variable or variables contain unique data values and that null data values are not allowed. Only one primary key can exist in a data set.

  Usually, a primary key constraint is created so that a foreign key constraint in a different data set can refer to it. A primary key is a general integrity constraint if it does not have any foreign key constraints referencing it.

  If you do not want to create a foreign key referential constraint, then you could create a unique constraint and a not-null constraint instead of a primary key constraint.

For a comparison to UNIQUE or NOMISS indexes, see “How Integrity Constraints Use Indexes” on page 105.

See also “Examples: Integrity Constraints” on page 109.

Referential (Foreign Key) Integrity Constraints

Definition of a Referential Integrity Constraint

A referential integrity constraint is created when a primary key integrity constraint in one data set is referenced by a foreign key integrity constraint in another data set. The constraint affects the foreign key and primary key data sets in different ways.

In the foreign key data set, values are checked when you create the foreign key constraint or if you change the values in the foreign key variable or variables. The operation is successful if one of the following conditions is true:

- The data value in the foreign key variable or variables has a matching value or values in the primary key data set.
- The data value in the foreign key variable or variables is null.

In the primary key data set, values are checked when you update or delete values in the primary key variable or variables. The modifications are controlled by a referential action that is defined in the foreign key constraint.

Separate referential actions can be defined for the Update and Delete operations. There are three types of referential actions:

restrict
: prevents the data values of the primary key variables from being updated or deleted if a matching value exists in one of the foreign key data set's corresponding foreign key variables. The restrict action is the default.

set null
: allows the data values of the primary key variables to be updated or deleted, but matching data values in the foreign key data sets are changed to null (missing) values.

cascade
: allows the data values in the primary key variables to be updated, and also updates matching data values in the foreign key data sets to the same value. The cascade type of action is supported for Update operations only.

Here are the requirements for establishing a referential relationship:

- You must create a primary key integrity constraint before you reference it in a foreign key integrity constraint.
- The primary key and foreign key must reference the same number of variables, and the variables must be in the same order.
- The variables must be of the same type (character or numeric) and length.
- There is no limit on the number of foreign keys that can reference a primary key. However, additional foreign keys can adversely impact the performance of Update and Delete operations.
- When a referential constraint exists, a primary key integrity constraint is not deleted until all foreign keys that reference it are deleted. There are no restrictions on deleting foreign keys.
- Referential integrity constraints cannot be assigned to data sets in concatenated libraries.

See also “Examples: Integrity Constraints” on page 109.

Inter-library Referential Integrity Constraints

The foreign key data set can exist in the same SAS library as the referenced primary key data set (intra-library) or in a different SAS library (inter-library). If the library that contains the foreign key data set is temporary, then the library that contains the primary key data set must be temporary as well.

Several SAS procedures can preserve the integrity constraints when you copy a data set or library, although you might have to reactivate inter-library foreign key constraints. See “Copying Data Sets That Have Integrity Constraints” on page 106.

When users share disk space over a network, and the foreign key and primary key data sets are in different SAS libraries, a standard should be established for the physical location. All network machines must use the same physical name in order to access the files, because the path names are stored in the descriptor information of the data sets. If the physical names do not match, SAS cannot open the referenced foreign key or primary key data set.

Here is an example of a problem regarding files that were created without a standard. Suppose a primary key and a foreign key data set are created on machine D4064 in different directories, **C:\Public\pkey_directory** and **C:\Public\fkey_directory**.

To access the primary key data set from a different machine such as F2760, the following LIBNAME statement would be executed:

```
libname pkds '\\D4064\Public\pkey_directory';
```

When the primary key data set is opened for an Update operation, SAS tries to open the foreign key data set. SAS uses the physical name of the foreign key data set, which is stored in the primary key data set. In this example, the physical name is **C:\Public\fkey_directory**. However, that directory does not exist on machine F2760. Therefore, opening the foreign key data set fails.

Overlapping Primary Key and Foreign Key Constraints

Variables in a SAS data set can be part of both a primary key (general integrity constraint) and a foreign key (referential integrity constraint). Here are the requirements:

- The foreign key's update and delete referential actions must both be RESTRICT.
- When the same variables are used in a primary key definition and a foreign key definition, the variables must be defined in a different order. If one of the constraints is defined for a single variable, then the other constraint must be defined for more than one variable.

See “Example: Define Overlapping Primary Key and Foreign Key Constraints” on page 116.

How Integrity Constraints Use Indexes

The primary key, foreign key, and unique integrity constraints store data values in an index file. If an index file already exists, it is used. Otherwise, one is created. Consider the following points when you create or delete an integrity constraint:

- When a user-defined index exists, the index's attributes must be compatible with the integrity constraint in order for the integrity constraint to be created. For example, when you add a primary key integrity constraint, the existing index must have the UNIQUE attribute. When you add a foreign key integrity constraint, the index must not have the UNIQUE attribute.
- The unique integrity constraint has the same effect as the UNIQUE index attribute. Therefore, when one is used, the other is not necessary.
- The NOMISS index attribute and the not-null integrity constraint have different effects. The integrity constraint prevents missing values from being written to the SAS data set and cannot be added to an existing data set that contains missing

values. The index attribute allows missing data values in the data set but excludes them from the index.

- When any index is created, it is marked as being "owned" by the user, the integrity constraint, or both. A user cannot delete an index that is also owned by an integrity constraint and vice versa. If an index is owned by both, the index is deleted only after both the integrity constraint and the user have requested the index's deletion. A note in the log indicates when an index cannot be deleted.

Operations That Affect Integrity Constraints

Copying Data Sets That Have Integrity Constraints

Here are some essential concepts about copying data sets:

- The following table lists SAS procedures that can preserve integrity constraints when their operation results in a copy of the original data set. For more details, see each procedure's documentation.
- Many procedures provide the CONSTRAINT=YES option to preserve integrity constraints. Inter-library referential integrity constraints are preserved in an inactive state, and you must use the IC REACTIVATE statement in a MODIFY RUN group of PROC DATASETS to reactivate them.
- When you copy a data set by using the DATA step with SET statement, integrity constraints are not copied.
- Do not use operating environment tools to copy a data set that has integrity constraints or to manage integrity constraints. This practice can damage the data set.

Table 9.1 *Circumstances That Preserve Integrity Constraints in a Copied Data Set*

Procedure	Condition	Constraints That Are Preserved
APPEND	BASE= data set does not exist	Only general constraints of the DATA= data set are copied to the new BASE= data set.
APPEND	BASE= and DATA= data sets both exist	All integrity constraints of only the BASE= data set are preserved.
COPY or	CONSTRAINT=yes	General constraints are preserved.

Procedure	Condition	Constraints That Are Preserved
CPORT and CIMPORT		Intra-library referential constraints are preserved. Inter-library referential constraints are preserved in an inactive state. (If you use the SELECT or EXCLUDE statement to copy the data sets, then referential integrity constraints are not copied.)
MIGRATE		General constraints are preserved. Intra-library referential constraints are preserved. Inter-library referential constraints are preserved in an inactive state. (PROC MIGRATE migrates one library at a time.)
SORT	OUT= data set is not specified	All integrity constraints are preserved.
UPLOAD and DOWNLOAD in SAS/CONNECT	CONSTRAINT=yes and OUT= data set is not specified	General constraints are preserved. Intra-library referential constraints are preserved. Inter-library referential constraints are preserved in an inactive state.
Copying or moving in the SAS Explorer window of the SAS windowing environment		Only general constraints are preserved.

CAUTION
Do not use operating environment tools to copy a data set that has integrity constraints or to manage integrity constraints. This practice can damage the data set. Instead, use SAS procedures such as PROC DATASETS or PROC SQL.

For additional details about integrity constraints for each procedure, see

- *Base SAS Procedures Guide*
- *SAS/CONNECT User's Guide*

Integrity Constraints and CEDA Processing

When the encoding or data representation of a data set does not match that of the SAS session, Cross-Environment Data Access (CEDA) is used to access the data. However, indexes and integrity constraints cannot be accessed. Output from PROC CONTENTS (or the CONTENTS statement in PROC DATASETS) for such a data set does not show the presence of an index or integrity constraint. Therefore, WHERE optimization that uses an index is not supported.

Similarly, copying such a data set with PROC COPY (or the COPY statement in PROC DATASETS) results in an output data set that has no indexes or integrity constraints. The only exception is PROC MIGRATE, which supports the re-creation of indexes and integrity constraints in the SAS session's encoding and data representation.

For information about CEDA, see "Cross-Environment Data Access" in *SAS Programmer's Guide: Essentials*.

Integrity Constraints and Locking

Integrity constraints support both member-level and record-level locking. You can override the default locking level with the CNTLLEV= data set option. See the "CNTLLEV= Data Set Option" in *SAS Data Set Options: Reference*.

Integrity Constraints and Encryption

SAS uses two types of encryption algorithms:

- SAS Proprietary encryption is implemented with the ENCRYPT=YES data set option.
- AES (Advanced Encryption Standard) encryption is implemented with the ENCRYPT=AES data set option.

SAS Proprietary encryption has no restrictions when using integrity constraints.

AES encryption requires that all primary key and foreign key data sets must use the same encryption key that opens all referencing foreign key and primary key data sets. You must specify the ENCRYPTKEY= data set option when using ENCRYPT=AES. For more information, see "ENCRYPT= Data Set Option" in *SAS Data Set Options: Reference* and "ENCRYPTKEY= Data Set Option" in *SAS Data Set Options: Reference*.

If an encryption key was not recorded for the metadata-bound library, then the encryption key must be the same for the primary key data set and the referencing encrypted foreign key data set. For more information about metadata-bound libraries, see "Metadata-Bound Library" in *Base SAS Procedures Guide*.

Examples: Integrity Constraints

Example: Create General Integrity Constraints by Using the DATASETS Procedure

Example Code

This example creates the `player_stats` data set and adds three integrity constraints.

```
data player_stats;
   set sashelp.baseball;
run;
proc datasets nolist;
   modify player_stats;
      ic create primary_name = primary key(name); /* 1 */
      ic create check(where=(nhits>1));           /* 2 */
      ic create not null(nerror);                 /* 3 */
quit;
```

1. The first IC CREATE statement creates a primary key constraint on the `name` variable. The constraint is named `primary_name`. This constraint is referenced in "Example: Create Referential Integrity Constraints by Using the DATASETS Procedure" on page 110.
2. This statement creates a check constraint on the variable `nhits` to ensure that the number is greater than 1. The constraint is not named, so SAS will generate a name.
3. This statement creates a not null constraint on the variable `nerror`. The constraint is not named, so SAS will generate a name.

Here is a portion of the log output. Notice the constraint names that are generated for the two constraints that were not named in the submitted code.

```
5       modify player_stats;
6          ic create primary_name = primary key(name);
NOTE: Integrity constraint primary_name defined.
7          ic create check(where=(nhits>1));
NOTE: Integrity constraint _CK0001_ defined.
8          ic create not null(nerror);
NOTE: Integrity constraint _NM0001_ defined.
```

Key Ideas

- When you create integrity constraints, you must specify a separate statement for each constraint.
- When you create an integrity constraint, the creation fails if the data violates the constraint.
- If you attempt to update a data set that has general integrity constraints, the update fails if the data violates the constraint.

See Also

- "General Integrity Constraints" on page 102
- "IC CREATE Statement" in *Base SAS Procedures Guide*

Example: Create Referential Integrity Constraints by Using the DATASETS Procedure

Example Code

This example creates the `player_notes` data set to demonstrate a referential (foreign key) integrity constraint. To run this example, you must first run "Example: Create General Integrity Constraints by Using the DATASETS Procedure" on page 109.

```
data player_notes;
   input name $1-18 info $19-48;
   datalines;
Mattingly, Don    uniform number 23
White, Frank      uniform number 20
;
run;

proc datasets nolist;
   modify player_notes;
      ic create foreign_name =                      /* 1 */
         foreign key(name) references player_stats /* 2 */
         on update cascade                          /* 3 */
         on delete set null;                        /* 4 */
quit;
```

1. The constraint is named `foreign_name`.
2. The constraint references the `name` variable in the `player_stats` data set, which is created in "Example: Create General Integrity Constraints by Using the DATASETS Procedure" on page 109.
3. The ON UPDATE action is CASCADE. If the `player_stats` data set is updated, then that change cascades to the `player_notes` data set. Only the `name` variable is affected. If the `name` value does not exist in `player_notes`, then the change is not cascaded. If a change occurs in a variable other than `name`, then the change is not cascaded.
4. The ON DELETE action is SET NULL. If an observation in the `player_stats` data set is deleted, then the `name` value for that observation is set to null (missing) in `player_notes`.

The following code demonstrates the referential integrity constraint on the foreign key data set. The PROC SQL code adds two observations to `player_notes`.

```
proc sql;
   insert into player_notes
      values ('Raines, Tim', 'uniform number 30')
      values ('Raines, Tim', 'National Baseball Hall of Fame');
   select name, info from player_notes;
quit;
```

The observations are successfully added to `player_notes` because the `name` value `Raines, Tim` is present in the primary key data set. Notice that there is no unique constraint on the `name` variable in the `player_notes` data set, so two values of `Raines, Tim` are allowed.

Output 9.1 *Rows Added Successfully in Foreign Key Data Set*

name	info
Mattingly, Don	uniform number 23
White, Frank	uniform number 20
Raines, Tim	uniform number 30
Raines, Tim	National Baseball Hall of Fame

The following code demonstrates a referential integrity constraint on the primary key data set and an ON DELETE action on the foreign key data set.

```
proc sql;
   delete from player_stats
      where name contains 'Mattingly';  /* 1 */
   select name, info from player_notes; /* 2 */
quit;
```

1. The DELETE statement is successful, because the `name` value `Mattingly, Don` is present in the foreign key data set and the action for ON DELETE is SET NULL.
   - In `player_stats`, the observation is deleted.
   - In `player_notes`, the observation is not deleted. Instead, the `name` value `Mattingly, Don` is removed and is set to missing.
2. The SELECT statement prints the `player_notes` data set, showing that `Mattingly, Don` has been removed, and the `name` value is set to missing.

Output 9.2 *Name Value Is Set to Missing in Foreign Key Data Set*

name	info
	uniform number 23
White, Frank	uniform number 20
Raines, Tim	uniform number 30
Raines, Tim	National Baseball Hall of Fame

Key Ideas

- You must create a primary key integrity constraint before you reference it in a foreign key integrity constraint.
- Referential integrity constraints affect the primary key and foreign key data sets in different ways.

See Also

- "Referential (Foreign Key) Integrity Constraints" on page 103
- "IC CREATE Statement" in *Base SAS Procedures Guide*

Example: Create General Integrity Constraints by Using the SQL Procedure

Example Code

This PROC SQL example creates the `player_sql` data set and adds three integrity constraints.

```
proc sql;
   create table player_sql as
      select * from sashelp.baseball;
   alter table player_sql
      add constraint prim_name primary key(name), /* 1 */
                     check (nhits>1),             /* 2 */
                     not null (nerror)            /* 3 */;
quit;
```

1 The `prim_name` constraint is a primary key constraint on the `name` variable. This constraint is referenced in "Example: Create Referential Integrity Constraints by Using the SQL Procedure" on page 113.

2 This check constraint on the `nhits` variable ensures that the number is greater than 1. The constraint is not named, so SAS will generate a name.

3 This not null constraint on the `nerror` variable ensures that the variable contains no missing values. The constraint is not named, so SAS will generate a name.

Key Ideas

- PROC SQL can create the same types of general integrity constraints as PROC DATASETS can.
- The PROC SQL syntax differs significantly from PROC DATASETS. For example, do not include the WHERE keyword in a check constraint.

See Also

- "General Integrity Constraints" on page 102
- "Creating and Using Integrity Constraints in a Table" in *SAS SQL Procedure User's Guide*

Example: Create Referential Integrity Constraints by Using the SQL Procedure

Example Code

This PROC SQL example creates the `player_notes_sql` data set to demonstrate a referential (foreign key) integrity constraint. To run this example, you must first run "Example: Create General Integrity Constraints by Using the SQL Procedure" on page 112.

```
proc sql;
   create table player_notes_sql
          (name char(18),
           info char(30),
           constraint for_key_name foreign key(name) /* 1 */
             references player_sql                   /* 2 */
             on update cascade                       /* 3 */
             on delete set null );                   /* 4 */
   insert into player_notes_sql
      values ('Mattingly, Don', 'uniform number 23')
      values ('White, Frank', 'uniform number 20');
   select * from player_sql
quit;
```

1. The constraint is named `for_key_name`.
2. The constraint references the `name` variable in the `player_sql` data set, which is created in "Example: Create General Integrity Constraints by Using the DATASETS Procedure" on page 109.
3. The ON UPDATE action is CASCADE. If the `player_sql` data set is updated, then that change cascades to the `player_notes_sql` data set. Only the `name` variable is affected. If the `name` value does not exist in `player_notes_sql`, then the change is not cascaded. If a change occurs in a variable other than `name`, then the change is not cascaded.
4. The ON DELETE action is SET NULL. If an observation in the `player_sql` data set is deleted, then the `name` value for that observation is set to null (missing) in `player_notes_sql`.

The data sets and the foreign key integrity constraint in this example are named differently than in "Example: Create Referential Integrity Constraints by Using the DATASETS Procedure" on page 110. However, the data in the data sets is the same, and the referential actions behave the same.

Key Ideas

- PROC SQL can create the same types of referential integrity constraints as PROC DATASETS can.
- The PROC SQL syntax differs significantly from PROC DATASETS. For example, you can create constraints within the CREATE TABLE statement or later in an ALTER TABLE statement.

See Also

- "Referential (Foreign Key) Integrity Constraints" on page 103
- "Creating and Using Integrity Constraints in a Table" in *SAS SQL Procedure User's Guide*

Example: Manage the Rejected Observations from an Integrity Constraint

Example Code

This example customizes the error message that is written to the log for observations that fail an integrity constraint. An audit trail collects the rejected observations and information about modifications to the data set.

```
data player_test;
   set sashelp.baseball;
   keep name team;
run;
proc datasets nolist;
   modify player_test;
      ic create unique(name)
         message='The value in the Name variable is not unique.' /* 1 */
         msgtype=user;                                            /* 2 */
   audit player_test;                                             /* 3 */
   initiate;
   log before_image=yes data_image=yes
      error_image=yes;                                            /* 4 */
quit;
proc sql;
   insert into player_test
      values ('Raines, Tim', 'Montreal');
quit;
```

1. The MESSAGE= option in the IC CREATE statement adds a custom message to the SAS error message for this constraint.
2. The MSGTYPE=USER option eliminates the usual SAS error message that states the operation failed due to an integrity constraint.
3. The AUDIT statement begins a block of statements to initiate an audit trail. These statements are called an audit run group.
4. These LOG options specify which observation (record) images to log.

The custom message is written to the SAS log:

```
ERROR: The value in the Name variable is not unique.
```

If MSGTYPE=USER was not specified, the custom message would be prepended to the usual error message:

```
ERROR: The value in the Name variable is not unique. Add/Update failed
for data set WORK.PLAYER_TEST because data value(s) do not comply with
integrity constraint _UN0001_.
```

Print the audit trail to see more information about the error.

```
proc print data=player_test (type=audit) noobs;
   var _ATDATETIME_ _ATOBSNO_ _ATMESSAGE_;
run;
```

Output 9.3 *Audit Trail Showing Modifications and Error Message*

ATDATETIME	_ATOBSNO_	_ATMESSAGE_
25APR2019:12:07:52	323	ERROR: The value in the Name variable is not unique.

Key Ideas

- The MESSAGE= option enables you to prepend a user-defined message to the SAS error message associated with an integrity constraint.
- The MSGTYPE= option suppresses the SAS portion of the message.
- The MESSAGE= and MSGTYPE=USER options are available in PROC DATASETS and PROC SQL.
- Use an audit trail to collect the rejected observations together with information about modifications to the data set. Audit trails provide several automatic variables that store modification data. You can also define user variables to store your own modification data. Use PROC DATASETS to create an audit trail.

See Also

- Chapter 10, “Audit Trails,” on page 127
- “<MESSAGE='message-string' <MSGTYPE=USER>>” in *Base SAS Procedures Guide*
- “AUDIT Statement” in *Base SAS Procedures Guide*
- “Creating and Using Integrity Constraints in a Table” in *SAS SQL Procedure User’s Guide*
- “ALTER TABLE Statement” in *SAS SQL Procedure User’s Guide*

Example: Define Overlapping Primary Key and Foreign Key Constraints

Example Code

The following code creates two data sets, `Singers1` and `Singers2`, to demonstrate primary key and foreign key constraints that overlap.

```
data Singers1;
   length LastName $13;
   input FirstName $ LastName $ Age;
   datalines;
Kris Kristofferson 66
Tom Jones 62
Willie Nelson 69
Barbra Streisand 60
Paul McCartney 60
```

```
Randy Travis 43
;
data Singers2;
   length LastName $13;
   input FirstName $ LastName $ Style $;
   datalines;
Kris Kristofferson Country
Tom Jones Rock
Willie Nelson Country
Barbra Streisand Contemporary
Paul McCartney Rock
Randy Travis Country
;
proc datasets library=work nolist;
   modify Singers1;
      ic create primary key (FirstName LastName); /* 1 */
   run;
   modify Singers2;
      ic create foreign key (FirstName LastName) references Singers1
         on delete restrict on update restrict;   /* 2 */
   run;
   modify Singers2;
      ic create primary key (LastName FirstName); /* 3 */
   run;
   modify Singers1;
      ic create foreign key (LastName FirstName) references Singers2
         on delete restrict on update restrict;   /* 4 */
   run;
quit;
```

1. The first IC CREATE statement defines a primary key constraint for the data set `Singers1`, for variables `FirstName` and `LastName`.
2. This statement defines a foreign key constraint for the data set `Singers2`, for variables `FirstName` and `LastName`, that references the primary key of the `Singers1` data set. Because the intention is to define overlapping constraints, the referential actions for UPDATE and DELETE must both be RESTRICT.
3. This statement defines a primary key constraint for the data set `Singers2` for variables `LastName` and `FirstName`. Because the same variables are already defined as a foreign key in `Singers2`, the order must be different than in the foreign key constraint in `Singers2`.
4. This statement defines a foreign key constraint for the data set `Singers1`, for variables `LastName` and `FirstName`, that references the primary key of the `Singers2` data set. As in step 3, the order must be different than in the primary key constraint in `Singers1`. As in step 2, due to the overlapping constraint, the referential actions for UPDATE and DELETE must both be RESTRICT.

Due to the overlapping constraints, no changes can be made to the `FirstName` or `LastName` variables in either of the data sets. Other variables can be changed. The following code successfully updates the `Style` variable in the `Singers2` data set for one observation. The output shows the change:

```
proc sql;
   update Singers2
      set Style = 'Broadway'
      where LastName='Streisand';
select * from Singers2;
```

```
quit;
```

Output 9.4 *Output Showing Successful Update*

LastName	FirstName	Style
Kristofferson	Kris	Country
Jones	Tom	Rock
Nelson	Willie	Country
Streisand	Barbra	Broadway
McCartney	Paul	Rock
Travis	Randy	Country

Key Ideas

- Overlapping primary key and foreign key integrity constraints can strengthen the integrity of multiple data sets. However, multiple referential integrity constraints can adversely impact performance.
- The foreign key's update and delete referential actions must both be RESTRICT.
- When the same variables are used in a primary key definition and a foreign key definition, the variables must be defined in a different order in each definition.

See Also

- "Overlapping Primary Key and Foreign Key Constraints" on page 105
- "IC CREATE Statement" in *Base SAS Procedures Guide*
- "UPDATE Statement" in *SAS SQL Procedure User's Guide*

Example: View Integrity Constraints Information

Example Code

The following PROC CONTENTS code shows integrity constraint information for the `player_notes` and `player_stats` data sets. To create these data sets and integrity constraints, see the preceding PROC DATASETS or PROC SQL examples.

```
proc contents data=player_notes;
proc contents data=player_stats;
run;
```

Output 9.5 *Portion of PROC CONTENTS Output for Player_Notes*

Alphabetic List of Integrity Constraints						
#	Integrity Constraint	Type	Variables	Reference	On Delete	On Update
1	foreign_name	Foreign Key	name	WORK.PLAYER_STATS	Set Null	Cascade

Alphabetic List of Indexes and Attributes			
#	Index	Owned by IC	# of Unique Values
1	name	YES	2

Output 9.6 *Portion of PROC CONTENTS Output for Player_Stats*

Alphabetic List of Integrity Constraints							
#	Integrity Constraint	Type	Variables	Where Clause	Reference	On Delete	On Update
1	_CK0001_	Check		nHits>1			
2	_NM0001_	Not Null	nError				
3	primary_name	Primary Key	Name				
	foreign_name	Referential			WORK.PLAYER_NOTES	Set Null	Cascade

Alphabetic List of Indexes and Attributes				
#	Index	Unique Option	Owned by IC	# of Unique Values
1	Name	YES	YES	322

Key Ideas

- PROC CONTENTS and PROC DATASETS report integrity constraint information without special options. In addition, you can print the information about integrity constraints and indexes to an output data set by using the OUT2= option.
- In PROC SQL, the DESCRIBE TABLE and DESCRIBE TABLE CONSTRAINTS statements report integrity constraint characteristics as part of the data set definition or alone, respectively.

See Also

- "Concepts: CONTENTS Procedure" in *Base SAS Procedures Guide*

- "DESCRIBE Statement" in *SAS SQL Procedure User's Guide*

Example: Delete Integrity Constraints

Example Code

The following PROC DATASETS code deletes integrity constraints.

```
proc datasets nolist;
   modify player_notes;
      ic delete foreign_name;
   modify player_stats;
      ic delete _ALL_;
run;
quit;
```

The following PROC SQL code deletes (drops) integrity constraints.

```
proc sql;
   alter table Singers2
      drop constraint _FK0001_;
   alter table Singers1
      drop constraint _FK0001_;
   alter table Singers2
      drop constraint _PK0001_;
   alter table Singers1
      drop constraint _PK0001_;
quit;
```

Key Ideas

- To delete integrity constraints in PROC DATASETS, use the IC DELETE statement. In PROC DATASETS, you can use the _ALL_ keyword to delete all integrity constraints.
- To delete integrity constraints in PROC SQL, use the DROP CONSTRAINT argument.
- When a referential constraint exists, a primary key integrity constraint cannot be deleted until all foreign keys that reference it are deleted. There are no restrictions on deleting foreign keys.

See Also

- "IC DELETE Statement" in *Base SAS Procedures Guide*

- "Creating and Using Integrity Constraints in a Table" in *SAS SQL Procedure User's Guide*
- "ALTER TABLE Statement" in *SAS SQL Procedure User's Guide*

Example: Reactivate an Inactive Integrity Constraint

Example Code

To run this example, create two data sets in separate libraries, and add an inter-library, referential (foreign key) integrity constraint.

```
libname one 'c:\temp1';
libname two 'c:\temp2';

data one.player_stats;
   set sashelp.baseball;
run;
proc datasets lib=one nolist;
   modify player_stats;
      ic create primary_name = primary key(name);
      ic create check(where=(nhits>1));
      ic create not null(nerror);
quit;

data two.player_notes;
   input name $1-18 info $19-48;
   datalines;
Mattingly, Don    uniform number 23
White, Frank      uniform number 20
;
run;
proc datasets lib=two nolist;
   modify player_notes;
      ic create foreign_name =
         foreign key(name) references one.player_stats
         on update cascade
         on delete set null;
quit;
```

Copy the foreign key data set to a different library. Specify CONSTRAINT=YES. The inter-library referential constraint is copied, but is left in an inactive state.

```
libname three 'c:\temp3';
proc datasets nolist;
   copy in=two out=three constraint=yes;
   contents data=three.player_notes;
run;
```

Output 9.7 *CONTENTS Output Showing an Inactive Constraint*

Alphabetic List of Integrity Constraints							
#	Integrity Constraint	Type	Variables	Reference	On Delete	On Update	Inactive
1	foreign_name	Foreign Key	name	ONE.PLAYER_STATS	Set Null	Cascade	YES

When referential constraints span libraries, and you move one of the data sets, you must update the foreign key constraint by specifying the libref of the primary key data set. The following code reactivates the `foreign_name` integrity constraint by referencing the `one` libref.

```
proc datasets lib=three nolist;
   modify player_notes;
      ic reactivate foreign_name references one;
run;
```

A warning in the log states that the constraint is reactivated.

```
WARNING: Integrity constraint foreign_name reactivated.
NOTE: MODIFY was successful for THREE.PLAYER_NOTES.DATA.
```

Key Ideas

- When you copy a data set or library, specify CONSTRAINT=YES to preserve integrity constraints. Inter-library referential constraints are preserved but are left in an inactive state.
- Use the IC REACTIVATE statement of PROC DATASETS to reactivate referential constraints that are inactive as a result of using the COPY, CPORT, CIMPORT, UPLOAD, or DOWNLOAD procedure.

See Also

- "Copying Data Sets That Have Integrity Constraints" on page 106
- "IC REACTIVATE Statement" in *Base SAS Procedures Guide*
- *Base SAS Procedures Guide*
- *SAS/CONNECT User's Guide*

Example: Create Integrity Constraints by Using SCL

Example Code

To add integrity constraints to a data set by using SCL, you must create and build an SCL catalog entry. The following example code creates and compiles catalog entry `example.ic_cat.allics.scl`.

```
INIT:
   put "Test SCL integrity constraint functions start.";
return;

MAIN:
   put "Opening WORK.ONE in utility mode.";
   dsid = open('work.one', 'V');/* Utility mode.*/
   if (dsid = 0) then
      do;
         _msg_=sysmsg();
         put _msg_=;
      end;
      else do;
      if (dsid > 0) then
         put "Successfully opened WORK.ONE in"
             "UTILITY mode.";
   end;

   put "Create a check integrity constraint named teen.";
   rc = iccreate(dsid, 'teen', 'check',
   '(age > 12) && (age < 20)');

   if (rc > 0) then
      do;
       put rc=;
       _msg_=sysmsg();
       put _msg_=;
     end;
     else do;
      put "Successfully created a check"
          "integrity constraint.";
     end;

   put "Create a not-null integrity constraint named nn.";
   rc = iccreate(dsid, 'nn', 'not-null', 'age');

   if (rc > 0) then
      do;
       put rc=;
       _msg_=sysmsg();
       put _msg_=;
     end;
```

```
    else do;
     put "Successfully created a not-null"
         "integrity constraint.";
    end;

  put "Create a unique integrity constraint named uq.";
  rc = iccreate(dsid, 'uq', 'unique', 'age');

  if (rc > 0) then
     do;
      put rc=;
      _msg_=sysmsg();
      put _msg_=;
    end;
    else do;
     put "Successfully created a unique"
         "integrity constraint.";
    end;

 put "Create a primary key integrity constraint named pk.";
 rc = iccreate(dsid, 'pk', 'Primary', 'name');

  if (rc > 0) then
     do;
      put rc=;
      _msg_=sysmsg();
      put _msg_=;
    end;
    else do;
     put "Successfully created a primary key"
         "integrity constraint.";
    end;

  put "Closing WORK.ONE.";
  rc = close(dsid);
  if (rc > 0) then
    do;
     put rc=;
     _msg_=sysmsg();
     put _msg_=;
    end;

  put "Opening WORK.TWO in utility mode.";
  dsid2 = open('work.two', 'V');
     /*Utility mode */
  if (dsid2 = 0) then
    do;
   _msg_=sysmsg();
     put _msg_=;
    end;
    else do;
     if (dsid2 > 0) then
       put "Successfully opened WORK.TWO in"
           "UTILITY mode.";
    end;
```

```
   put "Create a foreign key integrity constraint named fk.";
   rc = iccreate(dsid2, 'fk', 'foreign', 'name',
   'work.one','null', 'restrict');

   if (rc > 0) then
      do;
       put rc=;
       _msg_=sysmsg();
       put _msg_=;
     end;
     else do;
      put "Successfully created a foreign key"
          "integrity constraint.";
     end;

 put "Closing WORK.TWO.";
   rc = close(dsid2);
   if (rc > 0) then
     do;
      put rc=;
      _msg_=sysmsg();
      put _msg_=;
     end;
return;

TERM:
  put "End of test SCL integrity constraint"
      "functions.";
return;
```

The previous code creates the SCL catalog entry. The following code creates two data sets, `one` and `two`, and executes the SCL entry `example.ic_cat.allics.scl`:

```
   /* Submit to create data sets. */

data one two;
   input name $ age;
datalines;
Morris 13
Elaine 14
Tina 15
;

   /* after compiling, run the SCL program */

proc display catalog= example.ic_cat.allics.scl;
run;
```

The following code removes integrity constraints using SCL.

```
TERM:
   put "Opening WORK.TWO in utility mode.";
   dsid2 = open( work.two , V );  /* Utility mode.     */
   if (dsid2 = 0) then
      do;
      _msg_=sysmsg();
      put _msg_=;
   end;
```

```
   else do;
      if (dsid2 > 0) then
         put "Successfully opened WORK.TWO in Utility mode.";
   end;

rc = icdelete(dsid2, fk);
if (rc > 0) then
   do;
      put rc=;
      _msg_=sysmsg();
   end;
else
   do;
      put "Successfully deleted a foreign key integrity constraint.";
   end;
rc = close(dsid2);
return;
```

Key Ideas

- SCL provides the ICCREATE and ICDELETE functions to create and delete integrity constraints.
- SCL provides the ICTYPE, ICVALUE, and ICDESCRIBE functions for getting information about integrity constraints.

See Also

- "Preserving the Integrity of Table Data in SCL Programs" in *SAS Component Language: Reference*

10

Audit Trails

Definition of an Audit Trail

The audit trail is an optional SAS file that you can create in order to log modifications to a SAS data set.

You can use event logging in an audit trail in the following ways:

- track observations that are added, updated, or removed from a data set
- capture, report on, and correct observations that are rejected due to integrity constraints
- track when auditing is suspended and resumed

- analyze usage statistics and patterns

Here are some essential concepts:

- A data set can have one audit trail. The audit trail is stored in a separate file that has the same name as the data set. The member type is AUDIT. The file extension in most operating environments is .sas7baud. See “SAS Files and Member Types” on page 4.
- Use the AUDIT statement in the DATASETS procedure to initiate and control an audit trail. After you initiate an audit trail, each time an observation is added, deleted, or updated, information is written to the audit trail. The information identifies who made the modification, what was modified, and when.
- To view information about the audit trail, or to print the audit trail, use the TYPE=AUDIT data set option.
- An audit trail can negatively impact system performance. Consider suspending the audit trail during large, regularly scheduled batch updates. Audit variables are unavailable when the audit trail is suspended.
- Only SAS can open an audit trail for input. You cannot directly modify the audit trail.
- An ALTER= or PW= password is recommended in order to protect a data set that has an audit trail.
- Audit trails are not supported for SAS views. Audit trails are not supported by engines other than the V9 engine.
- The DLDMGACTION=REPAIR system option or data set option cannot recover an audit trail that has been damaged or deleted. An audit file that is damaged or deleted cannot be terminated and restarted. Recover the audit file from a backup device if available, or re-initiate the audit trail.

Using an Audit Trail to Log Events

Introduction to the Variables and Events in an Audit Trail

In an audit trail, each observation contains the following variables for each logged event:

- a *record image*, which includes every variable of the observation (record) that is being operated on (except for suspend and resume events)
- automatically generated _AT*_ variables that describe the event
- user variables, which are optional variables that you can define in order to add your own logging data

The events that are logged are determined by the LOG statement when the audit trail is initiated (seeTable 10.2 on page 129). You cannot change the logging level

after an audit trail is initiated. If the LOG statement is omitted, the default behavior is to log all images.

_AT*_ Logging Variables

The automatic _AT*_ logging variables are described in the following table.

*Table 10.1 _AT*_ Variables*

_AT*_ Variable	Description
ATDATETIME	Date and time of the event
ATUSERID	Logon user ID that is associated with the event
ATOBSNO	Observation number that is affected by the modification, except when REUSE=YES (because the observation number is always 0)
ATRETURNCODE	Event return code
ATMESSAGE	SAS log message at the time of the event
ATOPCODE	Operation code that describes the event

The _ATOPCODE_ (operation code) values are listed in the following table.

Table 10.2 _ATOPCODE_ Values and Events

ATOPCODE Code	Event	LOG Statement Argument	Record Image
AL	Auditing is resumed	ADMIN_IMAGE=YES	None
AS	Auditing is suspended	ADMIN_IMAGE=YES	None
DA	Data is added	DATA_IMAGE=YES	Added data record image
DD	Data is deleted	DATA_IMAGE=YES	Deleted data record image
DR	Data is read for update	BEFORE_IMAGE=YES	Before-update record image
DW	Data is written for update	DATA_IMAGE=YES	After-update record image

ATOPCODE Code	Event	LOG Statement Argument	Record Image
EA	Error, add failed	ERROR_IMAGE=YES	Rejected record image
ED	Error, delete failed	ERROR_IMAGE=YES	Rejected record image
EU	Error, update failed	ERROR_IMAGE=YES	Rejected record image

User-Defined Logging Variables

You can define additional variables to be logged in the audit trail with each event. For example, you could create a variable in order for end users to specify a reason for each update.

Here are some essential concepts:

- To create a user-defined logging variable, specify USER_VAR when you initiate the audit trail in the DATASETS procedure.
- You also use PROC DATASETS to rename a user variable or modify its attributes.
- For Delete operations, user variable values are not stored in the audit trail.
- Although interactive windows can show some information about audit trails, user variables cannot be displayed or updated. You can display and update user variables in the FSEDIT window of SAS/FSP. User variables are not available for browsing.

Operations That Affect Audit Trails

Copying, Replacing, or Sorting Data Sets That Have an Audit Trail

The following operations do not preserve an audit trail:

- sorting a data set
- replacing a data set
- the COPY procedure or the COPY statement of the DATASETS procedure

The following operations do preserve an audit trail:

- The MIGRATE procedure retains all deleted observations in migrated data sets. Therefore, PROC MIGRATE preserves and migrates audit trails. See "MIGRATE Procedure" in *Base SAS Procedures Guide*.
- The CPORT and CIMPORT procedures preserve the audit trail. However, the audit trail is not historically accurate because deleted observations are removed in order to recover disk space. See "CPORT Procedure" in *Base SAS Procedures Guide* and "CIMPORT Procedure" in *Base SAS Procedures Guide*.
- In a Copy operation on the same host, you can preserve the data set and audit trail by renaming them, by using the generation data sets feature. However, logging stops because neither the auditing process nor the generation data sets feature saves the source program that caused the replacement. See Chapter 11, "Generation Data Sets," on page 143.

CAUTION
If your data sets contain audit trails, do not use operating environment commands to copy, move, or delete the data set. Using operating environment commands to perform these operations makes your files unusable.

Audit Trails and CEDA Processing

When a SAS data set requires processing with CEDA, audit trails are not supported. For example, if you access a data set from a different operating environment, CEDA translates the data set for you but the audit trail is not available. See "Cross-Environment Data Access" in *SAS Programmer's Guide: Essentials*.

Audit Trails in a Shared Environment

The audit trail operates similarly in local and remote environments. The only difference for applications and users networking with SAS/CONNECT and SAS/SHARE is that the audit trail logs events when the observation is written to permanent storage. That is, when the data is written to the remote SAS session or server. Therefore, the time that the transaction is logged on the remote SAS session might be different from the time that is logged on the user's SAS session.

Audit Trails and Indexes

In indexed data sets, the fast-append feature can cause some observations to be written to the audit trail twice. See "Appending to an Indexed Data Set — Fast-Append Method" in *Base SAS Procedures Guide*. The first observation has a DA operation code, and the second observation has an EA operation code. The observations with EA represent the observations that are rejected by index restrictions. For information about DA, EA and other codes, see "_AT*_ Logging Variables" on page 129.

Audit Trails and Generation Data Sets

To suspend, resume, or terminate the audit trail for a generation data set, specify the GENNUM= data set option in the AUDIT statement. You cannot initiate an audit trail for a generation data set. For more information, see Chapter 11, “Generation Data Sets,” on page 143.

Examples: Audit Trails

Example: Initiate an Audit Trail

Example Code

The following DATA step creates the data set `mylib.purchases`. The DATASETS procedure adds an audit trail.

```
libname mylib 'c:\examples';
data mylib.purchases (alter=mypwd);      /* 1 */
   input item $ date date9. price;
   format date mmddyy10. price dollar8.2;
   datalines;
seal 07jan2018 245.00
sander 20jan2018 45.88
;
run;

proc datasets library=mylib nolist;
   audit purchases (alter=mypwd);        /* 2 */
   initiate;
   user_var reason_code $ 30;            /* 3 */
quit;
```

1. The ALTER= data set option protects against altering an audit trail.
2. The AUDIT statement of PROC DATASETS initiates and controls event logging to an audit file. You must provide the correct password in the ALTER= data set option in order to add variables in the data set.
3. The USER_VAR statement defines the user variable `reason_code` for additional logging. When a user updates the SAS data set, they will use the `reason_code` variable to provide a reason for their update.

Key Ideas

- To initiate an audit trail, use the AUDIT statement in PROC DATASETS. You can also use the AUDIT statement to suspend, resume, or terminate (delete) an audit trail.
- A data set can have only one audit file. It resides in the same library as the data set. The audit trail has the same name as the associated data set, but has a member type of AUDIT.
- Use the USER_VAR statement to create user-defined logging variables. User variables are stored in the audit trail, not in the data set.
- An ALTER= or PW= password is recommended for a data set that has an audit trail. If a password other than ALTER= or PW= is assigned, or if no password is assigned, a warning message is written in the SAS log. The message states that the files are not protected from accidental update or deletion.
  - ALTER= protects against deleting or replacing the entire data set, modifying variable attributes, and creating or deleting an index or an audit trail. ALTER= does not protect from reading and writing.
  - PW= protects from reading and writing and all the operations that ALTER= protects.
  - SAS passwords restrict access to data sets within SAS, but cannot prevent data sets from being accessed at the operating environment system level or by an external program.

See Also

- "Using an Audit Trail to Log Events" on page 128
- "AUDIT Statement" in *Base SAS Procedures Guide*

Example: Log Events and Print the Audit Trail

Example Code

This example appends an observation to a data set that has an audit trail. Before you run this example, create `mylib.purchases` and initiate the audit trail as in "Example: Initiate an Audit Trail" on page 132.

```
libname mylib 'c:\examples';
data newpurchases;                                    /* 1 */
   input item $ date date9. price reason_code $ 30.;  /* 2 */
```

```
   format date mmddyy10. price dollar8.2;
   datalines;                                          /* 3 */
filter 19feb2018 10.99 Velasquez invoice
;
proc datasets lib=mylib nolist;
   append base=mylib.purchases data=work.newpurchases; /* 4 */
run;
proc print data=mylib.purchases noobs;                 /* 5 */
proc print data=mylib.purchases (type=audit) noobs;    /* 6 */
   var item reason_code date price
       _atopcode_ _atdatetime_;
run;
```

1. The `newpurchases` data set is created in the temporary Work library.
2. The INPUT statement specifies the user variable `reason_code` along with the data set variables.
3. One observation is created. The `reason_code` for this observation is `Velasquez invoice`.
4. The APPEND statements adds the transaction data set, `newpurchases`, to the end of the BASE= data set, `mylib.purchases`.
5. The PRINT procedure prints the modified `mylib.purchases` data set.
6. PROC PRINT prints selected variables of the audit trail for `mylib.purchases`.

Here is the data set `mylib.purchases` that shows the appended observation. Notice that the user variable `reason_code` is not in the data set.

Output 10.1 *Appended Data Set mylib.purchases*

item	date	price
seal	01/07/2018	$245.00
sander	01/20/2018	$45.88
filter	02/19/2018	$10.99

Here is the PROC PRINT output of the audit trail for `mylib.purchases` that shows selected variables from the appended observation along with `reason_code`, `_atopcode_`, and `_atdatetime_`. The _ATOPCODE_ of `DA` indicates that data was added.

Output 10.2 *The Event Logged in mylib.purchases.audit*

item	reason_code	date	price	_ATOPCODE_	_ATDATETIME_
filter	Velasquez invoice	02/19/2018	$10.99	DA	24MAY2019:10:04:02

Key Ideas

- The audit trail stores modified observations along with two types of audit variables:

- _AT*_ variables automatically store modification data such as the date and time of a modification, the logon user ID that is associated with a modification, and so on.
- User variables are optional variables that you define in order to log data about modifications. User variables are stored in the audit file, not in the data set.

- You can use PROC SQL or PROC PRINT to print the contents of the audit file. Specify `(type=audit)` after the data set name.

See Also

- "Using an Audit Trail to Log Events" on page 128
- "APPEND Statement" in *Base SAS Procedures Guide*

Example: Use Before-Event and After-Event Logging

Example Code

This example modifies the `mylib.purchases` data set to demonstrate before-event and after-event logging.

```
data mylib.purchases;
   modify mylib.purchases;                          /* 1 */
   if item = 'filter' then
      do;
         price = 12.99;
         reason_code = 'data entry correction';     /* 2 */
      end;
   else reason_code = 'no change';                  /* 3 */
run;
proc print data=mylib.purchases noobs;              /* 4 */
proc print data=mylib.purchases (type=audit) noobs; /* 5 */
   var item reason_code date price
       _atopcode_ _atdatetime_;
run;
```

1. The `mylib.purchases` data set is specified for modification.
2. The IF statement selects observations to correct. The price of filters is changed to 12.99. The `reason_code` indicates `data entry correction`.
3. For observations that are not corrected, the `reason_code` indicates `no change`.
4. PROC PRINT prints the modified `mylib.purchases` data set.

5 PROC PRINT prints selected variables of the audit trail for `mylib.purchases`.

Here is the data set `mylib.purchases` that shows the updated value `12.99`.

Output 10.3 *Updated Data Set mylib.purchases*

item	date	price
seal	01/07/2018	$245.00
sander	01/20/2018	$45.88
filter	02/19/2018	$12.99

Here is the PROC PRINT output of the audit trail for `mylib.purchases`. The audit trail shows six new events. The DATA step IF-THEN/ELSE logic caused SAS to read and write all three observations, even though two of the observations did not meet the IF condition. The `DR` code indicates data was read, and a before-event record image was captured. The `DW` code indicates that data was written, and an after-event record image was captured.

Output 10.4 *Multiple Events Logged in mylib.purchases.audit*

item	reason_code	date	price	_ATOPCODE_	_ATDATETIME_
filter	Velasquez invoice	02/19/2018	$10.99	DA	24MAY2019:14:42:35
seal		01/07/2018	$245.00	DR	24MAY2019:14:49:02
seal	no change	01/07/2018	$245.00	DW	24MAY2019:14:49:02
sander		01/20/2018	$45.88	DR	24MAY2019:14:49:02
sander	no change	01/20/2018	$45.88	DW	24MAY2019:14:49:02
filter		02/19/2018	$10.99	DR	24MAY2019:14:49:02
filter	data entry correction	02/19/2018	$12.99	DW	24MAY2019:14:49:02

Key Ideas

- The LOG statement must appear after the INITIATE statement in an AUDIT RUN group. You cannot change the event logging at a later point in time.
- If you do not specify a LOG statement, all events are logged. BEFORE_IMAGE=YES captures the record image before an update event. AFTER_IMAGE=YES captures the record image after an update event.

See Also

- "MODIFY Statement" in *SAS DATA Step Statements: Reference*
- "IF-THEN/ELSE Statement" in *SAS DATA Step Statements: Reference*
- "LOG Statement" in *Base SAS Procedures Guide*

■ "_AT*_ Logging Variables" on page 129

Example: Use an Audit Trail to Capture Rejected Observations

Example Code

In this example, PROC DATASETS adds an integrity constraint to `mylib.purchases`. In the next step, the SQL procedure attempts to insert an observation that fails the integrity constraint. The failed observation and the custom error message are captured in `mylib.purchases.audit`.

```
proc datasets lib=mylib;
   modify purchases (alter=mypwd);
   ic create price_check = check
      (where=((price > 10) and (price < 1000)))         /* 1 */
      message="Price outside the range.";               /* 2 */
run;

proc sql;
   insert into mylib.purchases
      values ("brace","21feb18"d,2787,"Brown project"); /* 3 */
quit;

proc print data=mylib.purchases (type=audit) noobs;
   where _atopcode_ in ("EA","ED","EU");                /* 4 */
   format _atmessage_ $250.;
   var item reason_code date price                      /* 5 */
       _atopcode_ _atmessage_;
run;
```

1. The integrity constraint `price_check` ensures that the `price` is more than 10 and less than 1000.
2. If an observation fails the integrity constraint, a custom message is prepended to the error message in the SAS log: `Price outside the range.`
3. PROC SQL attempts to insert an observation. The `price` value should be 27.87 but is entered as 2787. This value is outside the acceptable range of the `price_check` integrity constraint.
4. In PROC PRINT, the WHERE statement selects error events only.
5. The VAR statement selects variables from the audit trail to print.

Here is the PROC PRINT output, which shows the variables that are selected from the rejected observation and its audit trail logging.

Output 10.5 *Rejected Observation Logged in the Audit Trail*

item	reason_code	date	price	_ATOPCODE_	_ATMESSAGE_
brace	Brown project	02/21/2018	$2787.00	EA	ERROR: Price outside the range. Add/Update failed for data set MYLIB.PURCHASES because data value(s) do not comply with integrity constraint price_check.

Key Ideas

- The audit trail is the only facility in SAS that stores observations from failed Append operations that were rejected by integrity constraints.
- You can write a DATA step to extract rejected observations from the audit trail. Then you can use the error messages that are captured in the audit trail to correct the observations and reapply them to the data set.
- In PROC DATASETS, when you create an integrity constraint, use the MESSAGE= option in the IC CREATE statement to write a custom error message. The custom message is prepended to the SAS log error message when an observation is rejected for failing the integrity constraint. If you print the audit trail, the error message is in the _ATMESSAGE_ variable. If you want to replace the SAS message instead of prepending to it, add the MSGTYPE=USER option.

See Also

- Chapter 9, “Integrity Constraints,” on page 101
- “<MESSAGE='message-string' <MSGTYPE=USER>>” in *Base SAS Procedures Guide*
- “Using an Audit Trail to Log Events” on page 128
- “INSERT Statement” in *SAS SQL Procedure User’s Guide*

Example: View Information about an Audit Trail

Example Code

The following CONTENTS procedure prints information about the `mylib.purchases.audit` audit trail. Use the TYPE= data set option to specify AUDIT.

```
proc contents data=mylib.purchases (type=audit);
run;
```

The PROC CONTENTS output shows all the variables from the corresponding data set, the _AT*_ variables, and any user variables.

Output 10.6 *PROC CONTENTS Output of an Audit Trail*

Data Set Name	MYLIB.PURCHASES.AUDIT	**Observations**	8
Member Type	AUDIT	**Variables**	10
Engine	V9	**Indexes**	0
Created	06/07/2019 12:26:37	**Observation Length**	265
Last Modified	06/07/2019 12:26:41	**Deleted Observations**	0
Protection	ALTER	**Compressed**	NO
Data Set Type	AUDIT	**Sorted**	NO
Label			
Data Representation	WINDOWS_64		
Encoding	wlatin1 Western (Windows)		

Output 10.7 *PROC CONTENTS Output of Data Set Variables and Audit Trail Variables*

Alphabetic List of Variables and Attributes				
#	**Variable**	**Type**	**Len**	**Format**
5	_ATDATETIME_	Num	8	DATETIME19.
10	_ATMESSAGE_	Char	153	
6	_ATOBSNO_	Num	8	
9	_ATOPCODE_	Char	2	
7	_ATRETURNCODE_	Num	8	
8	_ATUSERID_	Char	32	
2	date	Num	8	MMDDYY10.
1	item	Char	8	
3	price	Num	8	DOLLAR8.2
4	reason_code	Char	30	

Key Ideas

- You can read the audit trail with any SAS component that reads a data set such as PROC CONTENTS, PROC PRINT, or PROC SQL.
- The audit trail has the same libref and member name as the data set. Refer to the audit trail by specifying the TYPE=AUDIT data set option.

See Also

- "CONTENTS Statement" in *Base SAS Procedures Guide*
- "TYPE= Data Set Option" in *SAS Data Set Options: Reference*

Example: Modify a User Variable in an Audit Trail

Example Code

The following example uses PROC DATASETS to change the length of a user variable and to rename it.

```
proc datasets lib=mylib nolist;
   modify purchases (alter=mypwd);
     format reason_code $100.;
     rename reason_code=Reason;
quit;
```

Key Ideas

- You can modify the attributes of a user variable without reinitializing the audit trail.
- If the audit trail's associated data set is protected by a password, then that protection is also applied to the audit trail. An ALTER= password does not protect against reading and writing, but it does protect against modifying variables.

See Also

- "FORMAT Statement" in *Base SAS Procedures Guide*
- "RENAME Statement" in *Base SAS Procedures Guide*

Example: Suspend, Resume, or Terminate an Audit Trail

Example Code

The following PROC DATASETS suspends and resumes the audit trail.

```
proc datasets library=mylib nolist;
   audit purchases (alter=mypwd);
   suspend;
run;
proc datasets library=mylib nolist;
   audit purchases (alter=mypwd);
   resume;
quit;

proc print data=mylib.purchases (type=audit) noobs;
   where _atopcode_ in ("AL","AS");
   var _atdatetime_ _atopcode_ _atmessage_;
run;
```

In the PROC PRINT output, the WHERE statement selects administrative events only.

Output 10.8 *Administrative Events Logged in the Audit Trail*

ATDATETIME	_ATOPCODE_	_ATMESSAGE_
07JUN2019:14:34:30	AS	SUSPEND
07JUN2019:14:34:30	AL	RESUME

The following PROC DATASETS terminates and deletes the audit trail.

```
proc datasets library=mylib nolist;
   audit purchases (alter=mypwd);
   terminate;
quit;
```

Key Ideas

- Suspending or resuming an audit trail is an administrative event that is logged in the audit trail.
  - When the audit trail is suspended, the _ATOPCODE_ is AS and the _ATMESSAGE_ is SUSPEND.
  - When the audit trail is resumed, the _ATOPCODE_ is AL and the _ATMESSAGE_ is RESUME.

- When you terminate an audit trail, you delete it. If you want to keep the audit trail, then use SUSPEND instead of TERMINATE.

See Also

- "AUDIT Statement" in *Base SAS Procedures Guide*

11

Generation Data Sets

Definitions for Generation Data Sets

A generation data set is an archived version of a SAS data set that is created when the data set is updated. Multiple copies of a SAS data set are stored in a generation group, with each data set having the same root member name but with a different version number.

Here are some essential concepts for generation data sets:

- You can request a generation group for a SAS data set only. You cannot request a generation group for a SAS view.
- Use the GENMAX= data set option to request a generation group and specify the maximum number of versions. The default is GENMAX=0, which means that the generation data sets feature is not in effect.
- Use the GENNUM= data set option to reference a specific version from a generation group.
  - A positive number is an absolute reference to a historical version by its generation number. For example, GENNUM=1 refers to the oldest version.

- A negative number is a relative reference to a historical version. For example, GENNUM=-1 refers to the most recent version, previous to the base version.

- Passwords for versions in a generation group are maintained as follows:
  - If you assign a password to the base version, the password is maintained in subsequent historical versions. However, the password is not applied to any existing historical versions.
  - If you assign a password to a historical version, the password applies to that individual data set only.
- Do not write to an existing generation data set by using the IMPORT procedure with the REPLACE argument. The behavior depends on whether you specify the GENMAX= data set option:
  - If you specify the GENMAX= data set option, then all existing generations are deleted and replaced with a single new base generation data set.
  - If you omit the GENMAX= data set option, then all existing generations are deleted and replaced with a single new data set by the same name, but it is not a generation data set.

  Instead, use a DATA step with the REPLACE= data set option to replace a data set and maintain the generation group for that data set.
- Generation data sets provide historical versions of a data set. They do not track observation updates for an individual data set. For information about logging each time an observation is added, deleted, or updated, see Chapter 10, “Audit Trails,” on page 127.
- Each historical version is stored as a data set. Therefore, the member type is DATA and the file extension in most operating environments is .sas7bdat. See “SAS Files and Member Types” on page 4.

The following terms are relevant to generation data sets:

base version
: is the most recently created version of a data set. Its name does not have the four-character suffix for the generation number.

generation group
: is a group of data sets that represent a series of replacements to the original data set. The generation group consists of the base version and a set of historical versions.

generation number
: is a consistently increasing number that identifies one of the historical versions in a generation group.

historical version
: is an older copy of the base version of a data set in a generation group.

Naming Conventions for Generation Data Sets

Here is how SAS applies numbers to historical versions when the GENMAX= data set option is in effect:

- The data set name is limited to 28 bytes (rather than 32). Four bytes are reserved for a version number.
- When the data set is replaced, SAS keeps the previous version of the data set. A four-character suffix is appended to the file name. The suffix is the number sign (#) followed by a three-digit generation number.
- Each time the data set is replaced, the generation number is incremented by 1.

  For example, consider the data set `test`, which has GENMAX=5. If `test` is replaced two times, the generation group has the following three files:

  - `test.sas7bdat` is the base (or current) version, which has no suffix.
  - `test#002.sas7bdat` is the most recent version, previous to the base version. It is generation number 2.
  - `test#001.sas7bdat` is the oldest version. It is generation number 1.
- When generations exceed the GENMAX= setting, the oldest generation is deleted. The numbering continues to increment.

  For example, consider the data set `test` after it has been replaced several times. The generation group has the following five files:

  - `test.sas7bdat` is the base version, which has no suffix.
  - `test#006.sas7bdat` is the most recent version, previous to the base version. It is generation number 6.
  - `test#005.sas7bdat` is generation number 5.
  - `test#004.sas7bdat` is generation number 4.
  - `test#003.sas7bdat` is generation number 3.
- The limit for generation numbers is 1,000. After 1,000 generations, the numbering rolls over and begins again at 1.

  For example, consider the data set `test` after it has been replaced over 1,000 times. The generation group has the following five files:

  - `test.sas7bdat` is the base version, which has no suffix.
  - `test#002.sas7bdat` is the most recent version, previous to the base version. It is generation number 2.
  - `test#001.sas7bdat` is generation number 1.
  - `test#000.sas7bdat` is generation number 1000.
  - `test#999.sas7bdat` is generation number 999.

Examples: Generation Data Sets

Example: Archive Data Sets by Invoking a Generation Group

Example Code

This DATA step creates a SAS data set named `mylib.class15up` and invokes a generation group.

```
libname mylib '/mydata';
data mylib.class15up (genmax=3);        /* 1 */
   set sashelp.class;
   where age >= 15;                     /* 2 */
run;
proc print data=mylib.class15up noobs;  /* 3 */
run;
```

1. The data set option GENMAX=3 invokes a generation group by specifying that up to three versions are kept (one base version and two historical versions).
2. The data set selects only the observations where `age` is greater than or equal to 15.
3. The PRINT procedure prints `mylib.class15up`.

Output 11.1 *PROC PRINT of mylib.class15up Data Set*

Name	Sex	Age	Height	Weight
Janet	F	15	62.5	112.5
Mary	F	15	66.5	112.0
Philip	M	16	72.0	150.0
Ronald	M	15	67.0	133.0
William	M	15	66.5	112.0

Key Ideas

- To invoke a generation group and to specify the maximum number of versions to maintain, include the GENMAX= data set option when you create a data set.

- When the data set is replaced, SAS backs up the replaced version by appending a version number to its name. That data set is called a historical version in the generation group. The new version is called the base version, and it does not have a version number.

See Also

- "GENMAX= Data Set Option" in *SAS Data Set Options: Reference*
- "GENNUM= Data Set Option" in *SAS Data Set Options: Reference*

Example: View Information about a Generation Group

Example Code

This example replaces a data set to demonstrate generations. Before you run this example, create `mylib.class15up` and invoke a generation group by running "Example: Archive Data Sets by Invoking a Generation Group" on page 146.

```
proc sort data=mylib.class15up;      /* 1 */
   by age;
run;
proc contents data=mylib.class15up;  /* 2 */
run;
```

1. The SORT procedure does not specify an OUT= data set, so it replaces the `mylib.class15up` data set.
2. The CONTENTS procedure does not specify the GENNUM= data set option, so the base version is used. The output shows that the maximum generations is 3. The next generation number is 2, which confirms that the data set has been replaced once.

Output 11.2 *Portion of PROC CONTENTS Output for Generation Group*

Data Set Name	MYLIB.CLASS15UP	**Observations**	5
Member Type	DATA	**Variables**	5
Engine	V9	**Indexes**	0
Created	06/13/2019 15:48:59	**Observation Length**	40
Last Modified	06/13/2019 15:48:59	**Deleted Observations**	0
Protection		**Compressed**	NO
Data Set Type		**Sorted**	YES
Max Generations	3		
Next Generation Num	2		
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	latin1 Western (ISO)		

Key Ideas

- To view information about the base data set in a generation group, either omit the GENNUM= data set option or specify GENNUM=0.
- To view information about a historical version in a generation group, specify the GENNUM= data set option.
- Only replaced data sets are kept in a generation group. For event logging in an updated data set, see Chapter 10, “Audit Trails,” on page 127.

See Also

- “SORT Procedure” in *Base SAS Procedures Guide*
- “CONTENTS Procedure” in *Base SAS Procedures Guide*

Example: Append One Version of a Generation Group

Example Code

In this example, the APPEND statement of PROC DATASETS appends a historical version of `mylib.class15up` to the base version.

```
proc datasets lib=mylib nolist;
   append base=class15up(gennum=0) data=class15up(gennum=1);
quit;
proc print data=mylib.class15up noobs;
run;
```

In "Example: View Information about a Generation Group" on page 147, `class15up` was sorted by `age`. The unsorted data set was kept as generation number 1. The PROC PRINT output shows that the unsorted data set is appended to the sorted data set.

Output 11.3 *PROC PRINT of Appended Data Set*

Name	Sex	Age	Height	Weight
Janet	F	15	62.5	112.5
Mary	F	15	66.5	112.0
Ronald	M	15	67.0	133.0
William	M	15	66.5	112.0
Philip	M	16	72.0	150.0
Janet	F	15	62.5	112.5
Mary	F	15	66.5	112.0
Philip	M	16	72.0	150.0
Ronald	M	15	67.0	133.0
William	M	15	66.5	112.0

Key Ideas

- You can use the GENNUM= data set option to append a specific historical version to another data set.
- In the APPEND statement of PROC DATASETS or in PROC APPEND, if you omit GENNUM=, or specify GENNUM=0, then the base version of the data set is used.

See Also

- "APPEND Statement" in *Base SAS Procedures Guide*
- "Naming Conventions for Generation Data Sets" on page 145

Example: Delete Versions in a Generation Group

Example Code

The following example code creates a generation group that will be used to demonstrate deletion behavior.

```
libname mylib '/mydata';
data mylib.test (genmax=3);
   x='first data set created';
run;
data mylib.test ;
   x='second data set created';
run;
data mylib.test ;
   x='third data set created';
run;
proc print data=mylib.test(gennum=1) noobs;
run;
proc print data=mylib.test(gennum=2) noobs;
run;
proc print data=mylib.test(gennum=0) noobs;
run;
```

Here are the three data sets in the generation group before deletion:

Output 11.4 *PROC PRINT of test#001, test#002, and test*

x
first data set created

x
second data set created

x
third data set created

In the following example code, PROC DATASETS deletes the base version of `test`.

```
proc datasets lib=mylib nolist;
   delete test;
quit;
proc print data=mylib.test(gennum=1) noobs;
run;
proc print data=mylib.test(gennum=0) noobs;
run;
```

The PROC PRINT output shows that when `test` is deleted, it is replaced by the most recent version. In other words, `test#002` is now `test`.

Output 11.5 *PROC PRINT of test#001 and test*

x
first data set created

x
second data set created

If you try to delete `test#002`, SAS cannot find it.

```
proc datasets lib=mylib nolist;
   delete test (gennum=2);
quit;
```

Example Code 11.1 *SAS Log Showing Deletion Is Successful*

```
NOTE: The file MYLIB.TEST (memtype=DATA gennum=2) was not found, but appears on a
DELETE
      statement.
```

Key Ideas

- You can delete one generation data set or an entire generation group by specifying the GENNUM= data set option in the DELETE statement of PROC DATASETS.
- If you omit GENNUM= or specify GENNUM=0, you delete the base version. The most recent historical version becomes the base version. Because that data set is now the base version, its suffix and generation number are removed.

  This behavior differs from other PROC DATASETS statements, where omitting GENNUM= or specifying GENNUM=0 applies an action to the entire generation group.
- Here are additional values for GENNUM= in the DELETE statement of PROC DATASETS:
  - GENNUM=ALL deletes all data sets in the generation group, including the base version.

  - GENNUM=HIST deletes all data sets in the generation group, except the base version.
  - GENNUM=REVERT is the same as GENNUM=0. The base version is deleted and the most recent historical version becomes the base version.
- After you delete a historical version, you cannot specify it in GENNUM= as an absolute reference.
- After you delete a historical version, relative references in GENNUM= reflect the current ordering. Deleted historical versions are not counted.

See Also

- "GENMAX= Data Set Option" in *SAS Data Set Options: Reference*
- "DELETE Statement" in *Base SAS Procedures Guide*
- "PRINT Procedure" in *Base SAS Procedures Guide*

Example: Modify the Number of Versions or End a Generation Group

Example Code

In this example, the MODIFY statement in PROC DATASETS changes the number of generations for data set `class15up` to 2. The oldest versions are deleted. The generation group contains only the base version and the previous version.

```
proc datasets library=mylib nolist;
   modify class15up (genmax=2);
quit;
```

In this example, GENMAX=0 deletes all historical versions and ends the generation group.

```
proc datasets library=mylib nolist;
   modify class15up (genmax=0);
quit;
```

Key Ideas

- When you modify the attributes of a data set, you can increase or decrease the number of versions for an existing generation group.
- If you set GENMAX=0, you delete all historic versions, and you end (stop) generations. The base version is not deleted.

See Also

- "GENMAX= Data Set Option" in *SAS Data Set Options: Reference*
- "MODIFY Statement" in *Base SAS Procedures Guide*

Example: Copy a Generation Group

Example Code

In this example, the COPY statement of PROC DATASETS copies a generation group for the `test` data set from the `mylib` library to the `newlib` library.

```
libname mylib '/mydata';
libname newlib '/mydata/archive';
proc datasets;
   copy in=mylib out=newlib;
   select test;
quit;
```

CAUTION
Do not use operating system tools when managing generation data sets. This can cause limited access to the generation group files. Instead, use SAS tools such as PROC DATASETS or PROC COPY.

Key Ideas

- Use the COPY statement in the DATASETS procedure or the COPY procedure to copy a generation group.
- The COPY statement does not support the GENNUM= data set option. You cannot copy one version from a generation group.
- The MIGRATE procedure, the CPORT procedure, and the CIMPORT procedure also support generation groups.

See Also

- "COPY Statement" in *Base SAS Procedures Guide*
- "SAS Libraries" in *SAS Programmer's Guide: Essentials*

Example: Rename One Version or an Entire Generation Group

Example Code

The following PROC DATASETS renames one historic version by specifying the GENNUM= data set option.

```
proc datasets lib=mylib nolist;
   change test (gennum=-1) = test_copy;
quit;
```

The following PROC DATASETS renames the entire generation group because it omits the GENNUM= data set option.

```
proc datasets lib=mylib nolist;
   change test = mytest;
quit;
```

Key Ideas

- To rename one generation data set in a generation group, specify the GENNUM= data set option.

  When you rename one historical version in a generation group, you drop the numbered suffix. The file is not deleted when the number of generations exceeds the GENMAX= specification. Changing the name of one generation data set is a strategy for keeping a specific backup version of the data set.
- To rename all data sets in a generation group, either omit the GENNUM= data set option or specify GENNUM=0 or GENNUM=ALL.

  When you rename the entire generation group, the historical versions retain their numbered suffixes.

See Also

- “CHANGE Statement” in *Base SAS Procedures Guide*
- “GENNUM= Data Set Option” in *SAS Data Set Options: Reference*

Example: Understand Generation Numbers after a Rollover

Example Code

In this example, the macro `%create` creates a simple `test` data set *n* times.

```
libname mylib '/mydata';
%macro create(n);
   %do i=1 %to &n;                  /* 1 */
      data mylib.test(genmax=5);    /* 2 */
         x=1;
      run;
   %end;
%mend create;

%create(1003);                      /* 3 */

proc datasets lib=mylib;            /* 4 */
run;
proc datasets lib=mylib;
   contents data=test (gennum=-1); /* 5 */
quit;
```

1. The macro submits an iterative %DO group that runs *n* times.
2. The GENMAX= data set option requests a generation group and specifies a maximum of 5 versions (one base version and four historical versions).
3. The `%create` macro is invoked with *n*=1003. The `mylib.test` data set is created 1,003 times. Only 5 versions are kept in the generation group due to GENMAX=5.
4. PROC DATASETS prints the directory, showing the base version and four historical versions. The directory is listed in numeric order, not historical order. (In the user's file system, be aware that the historical version that has generation number 1000 is named `test#000.sas7bdat`.)
5. PROC DATASETS prints the contents of the GENNUM=–1 data set, which is the most recent historical version previous to the base version. Because the version numbering has rolled over, the most recent historical version is generation number 2.

Output 11.6 *PROC DATASETS Output of Directory after Rollover*

#	Name	GenNum	Member Type	File Size	Last Modified
1	TEST		DATA	128KB	06/12/2019 14:48:38
2	TEST	1	DATA	128KB	06/12/2019 14:48:38
3	TEST	2	DATA	128KB	06/12/2019 14:48:38
4	TEST	999	DATA	128KB	06/12/2019 14:48:38
5	TEST	1000	DATA	128KB	06/12/2019 14:48:38

Output 11.7 *GENNUM=-1 Data Set test#002*

Data Set Name	MYLIB.TEST (gennum=1002)	Observations	1
Member Type	DATA	Variables	1
Engine	V9	Indexes	0
Created	06/12/2019 14:48:38	Observation Length	8
Last Modified	06/12/2019 14:48:38	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Generation Number	2		

Key Ideas

- The limit for generation numbers is 1,000. After 1,000 generations, the numbering rolls over and begins again at 1.
- In the file system, generation number 1000 is represented by the suffix `#000`. Specify GENNUM=1000 or a relative reference (negative number).
- Deletion behavior does not change after a rollover. If you delete a historical version, relative references in GENNUM= reflect the current ordering. Deleted historical versions are not counted.

See Also

- "Naming Conventions for Generation Data Sets" on page 145
- "%DO macro-variable=start %TO stop Macro Statement" in *SAS Macro Language: Reference*
- "CONTENTS Statement" in *Base SAS Procedures Guide*

12

Compression

Definitions for Data Set Compression

Compressing a SAS data set is a process that reduces the number of bytes required to represent each observation. In a compressed file, each observation is a varying-length record. In an uncompressed file, each observation is a fixed-length record.

SAS provides two types of compression:

- COMPRESS=CHAR uses the RLE (Run Length Encoding) compression algorithm. RLE compresses observations by reducing repeated runs of the same character (including blanks) to two-byte or three-byte representations.
- COMPRESS=BINARY uses the RDC (Ross Data Compression) algorithm. RDC combines run-length encoding and sliding-window compression to compress the file by representing repeated byte patterns more efficiently.

Deciding Whether to Use Compression

Advantages of compression include the following:

- reduced storage requirements for the file
- fewer I/O operations necessary to read from or write to the data during processing

Disadvantages of compression include the following:

- increased CPU resource requirements to read a compressed file because of the overhead of uncompressing each observation and, if updating, compressing again when writing to disk
- situations when the resulting file size can increase rather than decrease

Here are some essential concepts about compression:

- After a data set is compressed, the setting is a permanent attribute. To change the setting, you must re-create or copy the data set.
- If you do not know details about the data, COMPRESS=BINARY might be the best choice. Repeated sequences of bytes are common in data, and compressing them tends to save more space than compressing repeated runs of the same character.
- If conserving space is a primary concern, then testing is recommended. To compare the effectiveness of CHAR and BINARY, run a test and check the SAS log. When you create a compressed data set, SAS writes a note to the log, indicating the percentage of reduction that is obtained by compressing the file. SAS calculates the compression percentage by comparing the size of the compressed file with the size of an uncompressed file of the same page size and record count.
- The V9 engine compresses one observation at a time, and adds a fixed-length block of data to each observation. Because of the additional block of data, some data sets would result in a larger file size if compressed. For example, a data set with an extremely short observation length would not benefit from compression.

  When a request is made to compress a data set, SAS attempts to determine whether compression would increase the size of the file. SAS examines the lengths of the variables. If the additional data that is needed for compression would increase the file size, then compression is disabled and a message is written to the SAS log. The size of the additional data depends on the operating environment. For example, on Windows the additional data is 12 bytes per observation for 32-bit and 64-bit SAS. On 64-bit UNIX, the additional data is 24 bytes.

The following table summarizes the types of data that can benefit from CHAR and BINARY compression.

Table 12.1 Use Cases for CHAR and BINARY Compression

Data Than Can Benefit from COMPRESS=CHAR	Data Than Can Benefit from COMPRESS=BINARY
■ single bytes that are repeated ■ character variables that are defined with a large length but contain mostly short strings ■ numeric variables that are integers	■ strings of bytes that are repeated ■ wide data sets, with an observation length that is several hundred bytes or larger ■ data sets whose contents are not well known

Operations That Affect Compression

Be aware of the following interactions:

- When you sort a compressed data set, the sort's temporary utility file can be very large.
- You cannot compress SAS views, because they contain no data.
- The COPY procedure (or the COPY statement of PROC DATASETS) retains the compression attribute unless you specify the NOCLONE option.
- The MIGRATE procedure retains the compression attribute.
- The CPORT and CIMPORT procedures retain the compression attribute.
- Using the DATA step to copy or replace a data set does not retain the compression attribute.

Examples: Compression

Example: Compress a Data Set by Using COMPRESS=CHAR

Example Code

This example uses the COMPRESS=CHAR data set option.

```
libname mylib 'c:\examples';
data mylib.charrepeats_char(compress=char);
   length ca $ 200;
   do i=1 to 100000;
      ca='aaaaaaaaaaaaaaaaaaaaaa';
      cb='bbbbbbbbbbbbbbbbbbbbbb';
      cc='cccccccccccccccccccccc';
      output;
   end;
run;
```

The following message is written to the SAS log.

Example Code 12.1 *SAS Log Message about Compression*

```
NOTE: Compressing data set WORK.MYLIBCHARREPEATS_CHAR decreased size by 88.55
percent.
      Compressed is 45 pages; un-compressed would require 393 pages.
```

For comparison, try COMPRESS=BINARY on the same data.

```
data mylib.charrepeats_bin(compress=binary);
   length ca $ 200;
   do i=1 to 100000;
      ca='aaaaaaaaaaaaaaaaaaaaaa';
      cb='bbbbbbbbbbbbbbbbbbbbbb';
      cc='cccccccccccccccccccccc';
      output;
   end;
run;
```

The following message is written to the SAS log. COMPRESS=BINARY decreased the size by 87.28%. In the previous example code, COMPRESS=CHAR decreased the size by 88.55%. Therefore, COMPRESS=CHAR is the better choice for this data set.

Example Code 12.2 *SAS Log Message about Compression*

```
NOTE: Compressing data set MYLIB.CHARREPEATS_BIN decreased size by 87.28 percent.
      Compressed is 50 pages; un-compressed would require 393 pages.
```

Key Ideas

- The COMPRESS= option is available as a data set option, a LIBNAME statement option, and a system option.
- By default, a SAS data set is not compressed. The COMPRESS= system option is shipped with no default, and if you do not specify the COMPRESS= option, the compress attribute is not set. The default behavior is equivalent to setting COMPRESS=NO.
- Specify COMPRESS=CHAR to use the RLE (Run Length Encoding) compression algorithm. COMPRESS=CHAR is recommended when single bytes are repeated.

- Specify COMPRESS=BINARY to use the RDC (Ross Data Compression) compression algorithm. COMPRESS=BINARY is recommended when strings of bytes are repeated.
- When you create a compressed data set, SAS writes a note to the log, indicating the percentage of reduction that is obtained by compressing the file. SAS obtains the compression percentage by comparing the size of the compressed file with the size of an uncompressed file of the same page size and record count.

See Also

- "COMPRESS= Data Set Option" in *SAS Data Set Options: Reference*
- "COMPRESS= LIBNAME Statement Option" on page 26
- "COMPRESS= System Option" in *SAS System Options: Reference*

Example: Compress a Data Set by Using COMPRESS=BINARY

Example Code

This example uses the COMPRESS=BINARY data set option.

```
libname mylib 'c:\examples';
data mylib.stringrepeats_bin(compress=binary);
   length cabcd $ 200;
   do i=1 to 1000000;
      cabcd='abcdabcdabcdabcdabcdabcdabcdabcd';
      cefgh='efghefghefghefghefghefghefghefgh';
      cijkl='ijklijklijklijklijklijklijklijkl';
      output;
   end;
run;
```

The following message is written to the SAS log:

Example Code 12.3 *SAS Log Message about Compression*

```
NOTE: Compressing data set MYLIB.STRINGREPEATS_BIN decreased size by 70.27 percent.
      Compressed is 1239 pages; un-compressed would require 4167 pages.
```

For comparison, try COMPRESS=CHAR on the same data.

```
data mylib.stringrepeats_char(compress=char);
   length cabcd $ 200;
   do i=1 to 1000000;
      cabcd='abcdabcdabcdabcdabcdabcdabcdabcd';
      cefgh='efghefghefghefghefghefghefghefgh';
```

```
        cijkl='ijklijklijklijklijklijklijklijkl';
        output;
    end;
run;
```

The following message is written to the SAS log. COMPRESS=CHAR compressed the size by 57.04%. In the previous example code, COMPRESS=BINARY decreased the size by 70.27%. Therefore, COMPRESS=BINARY is the better choice for this data set.

Example Code 12.4 *SAS Log Message about Compression*

```
NOTE: Compressing data set MYLIB.STRINGREPEATS_CHAR decreased size by 57.04 percent.
      Compressed is 1790 pages; un-compressed would require 4167 pages.
```

Key Ideas

- The COMPRESS= option is available as a data set option, a LIBNAME statement option, and a system option.
- By default, a SAS data set is not compressed. The COMPRESS= system option is shipped with no default, and if you do not specify the COMPRESS= option, the compress attribute is not set. The default behavior is equivalent to setting COMPRESS=NO.
- Specify COMPRESS=CHAR to use the RLE (Run Length Encoding) compression algorithm. COMPRESS=CHAR is recommended when single bytes are repeated.
- Specify COMPRESS=BINARY to use the RDC (Ross Data Compression) compression algorithm. COMPRESS=BINARY is recommended when strings of bytes are repeated.
- When you create a compressed data set, SAS writes a note to the log, indicating the percentage of reduction that is obtained by compressing the file. SAS obtains the compression percentage by comparing the size of the compressed file with the size of an uncompressed file of the same page size and record count.

See Also

- “COMPRESS= Data Set Option” in *SAS Data Set Options: Reference*
- “COMPRESS= LIBNAME Statement Option” on page 26
- “COMPRESS= System Option” in *SAS System Options: Reference*

Example: View Information about a Compressed Data Set

Example Code

This example prints information about the `mylib.charrepeats_char` data set.

```
libname mylib 'c:\examples';
proc datasets lib=mylib nolist;
   contents data=charrepeats_char;
quit;
```

Here is the output, showing CHAR compression.

Output 12.1 *PROC DATASETS Output*

Data Set Name	MYLIB.CHARREPEATS_CHAR	**Observations**	100000
Member Type	DATA	**Variables**	4
Engine	V9	**Indexes**	0
Created	06/18/2019 11:30:40	**Observation Length**	252
Last Modified	06/18/2019 11:30:40	**Deleted Observations**	0
Protection		**Compressed**	CHAR
Data Set Type		**Reuse Space**	NO
Label		**Point to Observations**	YES
Data Representation	WINDOWS_64	**Sorted**	NO
Encoding	wlatin1 Western (Windows)		

Key Ideas

- Use PROC CONTENTS or the CONTENTS statement of PROC DATASETS to show information about a data set. If the data set is compressed, the **Compression** field in the output shows the type of compression. The **Reuse Space** field shows whether the REUSE= option is set to YES or NO.
- Information about compression is also written to the SAS log when you create a compressed data set.

See Also

- "CONTENTS Statement" in *Base SAS Procedures Guide*

Example: Save Disk Space by Specifying REUSE=YES

Example Code

The following COPY procedure copies an uncompressed data set.

```
options compress=binary reuse=yes;      /* 1 */
libname mylib 'c:\examples';
proc copy in=sashelp out=mylib noclone; /* 2 */
   select baseball;
run;
```

1. The option REUSE=YES is not supported as a LIBNAME statement option, and PROC COPY does not support data set options. Therefore, REUSE=YES is specified as a system option.
2. The NOCLONE option of PROC COPY specifies to not copy a data set's compression attribute.

Key Ideas

- When you create a compressed data set, you can also specify REUSE=YES (as a data set option or system option) to track and reuse space. With REUSE=YES, new observations are inserted in available space when other observations are updated or deleted. When the default REUSE=NO is in effect, new observations are appended to the existing file.
- If you set REUSE=YES, then SAS automatically sets POINTOBS=NO. When POINTOBS=NO, the compressed data set cannot be processed with random access (by observation number). Sequential access is used.
- By default, PROC COPY (or the COPY statement of PROC DATASETS) copies a data set's compression attribute. Specify the NOCLONE option to disregard many data set attributes. (For details about the attributes that are affected, see CLONE | NOCLONE .)

See Also

- "REUSE= Data Set Option" in *SAS Data Set Options: Reference*
- "REUSE= System Option" in *SAS System Options: Reference*
- CLONE | NOCLONE

Example: When a Compression Request Is Disabled

Example Code

This example creates a small data set and requests compression.

```
data one (compress=char);
   length x y $2;
   input x y;
   datalines;
ab cd
;
```

SAS determines that it is not possible for the compressed data set to be smaller than an uncompressed one.

Example Code 12.5 *SAS Log Output When Compression Request Is Disabled*

```
NOTE: Compression was disabled for data set WORK.ONE because compression
      overhead would increase the size of the data set.
NOTE: The data set WORK.ONE has 1 observations and 2 variables.
```

Key Ideas

- The V9 engine compresses one observation at a time, and adds a fixed-length block of data to each observation. Because of the additional block of data, some data sets would result in a larger file size if compressed. For example, a data set with an extremely short observation length would not benefit from compression.
- When a request is made to compress a data set, SAS attempts to determine whether compression would increase the size of the file. SAS examines the lengths of the variables. If the additional data that is needed for compression would increase the file size, then compression is disabled and a message is written to the SAS log. The size of the additional data depends on the operating environment. For example, on Windows the additional data is 12 bytes per

observation for 32-bit and 64-bit SAS. On 64-bit UNIX, the additional data is 24 bytes.

See Also

- “Deciding Whether to Use Compression” on page 158

13

Catalogs

Definition of a SAS Catalog

A SAS catalog is a SAS file that stores many different types of information (such as user-defined formats or informats, macros, SAS/AF applications, and SAS/GRAPH output) in smaller units called catalog entries.

Here are some essential concepts about catalogs:

- Each entry has an entry type that identifies its purpose to SAS. One catalog can contain multiple types of catalog entries. See “Catalog Names and Catalog Entries” on page 168.

- The SAS member type is CATALOG. The file extension in most operating environments is .sas7bcat. See "SAS Files and Member Types" on page 4.
- Catalog entries are generally accessed automatically by SAS when the information stored in them is required for processing. In some SAS software products, you must specify the catalog entry in various procedures. Because the requirements vary according to the SAS procedure or software product, see the appropriate procedure or product documentation for details.
- Catalogs are not supported under CEDA processing. See "Operations That Affect Catalogs" on page 171. Storing data and programs in text files could be a better practice than storing them in a catalog.

Catalog Names and Catalog Entries

SAS catalog entries are fully identified by a four-level name in the following form:

libref.catalog-name.entry-name.entry-type

The catalog name must be a valid SAS name. SAS assigns the entry type when the entry is created.

Usually, you specify the two-level name for an entire catalog, *libref.catalog-name*. However, some SAS procedures also require the catalog entry name and entry type. In some contexts, you can use the catalog entry name alone.

Catalog entries can contain user-defined formats or informats, macros, SAS/AF applications, SAS/GRAPH output, SAS code, SCL code, data, and other entry types that are specific to various SAS procedures.

Some catalogs are created automatically by SAS. For example, the Sasuser.Profile catalog entries contain system information such as function key definitions, fonts for graphics applications, window attributes, and other information from interactive windowing procedures. See "Sasuser.Profile Catalog" on page 168.

Sasuser.Profile Catalog

Definition of a Sasuser.Profile Catalog

The Sasuser.Profile catalog is available for customizing how you work in the SAS windowing environment. SAS uses this catalog to store function key definitions, fonts for graphics applications, window attributes, and other information from interactive windowing procedures. See also "Sasuser Library" in *SAS Programmer's Guide: Essentials*.

Default Settings

The default settings for your SAS session are stored in several catalogs in the Sashelp library. If you do not make any customizations to key settings or other options, SAS uses the default settings. If you make changes, the new information is stored in your Sasuser.Profile catalog. To restore the original default settings, use the CATALOG procedure (or the CATALOG window in the SAS windowing environment) to delete the appropriate entries from your Sasuser.Profile catalog. Then, SAS uses the default setting from the Sashelp library.

How SAS Creates and Uses Sasuser.Profile

SAS creates the Sasuser.Profile catalog the first time it tries to refer to it and discovers that it does not exist. If you are using an interactive windowing environment, the Sasuser.Profile catalog is created during system initialization in your first SAS session. If you use one of the other modes of execution, the Sasuser.Profile catalog is created the first time you execute a SAS procedure that requires it.

At SAS start-up, SAS checks for an existing uncorrupted Sasuser.Profile catalog. If this catalog is found, then SAS copies the Sasuser.Profile catalog to Sasuser.Profbak. The backup is used if the Sasuser.Profile catalog becomes corrupted.

The information in the Sasuser.Profile catalog is accessed automatically by SAS when it is needed for processing. SAS stores each Sasuser.Profile catalog entry with the appropriate entry name and entry type.

How SAS Recovers Locked or Corrupted Profile Catalogs

Occasionally, a Sasuser.Profile catalog becomes locked or corrupted. SAS uses Sashelp.Profile and Sasuser.Profbak to replace the locked or corrupted catalog.

If your Sasuser.Profile catalog is locked, SAS checks for Sashelp.Profile. If Sashelp.Profile exists, SAS copies it to Work.Profile and then saves the customizations in Work.Profile instead of in Sasuser.Profile. The following notes appear in the SAS log:

```
ERROR: Expecting page 1, got page -1 instead.
ERROR: Page validation error while reading SASUSER.PROFILE.CATALOG.
NOTE: Unable to open SASUSER.PROFILE. WORK.PROFILE will be opened instead.
NOTE: SASHELP.PROFILE has been copied to WORK.PROFILE.
NOTE: All profile changes will be lost at the end of the session.
```

If your Sasuser.Profile catalog is corrupted, SAS copies the corrupted catalog to Sasuser.Badpro. SAS then checks for Sasuser.Profbak. If Sasuser.Profbak exists,

SAS copies it to Sasuser.Profile. Any changes made to the Sasuser.Profile catalog during the previous session are lost. The following notes appear in the SAS log:

```
ERROR: Expecting page 1, got page -1 instead.
ERROR: Page validation error while reading SASUSER.PROFILE.CATALOG.
NOTE: A corrupt SASUSER.PROFILE has been detected. A PROFILE catalog can
      become corrupt when a SAS session is prematurely terminated.
NOTE: SASUSER.PROFILE.CATALOG has been renamed to SASUSER.BADPRO.CATALOG.
NOTE: SASUSER.PROFILE.CATALOG has been restored from SASUSER.PROFBAK.CATALOG.
NOTE: Changes made to SASUSER.PROFILE.CATALOG during the previous SAS session
      have been lost. The type of data stored in the PROFILE catalog is
      typically related to SAS session customizations such as key
      definitions, fonts for graphics, and window attributes.
```

If your Sasuser.Profile catalog is corrupted and there is no Sasuser.Profbak, SAS checks for Sashelp.Profile. If Sashelp.Profile exists, SAS copies it to Work.Profile and then saves the customizations to Work.Profile instead of to Sasuser.Profile. The following notes appear in the SAS log:

```
ERROR: Expecting page 1, got page -1 instead.
ERROR: Page validation error while reading SASUSER.PROFILE.CATALOG.
NOTE: Unable to open SASUSER.PROFILE. WORK.PROFILE will be opened instead.
NOTE: SASHELP.PROFILE has been copied to WORK.PROFILE.
NOTE: All profile changes will be lost at the end of the session.
```

Catalog Concatenation

You can logically combine two or more SAS catalogs by concatenating them. Concatenation enables you to access the contents of several catalogs, using one catalog name. There are two ways to concatenate catalogs:

LIBNAME catalog concatenation
: results from a concatenation of libraries through a LIBNAME statement. When two or more libraries are logically combined through concatenation, any catalogs with the same name in each library become logically combined as well. See “Example: Concatenate Catalogs by Using Library Concatenation” on page 179.

CATNAME catalog concatenation
: is a concatenation that is specified by the global CATNAME statement in which the catalogs to be concatenated are specifically named. During CATNAME catalog concatenation, a logical catalog is created in memory. See “Example: Concatenate Catalogs by Using the CATNAME Statement” on page 177.

Here are the rules for catalog concatenation. These rules apply to the LIBNAME statement and the CATNAME statement:

- When a catalog entry is open for input or update, the concatenation is searched and the first occurrence of the specified entry is used.
- When an item is open for output, it is created in the catalog that is listed first in the concatenation.
  - A new catalog entry is created in the first catalog even if there is an item with the same name in another part of the concatenation.

  - If the first catalog does not exist, then the item is written to the next catalog that exists in the concatenation.
- When you delete or rename a catalog entry, only the first occurrence of the entry is affected.
- When a list of catalog entries is displayed, only one occurrence of the catalog entry is shown. Even if a catalog entry occurs multiple times in the concatenation, only the first occurrence is shown.

See also "Library Concatenation" in *SAS Programmer's Guide: Essentials*.

Operations That Affect Catalogs

CEDA and Catalogs

When the encoding or data representation of a SAS file does not match the encoding or data representation of the SAS session, cross-environment data access (CEDA) is used to access the data. See "Cross-Environment Data Access" in *SAS Programmer's Guide: Essentials*.

However, catalogs are not supported under CEDA processing. To avoid CEDA, you can copy or migrate a catalog to the new encoding or data representation.

Copying or Migrating Catalogs

The following procedures can be used to copy or migrate catalogs. Catalog entries might need to be updated in the target environment (for example, code could contain hardcoded pathnames).

If you copy or migrate between source and target environments with different encodings, be aware of the following issues:

- Catalog entries do not include encoding in the metadata. SAS assumes that all catalog entries have the current session encoding or a compatible encoding. Note that if a catalog entry contains 7-bit ASCII characters (U.S. English) only, then the catalog entry can be used in any other ASCII session, including UTF-8. Other ASCII encodings use the high-order bit for different national characters.
- If you rebuild a catalog in the target environment (for example, a formats catalog), truncation could be a concern. If the target encoding uses more bytes than the source encoding to represent the characters, truncation can occur if the column length does not accommodate the larger character size.
- Even when the encoding is compatible, you might experience transcoding errors that are not written to the log. For a summary of potential transcoding errors, see "How Character Encoding Affects Compatibility" on page 206.

Table 13.1 *Ways to Copy or Migrate Catalogs*

Procedure	Are Catalogs Supported When CEDA Is Invoked?	Examples
COPY statement in PROC CATALOG	No	"Copying, Deleting, and Moving Catalog Entries from Multiple Catalogs" in *Base SAS Procedures Guide*
COPY procedure (or the COPY statement in the DATASETS procedure)	No	"Manipulating SAS Files" in *Base SAS Procedures Guide*
FORMAT procedure	Use with CNTLOUT= and CNTLIN= options to avoid CEDA restrictions.	"Example: Avoid Truncation in Formats When Migrating Catalogs" on page 238
MIGRATE procedure Stored, compiled macros that are stored in catalogs cannot be migrated. Instead, you must recompile the source code in the target environment. As a best practice, always specify the SOURCE option in the %MACRO statement to save the source code for macro definitions. You could also save the source code in a separate file (for example, a text file).	If CEDA is invoked, SAS/CONNECT or SAS/SHARE is also required. Transcoding errors can occur if the encodings are not compatible.	"Example: Migrate Catalogs to Avoid CEDA Limitations" on page 180
CPORT and CIMPORT procedures	Transcoding errors can occur if the encodings are not compatible.	"Exporting Multiple Catalogs" in *Base SAS Procedures Guide* "Exporting Individual Catalog Entries" in *Base SAS Procedures Guide* "Applying a Translation Table" in *Base SAS Procedures Guide* "Importing Individual Catalog Entries" in *Base SAS Procedures Guide*
DOWNLOAD and UPLOAD procedures in SAS/CONNECT	Yes	"Transfer Specific Catalog Entry Types" in *SAS/CONNECT User's Guide*

Procedure	Are Catalogs Supported When CEDA Is Invoked?	Examples
		"Long Member Names in Catalog Transfers" in *SAS/CONNECT User's Guide*

Examples: Catalogs

Example: Create and Use a SAS Formats Catalog

Example Code

In this example, the FORMAT procedure creates two user-defined formats. Each format is stored as a catalog entry in a formats catalog.

```
libname mytest v9 '/mydata';
proc format lib=mytest;      /* 1 */
   value $myabc              /* 2 */
         'a'='apple'         /* 3 */
         'b'='banana';
   value myfmt               /* 4 */
         0='success'
         1='error';
run;
```

1. PROC FORMAT specifies the `mytest` libref in the LIB= option, but does not specify a catalog name. The default catalog name `mytest.formats` is used.
2. The VALUE statement creates the `$myabc.` format. The format name begins with a dollar sign ($), so it is a character format. This format assigns the labels `'apple'` and `'banana'`.
3. Notice that the character values are enclosed in single quotation marks (' ').
4. The VALUE statement creates the `$myfmt.` format. The format name does not begin with a dollar sign, so it is a numeric format. This format assigns the labels `'success'` and `'error'`.

The following code shows how to apply user-defined formats that are stored in a formats catalog.

```
data test;                              /* 1 */
   input item $ rc;
```

```
   datalines;
a 0
b 1
d 3
;
run;
options fmtsearch=(mytest);            /* 2 */
proc print data=test noobs;
   format item $myabc. rc myfmt.;      /* 3 */
run;
```

1 The DATA step creates a temporary data set named `test` that contains two variables and three observations.

2 The FMTSEARCH= option specifies the libref `mytest`, where the user-defined formats catalog is stored.

3 In the PRINT procedure, the FORMAT statement assigns the format `$myabc.` to the variable `item` and the format `myfmt.` to the variable `rc`.

In the following output, the formats print the data values differently than they are stored in the data set. Notice that two of the values are not printed differently than they are stored in the data set, because the values are not defined in the formats.

Output 13.1 *PROC PRINT Output That Has Formats Applied*

item	rc
apple	success
banana	error
d	3

Key Ideas

- When you create a permanent SAS format, it is stored in a formats catalog. The catalog entry type is FORMAT for a numeric format or FORMATC for a character format.
- Use the FMTSEARCH= system option to specify the library where the user-defined formats catalog is stored.

See Also

- "FORMAT Procedure" in *Base SAS Procedures Guide*
- "FMTSEARCH= System Option" in *SAS System Options: Reference*

Example: View Information about a Catalog by Using the CATALOG Procedure

Example Code

In this example, the CATALOG procedure with the CONTENTS statement prints information about the `mytest.formats` catalog. This catalog is created in "Example: Create and Use a SAS Formats Catalog" on page 173.

```
libname mytest v9 '/mydata';
proc catalog cat=mytest.formats;
   contents;
run;
quit;
```

Here is the output from the procedure. A description has not been added to either of the formats, so that field is blank.

Output 13.2 *Output from the CONTENTS Statement in PROC CATALOG*

Contents of Catalog MYTEST.FORMATS					
#	Name	Type	Create Date	Modified Date	Description
1	MYFMT	FORMAT	07/16/2019 10:16:21	07/16/2019 10:16:21	
2	MYABC	FORMATC	07/16/2019 10:16:21	07/16/2019 10:16:21	

Key Ideas

- SAS catalog entries are completely identified by a four-level name. In many contexts, you specify the two-level name *libref.catalog-name* only.
- PROC CATALOG is similar to the DATASETS procedure. You use PROC CATALOG to manage catalogs and catalog entries.
- PROC CATALOG is an interactive procedure. You can submit multiple RUN groups within one PROC CATALOG step. A best practice is to terminate the procedure by submitting a QUIT statement.
- Here are other ways to manage catalogs:
  - In the SAS windowing environment, you can manage catalog entries in SAS Explorer or in the CATALOG window.

    Similar catalog directory windows are available in some procedures for SAS/AF, SAS/FSP, and SAS/GRAPH.

- The DICTIONARY.CATALOGS table contains information about all the catalogs that SAS knows about in the current session. You can use PROC SQL to query DICTIONARY.CATALOGS or the Sashelp.Vcatalg view.
- The CEXIST function enables you to verify the existence of a SAS catalog or catalog entry.

See Also

- "CATALOG Procedure" in *Base SAS Procedures Guide*
- "Accessing SAS Information By Using DICTIONARY Tables" in *SAS SQL Procedure User's Guide*
- "CEXIST Function" in *SAS Functions and CALL Routines: Reference*

Example: Determine Whether a Library Contains Catalogs

Example Code

To determine whether a SAS library contains catalogs, use one of the following two methods:

- Use operating environment tools to examine the contents of the physical directories. The file extension is .sas7bcat. (The file extension of Version 6 catalogs is .sc2.)
- In a SAS session, submit the DATASETS procedure with MEMTYPE=CAT. Examine the output for a list of catalogs. If no catalogs exist in the library, the log prints a warning. Here is some example code:

```
libname mytest v9 '/mydata';
proc datasets lib=mytest memtype=cat;
run;
quit;
```

Here is the output from PROC DATASETS, showing that the `mytest` library contains one catalog.

Output 13.3 *PROC DATASETS List of Catalogs*

Directory	
Libref	MYTEST
Engine	V9
Physical Name	/mydata
Filename	/mydata
Owner Name	userid
File Size	4KB
File Size (bytes)	4096

#	Name	Member Type	File Size	Last Modified
1	FORMATS	CATALOG	17KB	08/23/2019 11:54:43

Key Ideas

- To list all the catalogs that exist in a library, use PROC DATASETS with the MEMTYPE=CAT option.
- To view the contents of one catalog, use PROC CATALOG with the CONTENTS statement, as shown in "Example: View Information about a Catalog by Using the CATALOG Procedure" on page 175.

See Also

- "DATASETS Procedure" in *Base SAS Procedures Guide*
- "CATALOG Procedure" in *Base SAS Procedures Guide*

Example: Concatenate Catalogs by Using the CATNAME Statement

Example Code

This CATNAME statement concatenates two SAS formats catalogs:

```
catname testformats (lib1.formats lib2.formats);
```

The `testformats` catalog combines the entries of the `lib1.formats` and `lib2.formats` catalogs.

The catref `testformats` is a shortcut name that can be used to refer to the concatenation of `lib1.formats` and `lib2.formats`. Because a libref is not specified for `testformats`, it is created in the Work library.

The following figure demonstrates the concatenation of catalog entries. Notice that two of the catalog entries, `formats.myfmt` and `formats.myabc`, are in `lib1.formats` and `lib2.formats`. Because `lib1` is listed first in the concatenation, the `lib1` formats are in the `testformats` concatenation.

Output 13.4 *CATNAME Concatenation of Lib1.Formats and Lib2.Formats*

lib1.formats Catalog

Entry Name	Entry Type
MYFMT	FORMAT
MYABC	FORMATC
CRNCY	FORMAT

+

lib2.formats Catalog

Entry Name	Entry Type
MYFMT	FORMAT
MYABC	FORMATC
MYDATES	FORMAT
MYTITLES	FORMATC

=

testformats Catalog

Entry Name	Entry Type
MYFMT	FORMAT
MYABC	FORMATC
CRNCY	FORMAT
MYDATES	FORMAT
MYTITLES	FORMATC

Key Ideas

- A catref is a name that is temporarily associated with a catalog or with concatenated catalogs. You use a CATNAME statement to assign a catref. The concept is similar to a libref.
- In the CATNAME statement, you can include a previously assigned libref when you assign a catref in the form *libref.catref*. However, the catalog concatenation exists logically in memory, so a libref is unnecessary. If you specify *catref* and not *libref.catref*, then the Work library is used. However, you must specify a libref for the catalogs that you are concatenating, unless they exist in the Work library.
- In CATNAME concatenation, only the specified catalogs are combined. In LIBNAME concatenation, any catalogs that have the same name in their respective libraries are concatenated when those libraries are concatenated.
- If a catalog exists that has the name as the catref, then that catalog is not accessible until the catref is deassigned (cleared). To deassign a concatenated catalog, in a CATNAME statement specify the catref name and CLEAR. Specify _ALL_ and CLEAR to deassign all currently assigned catrefs.

See Also

- "Catalog Concatenation" on page 170
- "CATNAME Statement" in *SAS Global Statements: Reference*

Example: Concatenate Catalogs by Using Library Concatenation

Example Code

This LIBNAME statement concatenates two SAS libraries that contain catalogs:

```
libname lib3 (lib1 lib2);
```

The following figure shows the concatenation of catalog entries. The `lib3.formats` catalog is the same as the `testformats` concatenated catalog in "Example: Concatenate Catalogs by Using the CATNAME Statement" on page 177.

In addition, because the LIBNAME concatenates all members of the library, any other catalogs in `lib1` and `lib2` are available by referring to the libref `lib3`.

Output 13.5 *LIBNAME Concatenation of Catalogs*

lib1 + **lib2** = **lib3**

lib1.formats Catalog

Entry Name	Entry Type
MYFMT	FORMAT
MYABC	FORMATC
CRNCY	FORMAT

lib1.sasmacr Catalog

Entry Name	Entry Type
APPLE	MACRO
BANANA	MACRO

lib2.formats Catalog

Entry Name	Entry Type
MYFMT	FORMAT
MYABC	FORMATC
MYDATES	FORMAT
MYTITLES	FORMATC

lib2.mymac Catalog

Entry Name	Entry Type
CREATE	SOURCE
UPDATE	SOURCE

lib3.formats Catalog

Entry Name	Entry Type
MYFMT	FORMAT
MYABC	FORMATC
CRNCY	FORMAT
MYDATES	FORMAT
MYTITLES	FORMATC

lib3.sasmacr Catalog

Entry Name	Entry Type
APPLE	MACRO
BANANA	MACRO

lib3.mymac Catalog

Entry Name	Entry Type
CREATE	SOURCE
UPDATE	SOURCE

Key Ideas

- Library concatenation includes any catalogs implicitly. If multiple catalogs have the same name, their entries are concatenated.

- If you want to concatenate specific catalogs only, then use the CATNAME statement instead of the LIBNAME statement.

See Also

- "Catalog Concatenation" on page 170
- "Example: Concatenate SAS Libraries" in *SAS Programmer's Guide: Essentials*

Example: Migrate Catalogs to Avoid CEDA Limitations

Example Code

In this example, PROC MIGRATE migrates a SAS library that contains catalogs. The target environment uses a different data representation or character encoding than the source library. Because the library contains catalogs, the example uses a SAS/CONNECT spawner to access the source library.

Submit this code in a session that is using the current release of SAS 9.

```
options comamid=tcp;                                  /* 1 */
%let myserver=host.name.com;                          /* 2 */
signon myserver.__1234 user=userid password='mypw';   /* 3 */
libname source '/mydata' server=myserver.__1234;      /* 4 */
libname target v9 '/mylinuxdata';                     /* 5 */
proc migrate in=source out=target;                    /* 6 */
run;
signoff myserver.__1234;                              /* 7 */
```

1. The COMAMID= option specifies to use TCP/IP as the communications access method for connecting to the SAS/CONNECT server.
2. The %LET statement creates the `myserver` macro variable. Use this variable to specify the name of your remote SAS/CONNECT server.
3. The SIGNON statement instructs the spawner to start a session on the SAS/CONNECT server that is named in the `myserver` macro variable. If your port number or service name is defined in the macro variable, then omit it from the SIGNON.
4. This LIBNAME statement assigns the `source` library to the location of the library that is to be migrated. The SERVER= argument specifies the `myserver` macro variable and the port number.
5. This LIBNAME statement assigns the `target` library to an existing location where the migrated library is to be stored.

6 PROC MIGRATE migrates the library members from the `source` library to the `target` library.

7 The SIGNOFF command ends the connection to the server.

Key Ideas

- Catalogs are not supported under cross-environment data access (CEDA) processing. CEDA is invoked when the character encoding or data representation of the SAS file and the SAS session do not match. CEDA processing is common when users upgrade to a new release of SAS that runs in a different operating environment.
- PROC MIGRATE is one way to change the character encoding or data representation of library members to match the current SAS session. PROC MIGRATE is also usually the best way to migrate the members in a SAS library to the current SAS version. PROC MIGRATE retains the data attributes that most users want in a data migration.
- If you are changing to a different data representation or character encoding and your library contains catalogs, then PROC MIGRATE requires a SAS/CONNECT or SAS/SHARE server libref in the IN= argument. (If the SAS/CONNECT or SAS/SHARE server is running in a release earlier than SAS 9.1.3, then use the SLIBREF= option.)
- To avoid CEDA processing, the server's operating environment and encoding must be compatible with the source library.
- If you are not changing to a different data representation or character encoding, then the PROC MIGRATE code is simpler. You can access catalogs and other library members by using a Network File System (NFS), or you can use a SAS/CONNECT or SAS/SHARE server as shown in this example.

See Also

- "Operations That Affect Catalogs" on page 171
- "MIGRATE Procedure" in *Base SAS Procedures Guide*
- "Cross-Environment Data Access" in *SAS Programmer's Guide: Essentials*
- "%LET Macro Statement" in *SAS Macro Language: Reference*
- "COMAMID= System Option" in *SAS/CONNECT User's Guide*
- "SIGNON Statement" in *SAS/CONNECT User's Guide*

Example: Avoid Truncation in Formats When Migrating Catalogs

Example Code

In this example, the user moves a formats catalog from a double-byte character set (DBCS) environment to a UTF-8 environment. A user-defined format contains Japanese characters. In UTF-8, the characters require more bytes to print correctly.

For this example, run the following code in a SAS 9 session with DBCS. If you do not have a DBCS version of SAS, then substitute any other character string for the Japanese characters.

```
libname mytest v9 'c:\examples';
proc format lib=mytest;
   value $waterformat                  /* 1 */
         'a'='湖'    /**/
         'b'='海洋';                    /* 2 */
run;
proc format library=mytest.formats
     cntlout=mytest.outformats         /* 3 */
     (keep=fmtname start label type); /* 4 */
   select $waterformat;                /* 5 */
run;
```

1. The VALUE statement of PROC FORMAT creates the `$waterformat.` format. A catalog name is not specified, so the format is stored in the default catalog, `mytest.formats`.
2. This format assigns two labels. The label `'湖'` is assigned to the value `'a'`. The label `'海洋'` is assigned to the value `'b'`.
3. In the next PROC FORMAT step, the CNTLOUT= option creates the `outformats` data set. This output data set contains information about formats that can be used to rebuild the formats.
4. The KEEP= data set option keeps only the variables `fmtname`, `start`, `label`, and `type`.
5. The SELECT statement selects only the `$waterformat.` format for the purposes of this demonstration. Typically, you would omit this statement, and include all the formats in the output data set.

Open SAS with Unicode Support, or invoke a SAS session with `-encoding utf8`. Alternatively, you can run this code in the same SAS session as in the first step to see how the code works.

```
libname mytest cvp "c:\examples" ;                              /* 1 */
libname mynew "c:\new";
proc format library=mynew.utf8formats cntlin=mytest.outformats; /* 2 */
run;
```

1 Specify the CVP engine in the LIBNAME statement for the location of the `mytest.outformats` data set. The CVP engine expands the length for all the character variables.

2 PROC FORMAT imports the formats from `mytest.outformats` to a new formats catalog, `mynew.utf8formats`. (In this example, only one format is in the `mytest.outformats` data set.) When you rebuild the formats catalog, the catalog entries have the encoding of the target environment.

Run the following code to test the new, expanded `$waterformat.` format:

```
data test;                              /* 1 */
   input item $;
   datalines;
a
b
c
;
run;

options fmtsearch=(mynew.utf8formats); /* 2 */
proc print data=test noobs;
   format item $waterformat. ;          /* 3 */
run;
```

1 The DATA step creates a temporary data set, `test`, in the Work library.

2 The OPTIONS statement submits the FMTSEARCH= option to specify the location of the `mynew.utf8formats` catalog.

3 The FORMAT statement assigns `$waterformat.` to the `item` variable.

Output 13.6 *PROC PRINT Output of Formatted Data Set*

item
湖
海洋
c

Key Ideas

- CEDA does not support catalogs. If you migrate to an incompatible encoding, you must migrate any user-defined formats catalogs to the target session's encoding.
- Be aware of the possibility of truncation when you migrate to an encoding such as UTF-8 that uses more bytes to represent the characters. Truncation might occur if any variable in a formats catalog has a length that does not accommodate the larger character size.
- To expand format lengths, use PROC FORMAT in a two-step process.

  1 Use the CNTLOUT= option with PROC FORMAT to write the formats to an output data set. Use the KEEP= data set option to keep only the necessary

variables. At a minimum, keep the `fmtname`, `start`, and `label` variables. The `type` variable is not necessary if all format names begin with the `@` or `$` prefix.

2. In the target session, use the CNTLIN= option with PROC FORMAT to rebuild the formats catalog from the data set. Assign two LIBNAME statements:
   - Specify the CVP engine in the LIBNAME statement for the data set.
   - Specify a different LIBNAME statement for the new formats catalog. Use a different location, and do not specify the CVP engine.

- If you do not have direct access to the output data set (for example, by using NFS), then you must insert the following intermediate tasks between step 1 and step 2: In the source environment, after step 1, use the CVP engine and a DATA step to expand the character variables in the output data set. Then, in the target environment, before step 2, use a SAS tool (such as PROC MIGRATE or PROC COPY) to migrate or copy the expanded output data set. After these intermediate tasks, continue with step 2. However, you do not need to use the CVP engine, because you already expanded the character variables in the output data set.
- In the CNTLOUT= data set, a best practice is to keep only the variables (`fmtname`, `start`, `label`, and `type`). Keeping only the necessary variables can reduce the possibility of truncation errors when you use the new format in the target environment. The CVP engine does not increase the values of numeric variables such as `default` or `length`. If you include numeric variables in the output data set, you might have to increase the values of those variables in the rebuilt formats catalog.
- For expansion, specify the CVP engine and not the options such as CVPMULT= or CVPBYTES=. The CVP engine determines the appropriate expansion. If you attempt to specify a larger expansion than is needed, then in the second step, PROC FORMAT reduces those lengths to the appropriate lengths. Here is an example: You specify CVPMULT=4, so the engine expands a `label` variable from 4 to 16 bytes. However, you find that in the import step, PROC FORMAT assigns only 6 bytes to the `label`. PROC FORMAT reduces the lengths automatically but cannot expand lengths.
- When truncation is an issue, you must use the two-step PROC FORMAT process for the following reasons.
  - In most cases, PROC MIGRATE is recommended for migrating a SAS library. However, PROC MIGRATE does not support using the CVP engine to expand variable lengths.
  - Normally the CVP engine is used with PROC COPY or the COPY statement of PROC DATASETS. In that usage, the default CVPFORMATWIDTH=YES option expands the length for SAS formats but does not affect user-defined formats.
- If truncation is not an issue, then do not use the CVP engine. You can use this example without CVP, or use one of the following methods:
  - “Example: Migrate Catalogs to Avoid CEDA Limitations” on page 180
  - “Example: Copy a SAS Library by Using a Transport File” on page 241
- Even when the encoding is compatible, you might experience transcoding errors that are not written to the log. For a summary of potential transcoding errors, see “How Character Encoding Affects Compatibility” on page 206.

- To see detailed information about a formats catalog, use the CNTLOUT= option in PROC FORMATS to write the formats to an output data set. Then use PROC CONTENTS or the CONTENTS statement in PROC DATASETS to view the names and lengths of the variables. Use PROC PRINT to view the values in the variables.

See Also

- "FORMAT Procedure" in *Base SAS Procedures Guide*
- "CVPBYTES=, CVPENGINE=, CVPFORMATWIDTH=, CVPMULTIPLIER= LIBNAME Statement Options" on page 28

14

Stored, Compiled DATA Step Programs

Definition of a Stored, Compiled DATA Step Program

A stored, compiled DATA step program is a SAS file that contains a DATA step program that has been compiled and then stored in a SAS library.

Here are some essential concepts about stored, compiled DATA step programs:

- Use a compiled DATA step program for a job that you execute often. The DATA step is already compiled, so you avoid the resources that are used in compilation. The savings are especially significant if the DATA step contains many statements.
- SAS processes the DATA step through the compilation phase and then stores an intermediate code representation of the program and associated data tables. When you execute the stored program, SAS resolves the intermediate code that was produced by the compiler, and generates the executable machine code for your operating environment.

- The SAS member type is PROGRAM. The file extension in most operating environments is .sas7bpgm. See “SAS Files and Member Types” on page 4.

Restrictions and Requirements for Stored, Compiled DATA Step Programs

The following restrictions and requirements apply for using stored, compiled DATA step programs:

- The programs are available for DATA step applications only.
- The program cannot contain global statements. If you include global statements such as FILENAME, FOOTNOTE, LIBNAME, OPTIONS, and TITLE in your source program, SAS stores the compiled program but not the global statements. SAS does not display a warning message in the SAS log.
- You can encrypt and assign a password to the program. To use the DESCRIBE statement in a password-protected DATA step program, you must specify its password. If the program has more than one password, you must specify the most restrictive password.
- SAS does not store raw data in the program.
- To move, copy, rename, or delete the program, use the DATASETS procedure. In the SAS windowing environment, you can also use the SAS Explorer window.
- You cannot move the program to an operating environment that has an incompatible computer architecture. Instead, copy the source code to the new environment in order to recompile and store the program. See “Example: Print the Source Code of a Stored, Compiled DATA Step Program” on page 194. (The default SOURCE=SAVE or SOURCE=ENCRYPT options automatically save your source code in the file when you create the program.)
- You can use the REDIRECT statement to specify different input or output data sets than are specified in the program. See “Example: Redirect Input or Output in a Stored, Compiled DATA Step Program” on page 193. The number and attributes of variables in the input SAS data sets that you read with the REDIRECT statement must match those of the input data sets in the SET, MERGE, MODIFY, or UPDATE statements of the source code. If they do not match, the following occurs:
  - If the variable length attributes differ, the length of the variable in the source code data set determines the length of the variable in the redirected data set.
  - If extra variables are present in the redirected data sets, the stored program stops processing, and an error message is sent to the SAS log.
  - If the variable type attributes are different, the program stops processing, and an error message is sent to the SAS log.

Comparing DATA Step Views and Stored, Compiled DATA Step Programs

DATA step views differ from stored, compiled DATA step programs in the following ways:

- A DATA step view is implicitly executed when it is referenced as an input data set by another DATA or PROC step. Its main purpose is to provide data, one record at a time, to the invoking procedure or DATA step.
- A stored, compiled DATA step program is explicitly executed when it is specified by the PGM= option in a DATA statement. Its purpose is usually a more specific task, such as creating SAS data sets or originating a report. The DATA step program is compiled when it is stored, which reduces processing when it is executed.
- You can use the REDIRECT statement when you execute a stored, compiled DATA step. You cannot use this statement with DATA step views.

See also:

- Chapter 6, “SAS Views,” on page 49
- “Storing and Reusing Macros” in *SAS Macro Language: Reference*

Examples: Stored, Compiled DATA Step Programs

Example: Create a Stored, Compiled DATA Step Program

Example Code

To run this example, first create the following data set, which is named `mydata.plantsinventory`. The data set contains plant names, plant codes, and the number of the plants in the inventory.

```
libname mydata v9 'c:\examples';
```

```
data mydata.plantsinventory;
   input plantname $ code inventory;
   datalines;
lily 1 11
dahlia 2 15
cyclamen 6 0
;
run;
proc print data=mydata.plantsinventory noobs;
run;
```

Here is the PROC PRINT output of the `mydata.plantsinventory` data set.

Output 14.1 *PROC PRINT of mydata.plantsinventory*

plantname	code	inventory
lily	1	11
dahlia	2	15
cyclamen	6	0

The following DATA step creates a data set that is named `mydata.nextorder` and a stored, compiled DATA step program that is named `myprogs.createorder`. The DATA step does not execute, so the `mydata.nextorder` data set is not created yet.

```
libname myprogs v9 'c:\myfiles';
data mydata.nextorder / pgm=myprogs.createorder; /* 1 */
   set mydata.plantsinventory;                   /* 2 */
   if code = 1 then                              /* 3 */
      do;
         type='perennial';
         order=30-inventory;
      end;
   else
   if code = 2 then
      do;
         type='annual';
         order=50-inventory;
      end;
   else
      do;
         type='ERROR';
         order=0;
      end;
run;
```

1 The DATA step creates the `myprogs.createorder` data set. The PGM= option specifies to create a stored, compiled DATA step program that is named `myprogs.createorder`.

2 The SET statement specifies to use `mydata.plantsinventory` as the input data set.

3 The IF-THEN/ELSE logic and DO loops use the `code` variable to assign a `type` and calculate `order`, which is the number of plants to order.

Key Ideas

- The ability to compile and store a DATA step program enables you to retrieve and execute the program. Stored, compiled DATA step programs can reduce processing costs by eliminating the need to compile DATA step programs repeatedly.
- To create a stored, compiled DATA step program, specify a name for the output SAS data set, followed by a slash (/) and the PGM= option in a DATA statement. In the PGM= option, specify a valid SAS name for the SAS file to contain the stored program.
- A stored, compiled DATA step program cannot contain global statements. If you include global statements such as FILENAME, FOOTNOTE, LIBNAME, OPTIONS, and TITLE in your program, SAS stores the compiled program but not the global statements. SAS does not display a warning message in the SAS log.

See Also

- PGM=
- "Restrictions and Requirements for Stored, Compiled DATA Step Programs" on page 188

Example: Execute a Stored, Compiled DATA Step Program

Example Code

This DATA step executes the stored, compiled DATA step program `myprogs.createorder` that is created in "Example: Create a Stored, Compiled DATA Step Program" on page 189. In that example, the DATA step does not execute. In this example, the DATA step executes and creates the `mydata.nextorder` data set.

```
libname mydata v9 'c:\examples'; /* 1 */
libname myprogs v9 'c:\myfiles';
data pgm=myprogs.createorder;    /* 2 */
   execute;                      /* 3 */
run;
proc print data=mydata.nextorder noobs;
run;
```

1. The two LIBNAME statements assign the `mydata` library, which contains data, and the `myprogs` library, which contains SAS programs.
2. The PGM= option specifies the stored, compiled DATA step program `myprogs.createorder`.
3. The EXECUTE statement executes the program.

Here is the PROC PRINT output of the `mydata.nextorder` data set.

Output 14.2 *PROC PRINT of mydata.nextorder*

plantname	code	inventory	type	order
lily	1	11	perennial	19
dahlia	2	15	annual	35
cyclamen	6	0	ERROR	0

Key Ideas

- To execute a stored, compiled DATA step program, specify the PGM= option and the program name in a DATA statement. Do not include a slash (/). You do not have to specify the input or output data set names.
- As a best practice, include the EXECUTE statement when you want to execute the program. In some contexts the EXECUTE statement is not necessary, but it is a good reminder in your code that you are executing the program.
- Always include the RUN statement or other step boundary when you create or execute a stored, compiled DATA step program.
- Because a stored, compiled DATA step program cannot contain global statements, you must specify statements such as FILENAME and LIBNAME when you execute the program.

See Also

- PGM=
- "EXECUTE Statement" in *SAS DATA Step Statements: Reference*
- "Restrictions and Requirements for Stored, Compiled DATA Step Programs" on page 188

Example: Redirect Input or Output in a Stored, Compiled DATA Step Program

Example Code

To run this example, first create the following data set named `mydata.q4inventory`, which is similar to `mydata.plantsinventory` in "Example: Create a Stored, Compiled DATA Step Program" on page 189.

```
libname mydata v9 'c:\examples';
data mydata.q4inventory;
   input plantname $ code inventory;
   datalines;
lily 1 5
dahlia 2 1
cyclamen 1 0
;
run;
```

In the following DATA step, the REDIRECT statements specify different input and output data sets than the ones that are stored in the program.

```
libname mydata v9 'c:\examples';                          /* 1 */
libname myprogs v9 'c:\myfiles';
data pgm=myprogs.createorder;                             /* 2 */
   redirect input mydata.plantsinventory=mydata.q4inventory; /* 3 */
   redirect output mydata.nextorder=mydata.q4order;       /* 4 */
   execute;                                               /* 5 */
run;
proc print data=mydata.q4order noobs;
run;
```

1. The two LIBNAME statements assign the `mydata` library, which contains data, and the `myprogs` library, which contains SAS programs.
2. The PGM= option specifies the stored, compiled DATA step program `myprogs.createorder`.
3. This REDIRECT statement with the INPUT argument specifies to use `mydata.q4inventory` instead of `mydata.plantsinventory` for input data.
4. This REDIRECT statement with the OUTPUT argument specifies to use `mydata.q4order` instead of `mydata.nextorder` for output data.
5. The EXECUTE statement executes the program.

Here is the PROC PRINT output of the `mydata.q4order` data set:

Output 14.3 *PROC PRINT of mydata.q4order*

plantname	code	inventory	type	order
lily	1	5	perennial	25
dahlia	2	1	annual	49
cyclamen	1	0	perennial	30

Key Ideas

- You can use the REDIRECT statement to specify different input or output data sets than are specified in the program. The number and attributes of variables in the input data sets must match those of the input data sets in the SET, MERGE, MODIFY, or UPDATE statements of the source code.
- To redirect input and output in external files, submit a FILENAME statement so that you can refer to the fileref in the REDIRECT statement.
- Because a stored, compiled DATA step program cannot contain global statements, you must specify statements such as FILENAME and LIBNAME when you execute the program.

See Also

- "REDIRECT Statement" in *SAS DATA Step Statements: Reference*
- "EXECUTE Statement" in *SAS DATA Step Statements: Reference*
- PGM=
- "Restrictions and Requirements for Stored, Compiled DATA Step Programs" on page 188

Example: Print the Source Code of a Stored, Compiled DATA Step Program

Example Code

In this DATA step, the DESCRIBE statement writes the source code to the SAS log from the stored, compiled DATA step program `myprogs.createorder`. The program is created in "Example: Create a Stored, Compiled DATA Step Program" on page 189.

```
libname mydata v9 'c:\examples';
```

```
libname myprogs v9 'c:\myfiles';
data pgm=myprogs.createorder;
   describe;
run;
```

The SAS log shows the source code that was used to create the program. If you need to re-create the stored, compiled DATA step program in a different operating environment, you can copy this DATA step code.

Example Code 14.1 *Partial SAS Log Showing the Source Code Generated by the DESCRIBE Statement*

```
NOTE: DATA step stored program MYPROGS.CREATEORDER is defined as:

data mydata.nextorder / pgm=myprogs.createorder;
   set mydata.plantsinventory;
   if code = 1 then
      do;
         type='perennial';
         order=30-inventory;
      end;
   else
      if code = 2 then
         do;
            type='annual';
            order=50-inventory;
         end;
      else
         do;
            type='ERROR';
            order=0;
         end;
run;
```

Key Ideas

- Use the DESCRIBE statement without the EXECUTE statement to retrieve source code from a stored, compiled DATA step program.

  Use the DESCRIBE statement with the EXECUTE statement to retrieve source code and also execute a stored, compiled DATA step program.
- Retrieving the source code can be helpful if you need to re-create a stored, compiled DATA step program in an operating environment that has an incompatible computer architecture. Copy the source code to the new environment to recompile and store the program.

See Also

- "DESCRIBE Statement" in *SAS DATA Step Statements: Reference*

15

Extended Attributes

Definition of Extended Attributes

Extended attributes are customized metadata that you can define for a SAS data set or variable. SAS data sets automatically store many attributes, such as the name, member type, engine, and variable labels. Extended attributes are additional metadata that you add for your own use.

Here are some essential concepts:

- To create or control extended attributes on page 198, use the XATTR statements in the DATASETS procedure. The XATTR statements must appear under a MODIFY statement.
- Extended attributes are stored as name-value pairs. Extended attributes data is stored in a separate file that has the same name as the data set. The member type is EXTATTR. The file extension in most operating environments is .sas7bxat. See “SAS Files and Member Types” on page 4.
- To view information about extended attributes on page 199, use the CONTENTS procedure or the CONTENTS statement of the DATASETS procedure. If the .sas7bxat file is absent from the directory, an error is written to the log.
- An extended attribute can have numeric or character values.
- There is no limit to the number of extended attributes that you can define for a data set or for a variable.
- There is no maximum length to an extended attribute character value. By default, each value is stored in 256-byte segments. The length of the segment can be changed with the SEGLEN= option of the XATTR OPTIONS statement. This option indicates the length of the storage element that will hold the character attribute value. If the segment size is not large enough for a character attribute

value, another segment is allocated. To minimize processing time, choose a length that would accommodate most attribute values for the data set.

- Extended attributes are not supported for data views.

Some operations can affect extended attributes:

- Extended attributes are a feature for the V9 engine only. Under engines that do not support extended attributes, you can copy a data set with the DATA step, the COPY procedure, or the COPY statement of the DATASETS procedure. However, extended attributes are dropped and a warning is written to the log.
- Here is the behavior when you use the APPEND procedure or the APPEND statement of the DATASETS procedure:
  - If the BASE= data set exists, then the extended attributes of the BASE= data set are retained. The extended attributes of the DATA= data set are not appended.
  - If the BASE= data set does not exist, then the extended attributes of the DATA= data set are retained in the new data set.

Examples: Extended Attributes

Example: Add Extended Attributes

Example Code

This example adds extended attributes to a data set and its variables.

```
libname myfiles v9 '/mydata';
proc datasets lib=sashelp nolist;                          /* 1 */
   copy out=myfiles;
      select cars;
quit;
proc datasets lib=myfiles nolist;
   modify cars;
      xattr add ds ref="copied from sashelp";              /* 2 */
      xattr add var invoice (role="admin" age="quarterly")
                    model (role="all");                    /* 3 */
quit;
```

1. To run this example, assign the `myfiles` libref and use the DATASETS procedure to copy the `cars` data set from the `sashelp` library.
2. In this PROC DATASETS step, the XATTR ADD statement with DS option adds the extended attribute `ref` to the `myfiles.cars` data set.

3 The XATTR ADD statement with the VAR option adds extended attributes to the `invoice` and `model` variables.

Key Ideas

- Extended attributes are customized metadata that you can create for SAS data sets and variables.
- To create and control extended attributes, use XATTR statements in the DATASETS procedure.
- The XATTR statements must appear in a PROC DATASETS MODIFY RUN group, which means that the XATTR statements are a subgroup of statements under the MODIFY statement.

See Also

- "XATTR ADD Statement" in *Base SAS Procedures Guide*
- "XATTR DELETE Statement" in *Base SAS Procedures Guide*
- "XATTR OPTIONS Statement" in *Base SAS Procedures Guide*
- "XATTR REMOVE Statement" in *Base SAS Procedures Guide*
- "XATTR SET Statement" in *Base SAS Procedures Guide*
- "XATTR UPDATE Statement" in *Base SAS Procedures Guide*

Example: View Information about Extended Attributes

Example Code

The following PROC DATASETS lists information about the `myfiles` library and the `cars` data set.

```
libname myfiles v9 '/mydata';
proc datasets lib=myfiles;
   contents data=cars;
quit;
```

The directory portion of the PROC DATASETS output shows the members in the `myfiles` library.

Output 15.1 *Portion of PROC CONTENTS Output Showing Directory Information*

Directory	
Libref	MYFILES
Engine	V9
Physical Name	/mydata
Filename	/mydata
Inode Number	83863546
Access Permission	rwxrwxrwx
Owner Name	userid
File Size	4KB
File Size (bytes)	4096

#	Name	Member Type	File Size	Last Modified
1	CARS	DATA	192KB	08/05/2019 15:00:25
	CARS	EXTATTR	40KB	08/05/2019 15:00:25

In the following portion of the PROC DATASETS output, notice the EXTATTR segment length.

Output 15.2 *Portion of PROC CONTENTS Output Showing EXTATTR Segment Length*

Data Set Name	MYFILES.CARS	Observations	428
Member Type	DATA	Variables	15
Engine	V9	Indexes	0
Created	08/05/2019 15:00:24	Observation Length	152
Last Modified	08/05/2019 15:00:25	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	YES
Label	2004 Car Data	EXTATTR Segment Length	256
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	us-ascii ASCII (ANSI)		

The following portion of the PROC DATASETS output lists the extended attributes. The `cars` data set has one extended attribute. The variable `invoice` has two extended attributes. The variable `model` has one extended attribute.

Output 15.3 *Portion of PROC CONTENTS Output Showing List of Extended Attributes*

Alphabetic List of Data Set Extended Attributes		
Extended Attribute	Numeric Value	Character Value
ref	.	copied from sashelp

Alphabetic List of Extended Attributes on Variables			
Extended Attribute	Attribute Variable	Numeric Value	Character Value
age	invoice	.	quarterly
role	invoice	.	admin
role	model	.	all

Key Ideas

- Extended attributes data is stored in a separate file that has the same name as the data set. The member type is EXTATTR.
- You can use the CONTENTS procedure or the CONTENTS statement of the DATASETS procedure to view information about extended attributes.
- The CONTENTS output includes the setting of the extended attributes segment length. You can change the length by using the SEGLEN= option of the XATTR OPTIONS statement.

See Also

- "CONTENTS Statement" in *Base SAS Procedures Guide*
- "XATTR OPTIONS SEGLEN=number-of-bytes" in *Base SAS Procedures Guide*

16

Compatibility and Migration

Definitions for Compatibility and Migration

Here are some essential concepts for compatibility and migration:

Cross-Environment Data Access (CEDA)
: enables a SAS file that was created in a directory-based operating environment (for example, UNIX or Windows) to be processed in an incompatible environment or under an incompatible session encoding. With CEDA, the processing is automatic and transparent. You do not need to create a transport file, use SAS procedures that convert the file, or change your SAS program. CEDA supports files that were created with SAS 7 and later releases. CEDA is a Base SAS feature.

data representation
: is the form in which data is stored in a particular operating environment. Different operating environments use different standards or conventions for storing data. (See "Compatible Data Representations" in *SAS Programmer's Guide: Essentials*.)

- Floating-point numbers can be represented in IEEE floating-point format or IBM floating-point format.
- Data alignment can be on a 1-byte, 4-byte, or 8-byte boundary, depending on data type requirements for the operating environment.
- Data type lengths can be 8 bits or more for a character data type, 16 bit, 32 bit, or 64 bit for an integer data type, 32 bit for a single-precision floating-point data type, and 64 bit for a double-precision floating-point data type.
- The ordering of bytes can be big Endian or little Endian.

encoding
: is a set of characters (letters, logograms, digits, punctuation, symbols, control characters, and so on) that have been mapped to numeric values (called code points) that can be used by computers. The code points are assigned to the characters in the character set by applying an encoding method. Some examples of encodings are Wlatin1 and Danish EBCDIC. (See "Encoding Combinations That Do Not Need CEDA Processing for Transcoding" in *SAS National Language Support (NLS): Reference Guide*.)

incompatible
: describes a file that has a different data representation or encoding than the current SAS session. CEDA enables access to many types of incompatible files.

source
: is the deployment before you upgrade to a new version, or the SAS session where data was created, or the SAS library that you want to migrate or copy.

target
: is the new deployment, or session, or library.

migrate a SAS library
: involves copying the members of a SAS library from source to target and also changing the file format, data representation, and character encoding if necessary. After migration, files are entirely compatible with the target SAS session.

move or copy a SAS library
: does not change the file format, data representation, or character encoding. You might encounter compatibility problems.

Cross-Release and Cross-Environment Compatibility

Avoiding Compatibility Issues

How the SAS Version Affects Compatibility

The SAS version does not usually affect compatibility. SAS files that are created in a previous version are generally compatible in the current version of SAS®9.

However, when you upgrade to a new SAS version, the data representation or encoding could change.

How Data Representation Affects Compatibility

SAS assigns a data representation attribute to data sets and to many other SAS files. Processing an existing SAS file in a different operating environment could affect compatibility.

Some operating environments store data very similarly, so they are compatible. When environments are not compatible, SAS uses cross-environment data access (CEDA) to process the data. CEDA processing is automatic and transparent, but you must be aware of the restrictions. See "Restrictions for CEDA" in *SAS Programmer's Guide: Essentials*.

Here are some resources to learn whether the source and target environments are compatible:

- Consult the table in the following topic: "Compatible Data Representations" in *SAS Programmer's Guide: Essentials*. If your source and target operating environments are in the same group, then CEDA is not used. Read the table's notes for important caveats.

- Try to process some source data in the target environment, and check the SAS log for a CEDA message.

How Character Encoding Affects Compatibility

SAS assigns an encoding attribute to data sets and to many other SAS files. When the encoding of the file is not compatible with the session encoding, SAS uses CEDA to transcode the data. CEDA processing is automatic and transparent, but you must be aware of the restrictions. See "Restrictions for CEDA" in *SAS Programmer's Guide: Essentials*.

For many customers who change operating environments, the encoding is compatible. However, if the data representation is not compatible, SAS still uses CEDA.

If the data contains 7-bit ASCII characters (U.S. English) only, then the data is compatible in any other ASCII session, including UTF-8. Other ASCII encodings use the high-order bit for different national characters.

Transcoding is usually an automatic process that remaps characters to the appropriate code points in a different encoding. Here are some potential transcoding issues:

- Truncation is one of the most common transcoding errors. When SAS transcodes character data to an encoding that uses more bytes to represent the characters, truncation can occur if the column length does not accommodate the larger character size. For example, a character might be represented in Wlatin1 encoding as one byte but in UTF-8 as two bytes. Truncation shortens a character string but can also cause "garbage" characters or incorrect characters in output.
- Encodings can use different code points for a character, or a character might not be present in an encoding.
- Microsoft smart quotation marks or hyphens can cause truncation errors. They are not ASCII characters, and they require more than one byte when converted to UTF-8.

Here are some resources to learn whether the source and target encodings are compatible:

- Consult the following: "Encoding Combinations That Do Not Need CEDA Processing for Transcoding" in *SAS National Language Support (NLS): Reference Guide*.
- Try to process some source data in the target environment, and check the SAS log for a CEDA message or a transcoding error:
  - "Example: Understand Log Messages about CEDA" on page 218
  - "Example: Understand Log Messages about Truncation" on page 219

Examples to Avoid Compatibility Issues

- "Examples: CEDA" on page 218
- "Examples: Migrate SAS Libraries" on page 229

Compatibility in Previous Releases

File Features That Are Not Supported in a Previous Release

SAS®9 files are generally compatible with earlier releases. However, new file features can cause errors, such as in the following use cases:

- You upgrade to a new release of SAS, but you cannot modify legacy data sets. You want to be aware of any SAS features that are not supported by the earlier file format.
- You exchange data with a SAS user whose system has not been upgraded. You cannot use file features that would make a file unusable in the earlier version of SAS.

A few SAS file features can result in the following error message when the data is used under an earlier SAS release. These features are explained in the following table.

```
ERROR: File MYLIB.TABLENAME.DATA not compatible with this SAS version.
```

Table 16.1 *Incompatible File Features in a Previous Release*

Feature	Issue	Compatibility Options
PROC SQL view that contains a USING clause	In SAS 9.4M6 and later releases, if you use the V9 engine to create a PROC SQL view that contains a USING clause, the view is not accessible in SAS 9.4M5 or earlier releases. A USING clause stores DBMS connection information in the view by embedding the SAS/ACCESS LIBNAME statement inside the view.	Avoid the USING clause for PROC SQL views.

Feature	Issue	Compatibility Options
AES and AES2 encryption	If you specify ENCRYPT=AES2 to encrypt a data set, the data set cannot be accessed by any release prior to SAS 9.4M5. If you specify ENCRYPT=AES to encrypt a data set, the data set cannot be accessed by any release prior to SAS 9.4.	Set the data set option ENCRYPT=YES or ENCRYPT=NO.
Extended attributes	If you add extended attributes to a data set or to a variable in a data set, the data set cannot be accessed by any release prior to SAS 9.4. See Chapter 15, "Extended Attributes," on page 197.	Avoid extended attributes.
Metadata-bound libraries	Metadata-bound data sets cannot be accessed by any release prior to SAS 9.3M2. See *SAS Guide to Metadata-Bound Libraries*.	Avoid metadata-bound libraries.
Data sets that have EXTENDOBSCOUNTER=YES, or greater than 2 billion observations	In SAS 9.4, extended observation count is the default. In SAS 9.3, it is optional. If a data set has EXTENDOBSCOUNTER=YES, then it is not accessible in SAS 9.2 and earlier releases. If a data set has more than 2 billion observations, it is unusable in SAS 9.2 and earlier releases in 32-bit SAS. The behavior depends on your operating environment and SAS release. See "Backward Compatibility of the Extended Observation Count Attribute" on page 45.	Set the system option, LIBNAME statement option, or data set option EXTENDOBSCOUNTER=NO.
Extended naming for data sets, views, and item stores	SAS 9.3 and later releases support extended naming rules for data sets, views, and item stores. See "Data Set Names" in *SAS Programmer's Guide: Essentials*.	Set the system option VALIDMEMNAME=COMPATIBLE. Messages are printed in the log when names cannot be transcoded.
CHECK integrity constraints	If you add the constraint to an existing SAS data set or create a SAS data set that includes the constraint, the data set cannot	Avoid CHECK integrity constraints.

Feature	Issue	Compatibility Options
	be used in any release prior to SAS 9.2.	
Encoded passwords from the PWENCODE procedure	If you use PROC PWENCODE to encode a password, and you protect a data set with that password, then the data set might not be accessible by prior releases. For example, the SAS005 method is added in SAS 9.4M5. A data set that is protected with a SAS005-encoded password cannot be opened in any release prior to SAS 9.4M5. Protecting a data set with an encoded password is not supported prior to SAS 9.2.	Use an encoding method that is supported in the target release, or avoid PROC PWENCODE.
Linguistic collation with the SORT procedure	When you use SORTSEQ=LINGUISTIC with PROC SORT in SAS 9.2 and later, metadata is stored in the data set header to indicate that the data set was sorted. Earlier releases do not support linguistic collation. If you use the sorted data set in earlier releases, then the data set is not shown to be sorted. In addition, the COMPARE procedure might show differences due to the sort indicator. The data set can be accessed.	If you use the SORTSEQ= system option, specify a translation table instead of LINGUISTIC.
Greater than 32,767 variables	A data set with greater than 32,767 variables is unusable in SAS 9.0 and earlier releases. See "Variable Attributes" in *SAS Programmer's Guide: Essentials*.	Avoid creating more than 32,767 variables in a data set.
Suppressed transcoding of a specified variable	Setting TRANSCODE=NO in the ATTRIB statement in a DATA step causes a data set to be unusable in SAS 9.0 and earlier releases. See *SAS National Language Support (NLS): Reference Guide*.	Avoid TRANSCODE=NO.
Format or informat names longer than 8 bytes	A data set that uses format or informat names longer than 8 bytes is unusable in SAS 8 and earlier releases.	Set the system option VALIDFMTNAME=FAIL or VALIDFMTNAME=WARN.

Feature	Issue	Compatibility Options
Encoding attribute	SAS 7 and 8 data sets must be updated or output in a SAS®9 session to be stamped with an encoding attribute. (The encoding attribute was supported prior to SAS®9 in China, Korea, and Japan.) The encoding attribute is not supported in SAS 6.	If you replace or update a data set that does not have an encoding attribute, then be aware that the session encoding is stamped on the data set by default. If that behavior is not desired, you can override the data set's encoding by using the data set option ENCODING= or the LIBNAME options INENCODING= or OUTENCODING=. If a session encoding is stamped on a data set incorrectly, and you are certain of the correct encoding, then you can set it with the CORRECTENCODING= option in the MODIFY statement of the DATASETS procedure.
Extended naming conventions for variables	The rules for SAS variable names are improved in SAS 8 and later. Some current conventions are not supported for SAS 7 and earlier releases. Variable names longer than 8 bytes are not supported in SAS 6. "Variable Names" in *SAS Programmer's Guide: Essentials*	Set the system option VALIDVARNAME=V7 or VALIDVARNAME=V6 to get an error message. Set the system option V6CREATEUPDATE= to control and monitor attempts to create or update a SAS 6 data set.

Regressing a File to a Previous Release

Regression to a previous release is usually unnecessary if you avoidfile features that are not supported. If you need to regress a SAS®9 file to an earlier file format, the following table lists your choices and some limitations.

Table 16.2 *Methods of Regressing SAS Files to an Earlier Release*

Method	Limitations for a SAS 8 or SAS 7 Target	Limitations for a SAS 6 Target
DATA step	Must be run in the SAS 7 or 8 session.	Specify the V6 engine in the target LIBNAME statement.

Method	Limitations for a SAS 8 or SAS 7 Target	Limitations for a SAS 6 Target
		SAS®9 and SAS 6 must be on the same type of operating environment.
PROC COPY (or COPY statement of PROC DATASETS) with the NOCLONE option	Must be run in the SAS 7 or 8 session.	Specify the V6 engine in the target LIBNAME statement. Set VALIDVARNAME=V6 before the PROC COPY. SAS®9 and SAS 6 must be on the same type of operating environment.
XPORT engine	Truncates variable names to 8 bytes.	Truncates variable names to 8 bytes.
OUTREP= data set option or LIBNAME statement option ENCODING= data set option or OUTENCODING= LIBNAME statement option	Not a true regression, but changes the data representation or encoding to avoid future CEDA processing.	Not a true regression, but changes the data representation or encoding to enable V6 engine compatibility.

CAUTION

Do not use PROC CPORT and PROC CIMPORT for regressing. This practice causes numerous errors when you import the transport file. Common errors include the following:

```
ERROR: TRANSPORT FILE IS BAD
ERROR: The value is not a valid SAS name
```

Coexistence of Multiple SAS Releases

It is usually possible to run multiple SAS releases simultaneously on the same machine. This topic lists known issues.

Sharing data sets and other SAS files:

- Because you are running multiple releases on the same operating environment, the SAS files are probably compatible. If you are uncertain, look in the SAS log for a CEDA message.
- If you need to share files among releases, do not use any file features that are not supported in the earlier releases.

- It is recommended that you do not store files from multiple releases in the same library. You could accidentally overwrite a file with the same name that was created under another release.

Sharing SAS code:

- For the most common issues, see "Modifying SAS Programs for a New SAS Release" on page 216.
- Syntax is usually compatible across releases. Occasionally, behavior can change in later releases, or new syntax is not supported in earlier releases. See the SAS Documentation What's New site.

Installation and configuration:

- If you are installing two or more releases, install the previous release first. Otherwise, you might run into problems with the shared files directory.
- Install multiple releases in separate locations on the machine.
- Make sure that the multiple releases use different TCP/IP port numbers.
- It is recommended that you use the SAS configuration file to separate the locations of SASWORK.
- You can add new orders within one release to a single SAS Software Depot, but do not mix multiple releases in one Software Depot. For example, do not add a SAS 9.4 order to a SAS 9.3 Software Depot.
- If you are migrating a metadata-based deployment to a new release, use the same installer account and administrative accounts for both deployments in order to avoid permission issues.
- Running multiple versions of SAS on a machine might require additional disk space or CPU time.

Execution:

- You can run multiple versions simultaneously in batch.
- You can run multiple versions simultaneously in interactive sessions from the SAS windowing environment.
- If a product or component is a web-based application or it produces output for a web browser, use a unique URL for each version.
- The `SAS` command can be set to one or the other release of SAS by the operating environment. The command can be set via search paths, registry settings, or operating environment conventions. When invoking SAS, ensure that you are getting the release of SAS that you want.
- SAS environment variables can differ between SAS releases. Failure to set the environment can result in errors, including incorrect user-ID authentication with the SAS Metadata Server. A recommended solution is to write a script that sets the environment variables when SAS is invoked. Here is a sample environment-setting program for UNIX:

```
PATH=/usr/local/bin:/usr/bin:/etc:/usr/sbin:/usr/ucb:/usr/bin/X11:/sbin:$HOME:.
export SASROOT=/usr/local/sas/sas913
export PATH=$PATH:/usr/local/sas:$SASROOT:$HOME/bin
```

Default Engine for a Mixed Library

The SAS®9 file format is very similar to that of SAS 8 and SAS 7. Therefore, the file extensions have not changed. A data set's file extension is .sas7bdat in most operating environments.

SAS 6 files and other engines' files differ significantly from SAS®9 files, so their file extensions are different from SAS®9. SAS 6 file extensions vary, depending on the operating environment.

For more information about library members and file extensions, see "Definitions for SAS Libraries" in *SAS Programmer's Guide: Essentials*.

The shipped default Base SAS engine is BASE. In SAS®9 and SAS® Viya®, the BASE engine is an alias for the V9 engine. If you do not specify an engine name when you create a new library, and if you have not specified the ENGINE system option, then the V9 engine is automatically selected. If the library location already contains SAS files, then SAS might be able to assign the correct engine based on those files. For example, if the location contains V9 data sets only, then SAS assigns the V9 engine. However, if a library location contains a mix of different engine files, then SAS might not assign the engine you want. Therefore, specifying the engine is a best practice. For details about engine assignment, see "SAS Engines" in *SAS Programmer's Guide: Essentials*.

Here is a summary of the default behavior if you do not specify an engine in a library assignment:

Table 16.3 *Default Library Engine Assignment in SAS®9*

SAS Library Contents	Default Engine Assignment
No SAS files; the library is empty	The default engine, which is specified in the ENGINE= system option. The default is V9 if the ENGINE= system option is not specified.
Only SAS®9 files	V9
Only SAS 8 files	V9
Only SAS 7 files	V9
Only SAS 6 files	V6
A mixed library of SAS®9, SAS 8, SAS 7, and SAS 6 files	V9
A mixed library of SAS®9, SAS 8, SAS 7, or SAS 6 files with other engine files	Behavior varies depending on the library contents and whether the ENGINE= system

SAS Library Contents	Default Engine Assignment
	option is set. Specify the appropriate engine for the files that you want to access.

V6 Compatibility Engine

If you cannot migrate SAS 6 files, the V6 compatibility engine can automatically process some files in a SAS®9 session.

The V6 engine supports the following processing if the data representation is compatible. If these SAS 6 files were created under 32-bit Windows, they can be processed in SAS 9.4 on 32-bit Windows. If they were created under z/OS, they can be process in SAS 9.4 on z/OS.

- SAS data set supports input, update, and output.
- DATA step view and PROC SQL view support input.
- SAS/ACCESS view supports input and update.
- SAS catalog supports input.
- Stored, compiled DATA step program supports input.
- MDDB file supports input.
- The DELETE statement is supported with PROC DATASETS in SAS 9.2 and later.

In SAS 9.4 on 64-bit Windows, the V6 engine provides the following processing for files that were created under 32-bit Windows.

- SAS data set supports input (read-only).
- No SAS views, catalogs, compiled programs, or MDDBs are supported.
- Indexes are not supported.
- The DELETE statement is supported with PROC DATASETS in SAS 9.2 and later.

The V6 engine processes SAS 6 files created by Release 6.08 through Release 6.12. For SAS 6 files prior to Release 6.08, see the SAS documentation for your operating environment.

Migrating SAS Files to a New Release

Migrating SAS Libraries to a New SAS®9 Release

SAS Files That Can and Cannot Be Migrated

The MIGRATE procedure is a one-step copy procedure that retains the data attributes that most users want in a data migration. See "Examples: Migrate SAS Libraries" on page 229.

Table 16.4 Recommended Migration Methods

File Type	Best Practices
SAS data set	Use PROC MIGRATE.
PROC SQL view DATA step view SAS/ACCESS view descriptor for Oracle, SAP, or DB2	Use PROC MIGRATE. For SAS/ACCESS view descriptors, PROC MIGRATE automatically uses the CV2VIEW procedure, which converts the view descriptors to SQL views. Migrating SAS/ACCESS views to a different operating environment is not supported.
SAS catalog	Use PROC MIGRATE. If the processing invokes CEDA on the target session, then a SAS/CONNECT or SAS/SHARE server is required. If you do not have a SAS/CONNECT or SAS/SHARE server, then use the CPORT and CIMPORT procedures.
item store	PROC MIGRATE is usually not a best practice or is not supported. ■ ODS template item store: Use the SOURCE statement in the TEMPLATE procedure to save source code to a file. Then, rebuild the item store in the target session. ■ ODS documents item store: These are usually compatible from previous releases if you do not change to a different operating environment. ■ Linear model item store: If the procedure stored the SAS code that created the item store, use the PLM procedure with the SHOW PROGRAM statement to show the SAS code. Write the output to the ODS LISTING destination.

File Type	Best Practices
	Open the output file, copy the SAS code, and paste the code into a session in the target environment. You also need the original data. ■ SAS registry file: Do not copy or migrate. If desired, you can examine the source environment's settings and determine whether they should be applied to the target. See "The SAS Registry" in *SAS Programmer's Guide: Essentials*.
stored, compiled DATA step program stored, compiled macro	Move the source code to the target environment where you can compile and store it.
SAS program	Move the source code to the target environment. SAS code that was written in a previous release is usually compatible in the current release. See "Modifying SAS Programs for a New SAS Release" on page 216.
SAS autoexec and configuration files	Do not copy or migrate. If desired, you can open configuration and autoexec files in a text editor to examine each option and statement. If the customizations are appropriate for the target environment, then make changes in the target configuration and autoexec files. For details, see the SAS documentation for your operating environment.
SASHELP, SASUSER, and WORK libraries	Do not copy or migrate. See "SAS Default Libraries" in *SAS Programmer's Guide: Essentials*.

Avoiding Truncation When Migrating

See the following examples:

- "Example: Avoid Truncation When Migrating a SAS Library by Using a Two-Step Process" on page 235
- "Example: Avoid Truncation in Formats When Migrating Catalogs" on page 238

Modifying SAS Programs for a New SAS Release

SAS programs are typically saved as text files, with a .sas file extension, to be run in batch mode. You might also own custom applications that contain SAS code. PROC MIGRATE does not move SAS programs to the target installation. Instead, you must use your normal transfer tool, such as FTP.

Table 16.5 Upgrading SAS Programs

Compatibility Issue	Resolution
New features and enhancements	See the following resources: ■ SAS Documentation What's New site ■ SAS Guide to Software Updates and Product Changes
User-defined formats	User-defined formats are stored in catalogs. ■ "Example: Avoid Truncation in Formats When Migrating Catalogs" on page 238
Hardcoded values	Fix any hardcoded values such as path names, engine names, values specific to the operating environment, and so on. ■ Path names can be set in LIBNAME, FILENAME, FILE, and INFILE statements, and in options, functions, macro variables, and so on. ■ PROC MIGRATE does not update embedded LIBNAME statements in a PROC SQL view. ■ If you alter or update the operating environment, then any SAS code that directly calls operating environment commands (for example, manipulating files, printing) must be updated. See the SAS documentation for your operating environment.
Output Delivery System (ODS)	ODS templates in item stores (styles, tagsets, and tables) might not be compatible in the target release. Use the SOURCE statement in the TEMPLATE procedure to save source code to a file. Then rebuild the item store in the target session.
User-defined names	For every SAS release, see the What's New documentation for new functions, CALL routines, macros, formats, and informats. SAS names generally take precedence over user-defined names. In other words, if your existing SAS code contains a user-defined name, and that name is identical to a new SAS name, then you might encounter an error.

Migrating to SAS Viya

Customers can continue to use Base SAS programs in SAS Viya, where SAS programs run on the SAS Compute Server. You might be able to improve performance if you modify some or all of a program to run in CAS.

For SAS®9 customers who use products like Base SAS that do not rely on metadata, use the information here, in Chapter 16, "Compatibility and Migration," on page 203, to copy or migrate SAS libraries to SAS Viya.

For SAS®9 customers in environments that are metadata-based, use the SAS Migration Utility to export resources such as SAS libraries. In SAS Viya, use the SAS Environment Manager to import content. This process is called *promotion*.

See the following documents:

- SAS Viya Administration: Promotion (Import and Export)
- *An Introduction to SAS Viya Programming*
- *Migrating Data to UTF-8 for SAS Viya*
- *SAS Viya FAQ for Processing UTF-8 Data*

Examples: CEDA

Example: Understand Log Messages about CEDA

Example Code

The following PRINT procedure generates a CEDA note because the SAS session has a different data representation than that of the data set. The `mytest` data set was created on Linux in "Example: Specify a Data Representation to Avoid CEDA" on page 221.

```
libname myfiles v9 'c:\examples';
proc print data=myfiles.mytest;
run;
```

Here is the CEDA note in the SAS log. The note does not indicate an error.

Example Code 16.1 *Log Output Showing CEDA Informational Note*

```
NOTE: Data file MYFILES.MYTEST.DATA is in a format that is native
      to another host, or the file encoding does not match the session
      encoding. Cross Environment Data Access will be used, which might
      require additional CPU resources and might reduce performance.
```

The following SQL procedure attempts to update the `mytest` data set.

```
proc sql;
   insert into myfiles.mytest
      values ('other data string');
quit;
```

As shown by the following error message, CEDA does not support update processing. The update fails.

Example Code 16.2 *Log Output Showing CEDA Update Error*

```
ERROR: File MYFILES.MYTEST cannot be updated because its encoding
       does not match the session encoding or the file is in a format
       native to another host, such as SOLARIS_X86_64, LINUX_X86_64,
       ALPHA_TRU64, LINUX_IA64.
```

Key Ideas

- CEDA processing is transparent and automatic, but most users want to know when CEDA processing occurs. CEDA has several restrictions. For example, update processing is not supported.
- Compatible encodings do not require CEDA processing for transcoding.

  When SAS writes a CEDA note to the log, this note is informational. The note does not indicate an error.

  However, transcoding could result in character data loss when encodings are incompatible. For example, a code point in one encoding could represent a different character in another encoding. Therefore, always check your output when you process a data set under a different encoding.

See Also

- "Restrictions for CEDA" in *SAS Programmer's Guide: Essentials*
- "Transcoding Considerations" in *SAS National Language Support (NLS): Reference Guide*
- Chapter 16, "Compatibility and Migration," on page 203

Example: Understand Log Messages about Truncation

Example Code

This example creates the `mytrunctest` data set in a double-byte character set (DBCS) session of SAS. The session encoding is `SHIFT-JIS`. The length of the `a` variable is 22 bytes.

```
libname myfiles v9 'c:\examples';
data myfiles.mytrunctest;
```

```
   a='サンプルテキスト文字列';
run;
proc print data=myfiles.mytrunctest;
run;
```

A Unicode SAS session is started. The session encoding is `UTF-8`. In this session, the PROC PRINT output is truncated.

```
libname myfiles v9 'c:\examples';
proc print data=myfiles.mytrunctest;
run;
```

Here is the truncation warning in the SAS log.

Example Code 16.3 *Log Output Showing Truncation Warning*

```
WARNING: Some character data was lost during transcoding in the dataset
         MYFILES.MYTRUNCTEST. Either the data contains characters that are not
         representable in the new encoding or truncation occurred during transcoding.
```

The truncation occurs because the `a` variable requires more bytes in `UTF-8` encoding than in `SHIFT-JIS` encoding. To prevent truncation, use the CVP engine to expand the variable length. See "Example: Avoid Truncation When Copying a SAS Library" on page 232 and "Example: Avoid Truncation When Migrating a SAS Library by Using a Two-Step Process" on page 235.

Output 16.1 *PROC PRINT Output Showing Truncated Data*

Obs	a
1	サンプルテキス

Key Ideas

- CEDA processing is transparent and automatic. However, transcoding could result in character data loss when encodings are incompatible.
- When SAS writes an error or warning to the log about transcoding, you are advised to carefully check the output. The most common reason for a transcoding error is truncation.

  When you process a file in an encoding that uses more bytes to represent the characters, truncation could occur if the column length does not accommodate the larger character size. For example, a character might be represented in wlatin1 encoding as one byte but in UTF-8 as two bytes. Truncation can also cause "garbage" characters or incorrect characters in output.

  Some Microsoft Word characters such as smart quotation marks can also cause truncation. Those characters require more than one byte in UTF-8 encoding.
- To prevent truncation, use the CVP engine to expand the variable length.

See Also

- "Restrictions for CEDA" in *SAS Programmer's Guide: Essentials*
- "Transcoding Considerations" in *SAS National Language Support (NLS): Reference Guide*

Example: Specify a Data Representation to Avoid CEDA

Example Code

In this example, the user is running 64-bit SAS on Microsoft Windows, which has a data representation of `WINDOWS_64`. The following DATA step uses the OUTREP= data set option to create a data set that has a data representation of `LINUX_X86_64`.

```
libname myfiles v9 'c:\examples';
data myfiles.mytest (outrep=linux_x86_64);
   a='sample data string';
run;
proc contents data=myfiles.mytest;
run;
```

The SAS log displays the following message when the data set is created. The message indicates that CEDA is invoked to create the data set. However, CEDA is not invoked when the `mytest` data set is accessed in a SAS session on 64-bit Linux. (CEDA is invoked if the encoding is not compatible.)

```
NOTE: Data file MYFILES.MYTEST.DATA is in a format that is native
      to another host, or the file encoding does not match the session
      encoding. Cross Environment Data Access will be used, which might
      require additional CPU resources and might reduce performance.
```

The CEDA message is written in the log again when the CONTENTS procedure runs. Below is the output from PROC CONTENTS, showing `LINUX_X86_64` as the data representation.

Also, notice that SAS assigns the encoding `latin1 Western (ISO)`. Without an OUTREP= specification, the user's session would assign `wlatin1 Western (Windows)`.

Output 16.2 *Portion of PROC CONTENTS Output*

Data Set Name	MYFILES.MYTEST	**Observations**	1
Member Type	DATA	**Variables**	1
Engine	V9	**Indexes**	0
Created	08/27/2019 10:22:01	**Observation Length**	18
Last Modified	08/27/2019 10:22:01	**Deleted Observations**	0
Protection		**Compressed**	NO
Data Set Type		**Sorted**	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	latin1 Western (ISO)		

Key Ideas

- SAS automatically invokes CEDA processing when a file's data representation or encoding is different from the data representation or encoding of the current session.
- CEDA processing can reduce performance. Therefore, when you create a data set to be accessed in a different operating environment, you might want to specify that data representation when you create the data set.
- Use the OUTREP= data set option or LIBNAME statement option to specify the data representation of the target environment. CEDA is invoked in the current session when the data set is created.
- When you use the OUTREP= option to specify a data representation, the current SAS session encoding is ignored and a default encoding is assigned instead. This default encoding is based on the locale of the current session and the operating environment that is represented by the OUTREP= option, not the encoding of the current session.
- To change the data representation of an existing file, use the COPY procedure (or the COPY statement in the DATASETS procedure). Specify the NOCLONE option for COPY and specify OUTREP= in the LIBNAME statement.

See Also

- "OUTREP= Data Set Option" in *SAS Data Set Options: Reference*
- "OUTREP= LIBNAME Statement Option" on page 34
- "Compatible Data Representations" in *SAS Programmer's Guide: Essentials*

■ Chapter 16, “Compatibility and Migration,” on page 203

Example: Specify an Encoding to Avoid CEDA

Example Code

In this example, the user is running SAS on Linux for x64. The session encoding is `latin1 Western (ISO)`. The following DATA step uses the ENCODING= data set option to create a data set that has a UTF-8 encoding.

```
libname mylnx v9 '/mydata';
data mylnx.mytest (encoding=utf8);
   a='sample data string';
run;
proc contents data=mylnx.mytest;
run;
```

The SAS log displays the following message when the data set is created. The message indicates that CEDA is invoked to create the data set. However, CEDA is not invoked when the `mytest` data set is accessed in a UTF-8 session encoding. (CEDA is invoked if the operating environment is not compatible.)

```
NOTE: Data file MYLNX.MYTEST.DATA is in a format that is native
      to another host, or the file encoding does not match the session
      encoding. Cross Environment Data Access will be used, which might
      require additional CPU resources and might reduce performance.
```

The CEDA message is written in the log again when the CONTENTS procedure runs. Below is the output from PROC CONTENTS, showing `utf-8 Unicode (UTF-8)` as the encoding.

Output 16.3 *Portion of PROC CONTENTS Output*

Data Set Name	MYLNX.MYTEST	**Observations**	1
Member Type	DATA	**Variables**	1
Engine	V9	**Indexes**	0
Created	08/27/2019 15:31:24	**Observation Length**	18
Last Modified	08/27/2019 15:31:24	**Deleted Observations**	0
Protection		**Compressed**	NO
Data Set Type		**Sorted**	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	utf-8 Unicode (UTF-8)		

Key Ideas

- SAS automatically invokes CEDA processing when a file's data representation or encoding is different from the data representation or encoding of the current session.
- CEDA processing can reduce performance. Therefore, when you create a data set to be accessed in a different session encoding, you might want to specify that encoding when you create the data set.
- Use the ENCODING= data set option or the OUTENCODING= LIBNAME statement option to specify the encoding of the target environment. CEDA is invoked in the current session when the data set is created.

See Also

- "ENCODING= Data Set Option" in *SAS National Language Support (NLS): Reference Guide*
- "INENCODING=, OUTENCODING= LIBNAME Statement Options" on page 32
- "Encoding for NLS" in *SAS National Language Support (NLS): Reference Guide*

Example: Specify a Data Representation and UTF-8 Encoding

Example Code

This example shows the correct way to specify a nondefault encoding such as UTF-8 when you use the OUTREP= data set option or LIBNAME statement option. When you specify the OUTREP= option, SAS ignores your session encoding and assigns a default encoding. If you do not want the default encoding, you must specify an encoding option.

In this example, the user is running SAS for Windows, and they want to create a data set for a target session that is on Linux for x64. The user's encoding and the target encoding are both set to UTF-8 in their SAS configuration files. As noted above, however, the user's UTF-8 session encoding is ignored when they specify the OUTREP= option. Therefore, the ENCODING=UTF8 option is needed below.

The following DATA step specifies both data representation and encoding for the target environment.

```
libname myfiles 'C:\examples';
```

```
data myfiles.test (outrep=linux_x86_64 encoding=utf8);
   x=1;
run;
proc contents data=myfiles.test;
run;
```

The SAS log displays the CEDA message when the data set is created and when the CONTENTS procedure runs:

```
NOTE: Data file MYFILES.TEST.DATA is in a format that is native
      to another host, or the file encoding does not match the session
      encoding. Cross Environment Data Access will be used, which might
      require additional CPU resources and might reduce performance.
```

The PROC CONTENTS output shows that the data representation and encoding are correctly assigned. Now the user can transport the data set to the target environment, where it will not invoke CEDA.

Output 16.4 *Portion of PROC CONTENTS Output*

Data Set Name	MYFILES.TEST	**Observations**	1
Member Type	DATA	**Variables**	1
Engine	V9	**Indexes**	0
Created	09/03/2019 14:51:14	**Observation Length**	8
Last Modified	09/03/2019 14:51:14	**Deleted Observations**	0
Protection		**Compressed**	NO
Data Set Type		**Sorted**	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	utf-8 Unicode (UTF-8)		

Key Ideas

- SAS has a default session encoding that is based on the following two values:
  - the operating environment of the current session
  - the locale of the current session
- Many sites use the ENCODING system option to specify a nondefault session encoding. The ENCODING system option is valid in a configuration file or at SAS invocation.
- As a best practice, when you specify the OUTREP= option to create a data set in a different data representation, also specify an encoding option. Use the ENCODING= data set option or the OUTENCODING= LIBNAME statement option.

If you do not specify an encoding option, the current SAS session encoding is ignored and a default encoding is assigned instead. This default encoding is based on the following two values:

- the operating environment that is represented by the OUTREP= option
- the locale of the current session

- The default behavior also occurs with the OVERRIDE=(OUTREP=*data-rep-value*) syntax in the COPY procedure or COPY statement of the DATASETS procedure. If you want a nondefault encoding value, then specify OVERRIDE=(OUTREP= *data-rep-value* ENCODING=*encoding-value*)).

See Also

- "Default Values for DFLANG, DATESTYLE, and PAPERSIZE System Options Based on the LOCALE= System Option" in *SAS National Language Support (NLS): Reference Guide*
- "ENCODING System Option: UNIX, Windows, and z/OS" in *SAS National Language Support (NLS): Reference Guide*
- "ENCODING= Data Set Option" in *SAS National Language Support (NLS): Reference Guide*
- "INENCODING=, OUTENCODING= LIBNAME Statement Options" on page 32
- Chapter 16, "Compatibility and Migration," on page 203

Example: Determine the Encoding and Data Representation of a SAS Session or a Data Set

Example Code

The following statements return the session encoding and locale:

```
%put %sysfunc(getoption(encoding));
%put %sysfunc(getoption(locale));
```

Here is an example of the information that is written in the SAS log. The session encoding is wlatin1, and the locale is en_us.

```
1    %put %sysfunc(getoption(encoding));
WLATIN1
2    %put %sysfunc(getoption(locale));
EN_US
```

The OPTIONS procedure is another way to check the session encoding and locale:

```
proc options option=(encoding locale) define value;
```

```
run;
```

PROC OPTIONS returns detailed information about the option settings. Many lines in the example output below are omitted to highlight the relevant lines.

```
Option Value Information For SAS Option ENCODING
    Value: WLATIN1
.
.
.
Option Value Information For SAS Option LOCALE
    Value: EN_US
```

To determine the data representation for your session, create a temporary data set and submit the CONTENTS procedure or the CONTENTS statement of the DATASETS procedure.

```
data sessiontest;
   x=1;
run;
proc contents data=sessiontest;
run;
```

The `sessiontest` data set is created in the temporary Work library. Below is a portion of the PROC CONTENTS output. The OUTREP= option was not specified when the data set was created, so `sessiontest` has the data representation of the session. The output also shows the session encoding.

Output 16.5 *Portion of PROC CONTENTS Output to Learn the Session Encoding and Data Representation*

Data Set Name	WORK.SESSIONTEST	**Observations**	1
Member Type	DATA	**Variables**	1
Engine	V9	**Indexes**	0
Created	09/03/2019 14:17:06	**Observation Length**	8
Last Modified	09/03/2019 14:17:06	**Deleted Observations**	0
Protection		**Compressed**	NO
Data Set Type		**Sorted**	NO
Label			
Data Representation	WINDOWS_64		
Encoding	wlatin1 Western (Windows)		

Key Ideas

- When you share data among different environments, you might need to check the encoding and data representation of the SAS session. Knowing the locale can also be useful.
- Use PROC OPTIONS or the GETOPTION function to query the session encoding.
- Use PROC CONTENTS (or the CONTENTS statement of PROC DATASETS) to see the data representation and encoding of a data set.
- You cannot use PROC OPTIONS or the GETOPTION function to query the session's data representation value, because the OUTREP= option is not available as a system option. Instead, create a temporary data set, without using OUTREP= or ENCODING= options, so that the temporary data set has the session's data representation and encoding. Submit PROC CONTENTS (or the CONTENTS statement of the DATASETS procedure). For an example that uses PROC SQL, see Sample 55054.
- In the output of PROC CONTENTS (or the CONTENTS statement of PROC DATASETS), for some operating environments, several data representations are listed. In this case, the operating environments are compatible and do not invoke CEDA processing. However, under some compatible environments, catalogs are not compatible. See "Compatible Data Representations" in *SAS Programmer's Guide: Essentials*.
- The information that is returned by PROC CONTENTS (or the CONTENTS statement of PROC DATASETS) varies according to the operating environment and the engine.
- When you share data sets across environments, another attribute that could be helpful is **Host Created**, which is reported in the **Engine/Host Dependent Information** portion of PROC CONTENTS output.

See Also

- "GETOPTION Function" in *SAS System Options: Reference*
- "OPTIONS Procedure" in *SAS System Options: Reference*
- "CONTENTS Procedure" in *Base SAS Procedures Guide*

Examples: Migrate SAS Libraries

Example: Migrate a SAS Library

Example Code

The following MIGRATE procedure example migrates members in a SAS library to take advantage of features that are provided in a newer SAS release. This example does not use a SAS/CONNECT or SAS/SHARE server, which is required in some cases.

Run this code in a session of the SAS release that you are migrating to.

```
libname myfiles 'c:\example';
libname target 'd:\new';
proc migrate in=myfiles out=target;
run;
```

The SAS log shows the results of the migration. Notice that PROC MIGRATE calls PROC CPORT to migrate catalogs.

```
NOTE: The BUFSIZE= option was not specified with the MIGRATE procedure.
The migrated library members will use the current value for BUFSIZE. For more
information, see the PROC MIGRATE documentation.
NOTE: Migrating MYFILES.CONTAINERS to TARGET.CONTAINERS (memtype=VIEW).
NOTE: Migrating MYFILES.FLOWERS to TARGET.FLOWERS (memtype=DATA).
NOTE: Simple index plantname has been defined.
NOTE: There were 7 observations read from the data set MYFILES.FLOWERS.
NOTE: The data set TARGET.FLOWERS has 7 observations and 3 variables.
NOTE: Migrating MYFILES.RESTOCK to TARGET.RESTOCK (memtype=DATA).
NOTE: There were 7 observations read from the data set MYFILES.RESTOCK.
NOTE: The data set TARGET.RESTOCK has 7 observations and 4 variables.

NOTE: PROC CPORT begins to transport catalog MYFILES.FORMATS
NOTE: Entry TESTFMT.FORMAT has been transported.
NOTE: PROCEDURE MIGRATE used (Total process time):
      real time           2.34 seconds
      cpu time            0.15 seconds
```

If you migrate to a different operating environment (for example, from Windows to UNIX), PROC MIGRATE has some limitations. For example, the following log note says that cross environment data access (CEDA) was used. This message informs you that performance can be slowed by CEDA.

```
NOTE: Data file MYFILES.FLOWERS.DATA is in a format that is native to another
host, or the file encoding does not match the session encoding. Cross Environment
Data Access will be used, which might require additional CPU resources and might
reduce performance.
```

An error message like the following is also due to CEDA, and it indicates that the `formats` catalog was not migrated. To migrate catalogs with PROC MIGRATE to an incompatible operating environment, you must use a SAS/CONNECT or SAS/SHARE server to access the IN= library.

```
ERROR: File MYFILES.FORMATS.CATALOG was created for a different operating system.
WARNING: No data is available.
```

Key Ideas

- PROC MIGRATE is usually the best way to migrate members in a SAS library to the current SAS version. PROC MIGRATE is a one-step copy procedure that retains the data attributes that most users want in a data migration.
- If either of the following issues is present, then a SAS/CONNECT or SAS/SHARE server is required, and different syntax is used. See "Migrating from a SAS®9 Release by Using SAS/CONNECT" in *Base SAS Procedures Guide*.
  - if you do not have direct access to the source library from the target session via a Network File System (NFS)
  - if the source library contains catalogs and if the processing invokes CEDA on the target session
- Alternatively, if you have direct access to the source library through NFS, then you can use cross-environment data access (CEDA) instead of migrating. This Read-Only access is automatic and transparent, but you must be aware of the restrictions. See "Cross-Environment Data Access" in *SAS Programmer's Guide: Essentials*.
- If you are changing to a different character encoding that uses more bytes to represent the characters, you might want to use the CVP engine as part of the copy or migration process. See "Example: Avoid Truncation When Migrating a SAS Library by Using a Two-Step Process" on page 235.

See Also

- "MIGRATE Procedure" in *Base SAS Procedures Guide*
- Chapter 16, "Compatibility and Migration," on page 203

Example: Migrate a SAS Library across Environments by Using SAS/CONNECT

Example Code

The following MIGRATE procedure example migrates members in a SAS library to take advantage of features that are provided in a newer SAS release. The example uses a SAS/CONNECT server, which is required in some cases. In this example, a spawner starts a session on the SAS/CONNECT server.

```
options comamid=tcp;                                /* 1 */
%let myserver=host.name.com;                        /* 2 */
signon myserver.__1234 user=userid password='mypw'; /* 3 */
libname source '/mydata' server=myserver.__1234;    /* 4 */
libname target v9 '/mylinuxdata';                   /* 5 */
proc migrate in=source out=target;                  /* 6 */
run;
signoff myserver.__1234;                            /* 7 */
```

1. The COMAMID= option specifies to use TCP/IP as the communications access method for connecting to the SAS/CONNECT server.
2. The %LET statement creates the `myserver` macro variable. Use this variable to specify the name of your remote SAS/CONNECT server.
3. The SIGNON statement instructs the spawner to start a session on the SAS/CONNECT server that is named in the `myserver` macro variable. If your port number or service name is defined in the macro variable, then omit it from the SIGNON.
4. This LIBNAME statement assigns the `source` library to the location of the library that is to be migrated. The SERVER= argument specifies the `myserver` macro variable and the port number.
5. This LIBNAME statement assigns the `target` library to an existing location where the migrated library is to be stored.
6. PROC MIGRATE migrates the library members from the `source` library to the `target` library.
7. The SIGNOFF command ends the connection to the server.

Key Ideas

- Here are two reasons to use a SAS/CONNECT or SAS/SHARE server:
  - You can use a server to migrate across machines when you do not have direct access.

- You are required to use a server if both of the following two conditions are met:
  - The source library contains catalogs.
  - Processing would invoke CEDA in the target session.

  In general, CEDA is invoked when you migrate to an incompatible operating environment. For more information about CEDA, see "Cross-Environment Data Access" in *SAS Programmer's Guide: Essentials*.

- The operating environment and encoding of the SAS/CONNECT server must be compatible with the source library.
- If you have direct access to the source library through NFS, and if incompatible catalogs are not present, then you can migrate without using a SAS/CONNECT or SAS/SHARE server. See "Example: Migrate a SAS Library" on page 229.
- Alternatively, if you have direct access to the source library through NFS, then you might be able to use CEDA instead of migrating. This Read-Only access is automatic and transparent, but you must be aware of the restrictions. See "Cross-Environment Data Access" in *SAS Programmer's Guide: Essentials*.

See Also

- "MIGRATE Procedure" in *Base SAS Procedures Guide*
- *SAS/CONNECT User's Guide*

Example: Avoid Truncation When Copying a SAS Library

Example Code

To run this example, first create a data set named `myclass` as in "Example: Assign the V9 Engine in a LIBNAME Statement" in *SAS Programmer's Guide: Essentials*. Run PROC CONTENTS to see the length of the variables:

```
libname myfiles v9 'c:\examples';
proc contents data=myfiles.myclass;
run;
```

In the PROC CONTENTS output, notice the two character variables. Name has a length of 8, and Sex has a length of 1.

Output 16.6 *PROC CONTENTS Showing Variable Lengths before Expansion*

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
3	Age	Num	8
4	Height	Num	8
1	Name	Char	8
2	Sex	Char	1
5	Weight	Num	8

The example below uses the CVP engine with the V9 engine to expand the size of character variables. The CVP engine can help you avoid truncation if you copy a data set to an encoding that uses more bytes to represent the characters.

```
libname srclib cvp 'c:\examples' cvpengine=v9 cvpmult=2.5; /* 1 */
libname target v9 'c:\temp';                               /* 2 */
proc copy in=srclib out=target;                            /* 3 */
run;

proc contents data=target.myclass;                         /* 4 */
run;
```

1 This LIBNAME statement assigns the `srclib` library to the CVP engine and the location of the data that you want to copy. The CVPENGINE= option specifies the V9 engine as the underlying engine to process the data. The CVPMULT= option specifies a multiplication factor of 2.5 to expand all character variables.

2 This LIBNAME statement assigns the `target` library to contain the copied data.

3 The COPY procedure copies the `srclib` library to the `target` library. During the copy, the CVP engine expands the character variable lengths 2.5 times larger.

4 The CONTENTS procedure shows that the lengths of the character variables have been multiplied by 2.5:

For Name, 8 × 2.5 = 20.

For Sex, 1 × 2.5 = 2.5, which is 3 when rounded up to a whole number.

Output 16.7 *PROC CONTENTS Showing Variable Lengths after Expansion*

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
3	Age	Num	8
4	Height	Num	8
1	Name	Char	20
2	Sex	Char	3
5	Weight	Num	8

Key Ideas

- When you copy a data set to an encoding that uses more bytes to represent the characters, truncation might occur if the column length does not accommodate the larger character size. For example, a character might be represented in wlatin1 encoding as one byte but in UTF-8 as two bytes.
- If an error in the log states `character data was lost during transcoding`, it usually indicates that truncation has occurred. You can troubleshoot the error by using the CVP engine to expand the length of character variables.
- The CVPMULTIPLIER=2.5 value is usually sufficient to avoid truncation. If disk space is a concern, try a smaller value such as 1.5.
- The default CVPFORMATWIDTH=YES option expands the length for formats but does not affect user-defined formats. For user-defined formats, see "Example: Avoid Truncation in Formats When Migrating Catalogs" on page 238.
- If you copy a library to a different operating environment, or to a different character encoding, you probably want to specify options such as NOCLONE for PROC COPY. NOCLONE does not copy certain data set attributes, such as data representation and character encoding, so that the library is compatible in the target environment.

  As an alternative to the NOCLONE option, you can use the OVERRIDE= option to specify ENCODING= and OUTREP= options. This method keeps other data set attributes from the source library, so make sure that you want those attributes.
- If the library contains indexes or integrity constraints, then cross-environment data access (CEDA) is an important issue. If you have experienced truncation across environments, then SAS is probably using CEDA to access the files. Check the log for a CEDA message.

  PROC COPY does not copy indexes or integrity constraints under CEDA processing, but PROC MIGRATE does. However, PROC MIGRATE does not support the CVP engine to expand character column lengths. Therefore, if you want to migrate indexes or integrity constraints, you must copy the library with the CVP engine first and then migrate (in other words, a two-step process). See "Example: Avoid Truncation When Migrating a SAS Library by Using a Two-Step Process" on page 235.
- PROC COPY or the COPY statement of the DATASETS procedure do not preserve an audit trail.

See Also

- *SAS National Language Support (NLS): Reference Guide*
- Chapter 16, "Compatibility and Migration," on page 203
- "COPY Procedure" in *Base SAS Procedures Guide*

Example: Avoid Truncation When Migrating a SAS Library by Using a Two-Step Process

Example Code

This example is similar to "Example: Avoid Truncation When Copying a SAS Library" on page 232. The library is migrated to an environment where the encoding uses more bytes than the source environment to represent the characters. In this example, however, the source library contains indexes or integrity constraints, which are not supported under CEDA processing. A two-step process is necessary.

To run this example, create a data set that has an index. For this example, the data is created in a UNIX environment that has Latin1 encoding.

```
libname myfiles v9 '/mydata';
data myfiles.myclass (index=(age));
   set sashelp.class;
run;
```

Here is the two-step process to avoid truncation while migrating a library:

1. To avoid CEDA processing, perform this step in the source environment where the data was created. Use PROC COPY with the CVP engine to expand the column length for all character variables. Do not specify NOCLONE.

   ```
   libname myfiles cvp '/mydata' cvpmult=2.5 cvpformatwidth=yes;
   libname mycopy v9 '/mydatacopy';
   proc copy in=myfiles out=mycopy constraint=yes;
   run;
   ```

2. Perform this step in the target environment. For this example, the target is a compatible UNIX environment and the encoding is UTF-8. In the target environment, use PROC MIGRATE to migrate the library.

   ```
   libname mycopy '/mydatacopy';
   libname target '/new';
   proc migrate in=mycopy out=target;
   run;
   ```

   This example does not migrate catalogs if they are not compatible with the target operating environment. Instead, see "Example: Migrate a SAS Library across Environments by Using SAS/CONNECT" on page 231. Also, audit trails are not migrated, because the first step uses PROC COPY, which does not support audit trails.

Key Ideas

- If the library contains indexes or integrity constraints, then CEDA might cause issues. If you have experienced truncation across environments, then SAS is probably using CEDA to access the files. Check the log for a CEDA message.

  Although PROC COPY does not copy indexes or integrity constraints under CEDA processing, PROC MIGRATE does. However, PROC MIGRATE does not support the CVP engine to expand character column lengths. Therefore, if you want to migrate indexes or integrity constraints, you must copy the library with the CVP engine first and then migrate (in other words, a two-step process).
- The CVPMULTIPLIER=2.5 value is usually sufficient in order to avoid truncation. If disk space is a concern, try a smaller value such as 1.5.
- PROC MIGRATE migrates a library's members and member attributes that most customers want to keep in a migration.

See Also

- "COPY Procedure" in *Base SAS Procedures Guide*
- "MIGRATE Procedure" in *Base SAS Procedures Guide*
- *Migrating Data to UTF-8 for SAS Viya*
- "%COPY_TO_NEW_ENCODING Macro Function" in *SAS National Language Support (NLS): Reference Guide*

Example: Migrate Catalogs to Avoid CEDA Limitations

Example Code

In this example, PROC MIGRATE migrates a SAS library that contains catalogs. The target environment uses a different data representation or character encoding than the source library. Because the library contains catalogs, the example uses a SAS/CONNECT spawner to access the source library.

Submit this code in a session that is using the current release of SAS 9.

```
options comamid=tcp;                                /* 1 */
%let myserver=host.name.com;                        /* 2 */
signon myserver.__1234 user=userid password='mypw'; /* 3 */
libname source '/mydata' server=myserver.__1234;    /* 4 */
```

```
libname target v9 '/mylinuxdata';                    /* 5 */
proc migrate in=source out=target;                   /* 6 */
run;
signoff myserver.__1234;                             /* 7 */
```

1 The COMAMID= option specifies to use TCP/IP as the communications access method for connecting to the SAS/CONNECT server.

2 The %LET statement creates the `myserver` macro variable. Use this variable to specify the name of your remote SAS/CONNECT server.

3 The SIGNON statement instructs the spawner to start a session on the SAS/CONNECT server that is named in the `myserver` macro variable. If your port number or service name is defined in the macro variable, then omit it from the SIGNON.

4 This LIBNAME statement assigns the `source` library to the location of the library that is to be migrated. The SERVER= argument specifies the `myserver` macro variable and the port number.

5 This LIBNAME statement assigns the `target` library to an existing location where the migrated library is to be stored.

6 PROC MIGRATE migrates the library members from the `source` library to the `target` library.

7 The SIGNOFF command ends the connection to the server.

Key Ideas

- Catalogs are not supported under cross-environment data access (CEDA) processing. CEDA is invoked when the character encoding or data representation of the SAS file and the SAS session do not match. CEDA processing is common when users upgrade to a new release of SAS that runs in a different operating environment.
- PROC MIGRATE is one way to change the character encoding or data representation of library members to match the current SAS session. PROC MIGRATE is also usually the best way to migrate the members in a SAS library to the current SAS version. PROC MIGRATE retains the data attributes that most users want in a data migration.
- If you are changing to a different data representation or character encoding and your library contains catalogs, then PROC MIGRATE requires a SAS/CONNECT or SAS/SHARE server libref in the IN= argument. (If the SAS/CONNECT or SAS/SHARE server is running in a release earlier than SAS 9.1.3, then use the SLIBREF= option.)
- To avoid CEDA processing, the server's operating environment and encoding must be compatible with the source library.
- If you are not changing to a different data representation or character encoding, then the PROC MIGRATE code is simpler. You can access catalogs and other library members by using a Network File System (NFS), or you can use a SAS/CONNECT or SAS/SHARE server as shown in this example.

See Also

- "Operations That Affect Catalogs" on page 171
- "MIGRATE Procedure" in *Base SAS Procedures Guide*
- "Cross-Environment Data Access" in *SAS Programmer's Guide: Essentials*
- "%LET Macro Statement" in *SAS Macro Language: Reference*
- "COMAMID= System Option" in *SAS/CONNECT User's Guide*
- "SIGNON Statement" in *SAS/CONNECT User's Guide*

Example: Avoid Truncation in Formats When Migrating Catalogs

Example Code

In this example, the user moves a formats catalog from a double-byte character set (DBCS) environment to a UTF-8 environment. A user-defined format contains Japanese characters. In UTF-8, the characters require more bytes to print correctly.

For this example, run the following code in a SAS 9 session with DBCS. If you do not have a DBCS version of SAS, then substitute any other character string for the Japanese characters.

```
libname mytest v9 'c:\examples';
proc format lib=mytest;
   value $waterformat                  /* 1 */
         'a'='湖'    /**/
         'b'='海洋';                    /* 2 */
run;
proc format library=mytest.formats
     cntlout=mytest.outformats         /* 3 */
     (keep=fmtname start label type);  /* 4 */
   select $waterformat;                /* 5 */
run;
```

1. The VALUE statement of PROC FORMAT creates the `$waterformat.` format. A catalog name is not specified, so the format is stored in the default catalog, `mytest.formats`.
2. This format assigns two labels. The label `'湖'` is assigned to the value `'a'`. The label `'海洋'` is assigned to the value `'b'`.
3. In the next PROC FORMAT step, the CNTLOUT= option creates the `outformats` data set. This output data set contains information about formats that can be used to rebuild the formats.

4 The KEEP= data set option keeps only the variables `fmtname`, `start`, `label`, and `type`.

5 The SELECT statement selects only the `$waterformat.` format for the purposes of this demonstration. Typically, you would omit this statement, and include all the formats in the output data set.

Open SAS with Unicode Support, or invoke a SAS session with `-encoding utf8`. Alternatively, you can run this code in the same SAS session as in the first step to see how the code works.

```
libname mytest cvp "c:\examples" ;                                /* 1 */
libname mynew "c:\new";
proc format library=mynew.utf8formats cntlin=mytest.outformats; /* 2 */
run;
```

1 Specify the CVP engine in the LIBNAME statement for the location of the `mytest.outformats` data set. The CVP engine expands the length for all the character variables.

2 PROC FORMAT imports the formats from `mytest.outformats` to a new formats catalog, `mynew.utf8formats`. (In this example, only one format is in the `mytest.outformats` data set.) When you rebuild the formats catalog, the catalog entries have the encoding of the target environment.

Run the following code to test the new, expanded `$waterformat.` format:

```
data test;                              /* 1 */
   input item $;
   datalines;
a
b
c
;
run;

options fmtsearch=(mynew.utf8formats); /* 2 */
proc print data=test noobs;
   format item $waterformat. ;          /* 3 */
run;
```

1 The DATA step creates a temporary data set, `test`, in the Work library.

2 The OPTIONS statement submits the FMTSEARCH= option to specify the location of the `mynew.utf8formats` catalog.

3 The FORMAT statement assigns `$waterformat.` to the `item` variable.

Output 16.8 *PROC PRINT Output of Formatted Data Set*

item
湖
海洋
c

Key Ideas

- CEDA does not support catalogs. If you migrate to an incompatible encoding, you must migrate any user-defined formats catalogs to the target session's encoding.
- Be aware of the possibility of truncation when you migrate to an encoding such as UTF-8 that uses more bytes to represent the characters. Truncation might occur if any variable in a formats catalog has a length that does not accommodate the larger character size.
- To expand format lengths, use PROC FORMAT in a two-step process.

  1. Use the CNTLOUT= option with PROC FORMAT to write the formats to an output data set. Use the KEEP= data set option to keep only the necessary variables. At a minimum, keep the `fmtname`, `start`, and `label` variables. The `type` variable is not necessary if all format names begin with the `@` or `$` prefix.
  2. In the target session, use the CNTLIN= option with PROC FORMAT to rebuild the formats catalog from the data set. Assign two LIBNAME statements:
     - Specify the CVP engine in the LIBNAME statement for the data set.
     - Specify a different LIBNAME statement for the new formats catalog. Use a different location, and do not specify the CVP engine.
- If you do not have direct access to the output data set (for example, by using NFS), then you must insert the following intermediate tasks between step 1 and step 2: In the source environment, after step 1, use the CVP engine and a DATA step to expand the character variables in the output data set. Then, in the target environment, before step 2, use a SAS tool (such as PROC MIGRATE or PROC COPY) to migrate or copy the expanded output data set. After these intermediate tasks, continue with step 2. However, you do not need to use the CVP engine, because you already expanded the character variables in the output data set.
- In the CNTLOUT= data set, a best practice is to keep only the variables (`fmtname`, `start`, `label`, and `type`). Keeping only the necessary variables can reduce the possibility of truncation errors when you use the new format in the target environment. The CVP engine does not increase the values of numeric variables such as `default` or `length`. If you include numeric variables in the output data set, you might have to increase the values of those variables in the rebuilt formats catalog.
- For expansion, specify the CVP engine and not the options such as CVPMULT= or CVPBYTES=. The CVP engine determines the appropriate expansion. If you attempt to specify a larger expansion than is needed, then in the second step, PROC FORMAT reduces those lengths to the appropriate lengths. Here is an example: You specify CVPMULT=4, so the engine expands a `label` variable from 4 to 16 bytes. However, you find that in the import step, PROC FORMAT assigns only 6 bytes to the `label`. PROC FORMAT reduces the lengths automatically but cannot expand lengths.
- When truncation is an issue, you must use the two-step PROC FORMAT process for the following reasons.

  - In most cases, PROC MIGRATE is recommended for migrating a SAS library. However, PROC MIGRATE does not support using the CVP engine to expand variable lengths.
  - Normally the CVP engine is used with PROC COPY or the COPY statement of PROC DATASETS. In that usage, the default CVPFORMATWIDTH=YES option expands the length for SAS formats but does not affect user-defined formats.
- If truncation is not an issue, then do not use the CVP engine. You can use this example without CVP, or use one of the following methods:
  - "Example: Migrate Catalogs to Avoid CEDA Limitations" on page 236
  - "Example: Copy a SAS Library by Using a Transport File" on page 241
- Even when the encoding is compatible, you might experience transcoding errors that are not written to the log. For a summary of potential transcoding errors, see "How Character Encoding Affects Compatibility" on page 206.
- To see detailed information about a formats catalog, use the CNTLOUT= option in PROC FORMATS to write the formats to an output data set. Then use PROC CONTENTS or the CONTENTS statement in PROC DATASETS to view the names and lengths of the variables. Use PROC PRINT to view the values in the variables.

See Also

- "FORMAT Procedure" in *Base SAS Procedures Guide*
- "CVPBYTES=, CVPENGINE=, CVPFORMATWIDTH=, CVPMULTIPLIER= LIBNAME Statement Options" on page 28

Example: Copy a SAS Library by Using a Transport File

Example Code

This example copies a SAS library across environments by using the CPORT and CIMPORT procedures. A multistep process is necessary:

1. In the source environment, use PROC CPORT to create a transport file.
2. Use communication software (such as FTP) or a storage device to move the transport file to the target environment. If you use FTP, transfer the file in binary mode.
3. In the target environment, use PROC CIMPORT to import the library from the transport file.

For step 1, PROC CPORT creates the transport file `mytransfer`, which is referenced by the fileref `tranfile`.

```
libname source 'c:\example';
filename tranfile 'c:\myfiles\mytransfer';
proc cport library=source file=tranfile;
run;
```

In the log, notice that `containers` is not ported, because it is a SAS view.

```
NOTE: Not porting SOURCE.CONTAINERS because memtype is not supported.

NOTE: PROC CPORT begins to transport data set SOURCE.FLOWERS
NOTE: The data set contains 3 variables and 7 observations.
      Logical record length is 24.
NOTE: Transporting data set index information.

NOTE: PROC CPORT begins to transport catalog SOURCE.FORMATS
NOTE: The catalog has 1 entries and its maximum logical record length is 120.
NOTE: Entry TESTFMT.FORMAT has been transported.

NOTE: PROC CPORT begins to transport data set SOURCE.RESTOCK
NOTE: The data set contains 4 variables and 7 observations.
      Logical record length is 40.
NOTE: PROCEDURE CPORT used (Total process time):
      real time           18.14 seconds
      cpu time            0.17 seconds
```

For step 2, the user copies the `mytransfer` file from their Windows environment to a UNIX environment.

For step 3, PROC CIMPORT creates the `target` library by importing the contents of `mytransfer`.

```
libname target '/mydata/example';
filename tranfile '/mydata/mytransfer';
proc cimport library=target infile=tranfile;
run;
```

The log indicates that the import was successful.

```
NOTE: PROC CIMPORT begins to create/update data set TARGET.FLOWERS
NOTE: The data set index plantname is defined.
NOTE: Data set contains 3 variables and 7 observations.
      Logical record length is 24

NOTE: PROC CIMPORT begins to create/update catalog TARGET.FORMATS
NOTE: Entry TESTFMT.FORMAT has been imported.
NOTE: Total number of entries processed in catalog TARGET.FORMATS: 1

NOTE: PROC CIMPORT begins to create/update data set TARGET.RESTOCK
NOTE: Data set contains 4 variables and 7 observations.
      Logical record length is 40

NOTE: PROCEDURE CIMPORT used (Total process time):
      real time           0.74 seconds
      cpu time            0.00 seconds
```

Key Ideas

- PROC CPORT and PROC CIMPORT have several limitations as compared to PROC MIGRATE. Only use this method for migration if PROC MIGRATE would require SAS/CONNECT or SAS/SHARE software, and you do not have access to that software. PROC MIGRATE migrates an entire library, including data sets, catalogs, and most other member types. See "Example: Migrate a SAS Library across Environments by Using SAS/CONNECT" on page 231.
- PROC CPORT supports SAS data sets and catalogs but not other member types.
- If you use FTP to move the transport file to the target environment, transfer the file in binary mode.
- When you are transcoding to a new encoding, truncation could occur. If truncation occurs, you must expand variable lengths. You can either use the CVP engine with PROC CPORT or use the EXTENDVAR= option with PROC CIMPORT.
- Transport files that are created by the CPORT procedure are not interchangeable with transport files that are created by the XPORT engine.

See Also

- "CPORT Procedure" in *Base SAS Procedures Guide*
- "CIMPORT Procedure" in *Base SAS Procedures Guide*
- "PROC CPORT and PROC CIMPORT" in *Moving and Accessing SAS Files*
- Chapter 16, "Compatibility and Migration," on page 203