CHAPTER

4

Writing the End-User Application

Audience

This chapter is recommended primarily for programmers who write applications that access shared data. It may also be of interest to users of SAS/SHARE applications and server administrators.

Accessing Libraries through a Server

To access a SAS data library or an external DBMS through a SAS/SHARE server, you must use the REMOTE engine to define a libref for the library. Use a LIBNAME statement that specifies the libref and that identifies the library or the DBMS and the SAS/SHARE server that you will use to access it. Here is a typical LIBNAME statement:

```
libname invoice '/dept/acct/data/invoice' server=share1;
```

In this example, the engine name REMOTE, which is normally specified between the libref and the pathname, is omitted because it is implied by the SERVER= option.

If the library is pre-defined to the server by a server administrator, you can omit the pathname and use only the libref, which is defined for the library in the server SAS session, to identify the library:

```
libname invdata server=share1;
```

In this case, the REMOTE engine and the server assume that the libref that you define for your SAS session is also the libref that is defined by the server administrator in the server session. Omitting the path name protects your application if the path name for the library has changed.

If the library is pre-defined to the server but you want to define a different libref from the one that is defined in the server session, use the SLIBREF= option to specify the server libref:

```
libname invoice server=share1 slibref=invdata;
```

If the server runs with the option NOALLOC in effect, all libraries that are accessed through that server must be pre-defined by the server administrator.

For details about the LIBNAME statement syntax, see Chapter 9, "The LIBNAME Statement," on page 103.

Using Macros to Generate a LIBNAME Statement

Hard coding the server name in a LIBNAME statement can be a problem if the server administrator shifts one server's traffic to another server, thereby invalidating the server name.

You can avoid this problem by using a SAS macro to generate the required LIBNAME statement. If you use a macro in your end-user application, you can change the name of the server in one place, even though multiple applications access data through that server. This makes it easier to maintain your SAS programs.

SAS/SHARE software provides macros for generating LIBNAME statements. To associate a server name with a SAS data library, register the library in a table that the server administrator maintains. Instead of specifying a LIBNAME statement to access the library, use the LIBDEF macro. When you invoke the LIBDEF macro, it searches the table of registered data libraries for the specified library. When the LIBDEF macro finds the library, it uses the associated server name and the specified libref to construct the LIBNAME statement. Before the first invocation of the LIBDEF macro in a SAS execution, you must invoke the SHRMACS macro with the keyword USER.

```
%shrmacs(user);
```

The SHRMACS macro generates and compiles other SAS/SHARE macros and builds the SAS server-alias and library-alias tables in which server libraries are registered.

To invoke the LIBDEF macro, specify the libref and the optional server library, as shown in this example:

```
%libdef(datalib<,SAS-data-library>);
```

For more information about using the LIBDEF macro, see "Generating a LIBNAME Statement (LIBDEF)" on page 85. For complete details about macro syntax, see Chapter 14, "SAS/SHARE Macros," on page 133. Contact your server administrator for information about how the macros and tables are implemented and how they are used at your site.

How Data Object Locking Works in your Programming Environment

The SAS/SHARE lock manager enables multiple clients to share the same SAS file at the same time. The server's ability to manage multi-client access of selected data objects hinges on a complex set of locking rules that are affected by many factors. Locking considerations must be addressed for each programming product (both SAS products and other products) that you use to write applications to access data through the SAS/SHARE server. For complete information about locking concepts, see Chapter 5, "Locking SAS Data Objects," on page 59.

Traditional SAS Programming Considerations

The following sections discuss considerations, including locking implications, that are specific to SAS programming:

- data step processing
- using ordered data
- using non-interactive SAS applications.

DATA Step Processing

The following example shows the effect of implicit locking when two clients, John and Maria, share access to the SAS data set FUEL in their respective sessions at the same time.

While Maria is editing data set DATALIB.FUEL in an FSEDIT window, John can use a DATA step to read DATALIB.FUEL by using a SET, a MERGE, or an UPDATE statement. He cannot create a new version of DATALIB.FUEL, but he can create other data sets or written reports:

```
data _null_;
   set datalib.fuel;
   file report ps=24 n=ps;
   ...
run;


data composit;
   merge datalib.fuel fuel96;
run;
```

If John uses a SET statement to read DATALIB.FUEL, he cannot specify the options KEY= or POINT= unless he overrides the member-level control. By default, member-level control is required for a SET statement that includes one of those options:

```
data pressure;
   set fuel (keep=fuel maxpress);
   set datalib.fuel (cntllev=rec) key=fuel;
   ...
run;
```

If John uses an UPDATE statement or a SET or a MERGE statement with a BY statement to read DATALIB.FUEL, he may want to specify member-level control to ensure that the data set remains properly ordered while his DATA step runs:

```
data composit;
   merge datalib.fuel (cntllev=mem)
         fuel96;
   by grade;
run;
```

Although John cannot create a new version of DATALIB.FUEL, he can use a DATA step with a MODIFY statement to update it:

```
data datalib.fuel;
   modify datalib.fuel;
   if (grade='03N') then
       do;
           grade='3Np';
           revised=today();
           replace datalib.fuel;
       end;
run;
```

When this DATA step tries to update an observation that Maria is editing in her FSEDIT window, the replace operation for that observation fails because Maria has the observation locked. This failure causes the DATA step to terminate immediately. However, any observations that are updated prior to the termination retain their updated values.

For applications that update shared data with the MODIFY statement, it is very important to include error-checking statements to prevent this kind of failure and premature termination. The return codes from the read operation (performed by the MODIFY statement) and the update operations (performed by the REPLACE, the OUTPUT, and the REMOVE statements) are available in the automatic variable _IORC_. For example, the preceding DATA step would be more complete if it looked like this:

```
data datalib.fuel;
   modify datalib.fuel;
   if (grade='03N') then
       if (_iorc_ = 0) then
         /* read with lock for update successfully */
           do;
             grade='3Np';
             revised=today();
             replace datalib.fuel;
           end;
       else
             put 'Observation' _n_
                '(fuel' fuel ') was not replaced.';
run;
```

The DATA step checks the value of _IORC_ to determine if a warning or error condition occurred while reading an observation from DATALIB.FUEL. If the observation was read successfully, it can be replaced. Otherwise, a message is written to the SAS log to record the failure and to identify the observation that was not updated.

To check for specific values of _IORC_, use the SYSRC macro. For example,

```
data datalib.fuel;
   modify datalib.fuel;
   if (grade='03N') then
       if (_iorc_ = 0) then
```

```
            /* read with lock for update successfully */
             do;
                 grade='3Np';
                 revised=today();
                 replace datalib.fuel;
             end;
         else if (_iorc_ = %sysrc(_SWNOUPD)) then
                  put 'Observation' _n_
                     '(fuel' fuel ') was not replaced.';
         else
                  put 'Observation' _n_
                     '(fuel' fuel ') read with rc' _iorc_;
run;
```

For complete information about the MODIFY statement, see *SAS Language Reference: Dictionary*. For information about the SYSRC macro and _IORC_ return code checking, see *SAS Macro Language: Reference*.

Using Ordered Data in a Shared Environment

Many applications that use SAS data sets require the data to be stored in sorted order according to the value (or values) of one or more variables. In Version 6 of SAS software, indexes can be defined on one or more variables in a SAS data file to help SAS applications maintain the order of the observations in SAS data sets. This prevents the application from having to sort the entire data set with each use. Because SAS determines if indexes are used in its processing, indexes must be defined carefully to avoid inadvertently causing less efficient SAS performance. For more information about defining indexes, see the chapter about SAS files in *SAS Language Reference: Concepts*.

Shared SAS data sets are frequently ordered according to one or more variables. Programmers who develop SAS applications that use shared, ordered data need to be aware of the two ways in which shared data can be used:

- concurrent-update applications
- reporting applications.

Concurrent-update applications generally involve several users who repeat the following type of cycle: position to an observation, update data, position to another observation, update that data, and so on. If these users specify a WHERE clause to position to the next observation and the variable (or variables) in the WHERE clause are indexed, indexing can improve the server's performance by minimizing the server's effort to search for each observation. Because the concurrent-update users' access pattern tends to be random rather than sequential, processing with an index generally does not increase the amount of physical I/O that is performed by the server for each user.

On the other hand, reporting applications frequently read the data of one or more shared data sets - capturing the data as they are at that moment - and develop a report from that data. If the application uses a BY statement so that the data is returned in sorted order, the server's performance can vary greatly while the data is being read. The server's performance can vary due to a number of factors, such as whether the BY variable is indexed and whether options are added to the BY statement that cause the index not to be used.

Because of this performance concern, it is recommended that the server read the data in its physical, unsorted order, and then sort the data in the SAS process that was used to produce the report. One way to do this is by using the SORT procedure to read the data in physical order through the server and to produce a sorted data file in your library WORK.

```
proc sort data=datalib.fuel out=fuel;
   by area;
run;
```

Another way is to use the SQL procedure to create a temporary SAS data file and to sort it using an ORDER BY clause.

```
proc sql;
create table fuel as
   select * from datalib.fuel
   order by area;
```

Defining more indexes than are necessary on shared SAS data sets can increase the amount of memory that a server needs. Avoid defining indexes that will not be used by your applications when they access shared data sets through a server.

Using Non-Interactive SAS Applications in a Shared Environment

Shared data is sometimes maintained by SAS applications that use the batch or non-interactive methods of processing. As in interactive applications, these non-interactive applications update SAS files through a server. Such applications may be written as one or more SCL programs or as a combination of DATA steps and procedures.

Generally, it is important that no other users access any of the shared SAS files while this type of application runs. To ensure uninterrupted access, use either the LOCK statement or the SCL LOCK function (for SCL programs) at the beginning of your program to gain exclusive access to the SAS files that your application uses. After your program has completed, be sure to release your exclusive access to these SAS files so that other users can access them.

The example that follows uses a two-step SAS program that gains exclusive access to a specific SAS file by using the LOCK statement. First, the program opens a shared SAS data set and copies to another data set data that has not been updated for one month. Then, the program deletes the data from the original data set. After the program has completed processing, it clears the exclusive lock.

```
%libdef(datalib);

/* Try to get exclusive access to the SAS data set. */
lock datalib.fuel;

/* Did we succeed? If not, go no further. */
data _null_;
   x=symget('SYSLCKRC');
   put x=hex16.;
   if (x^='0') then endsas;
run;

/* Copy any observations that have not been updated in */
/* 30 days to a different, locally-accessed library.  */

data permlib.a;
   drop now;
   retain now;
   if (_N_=1) then now=today();
   set datalib.fuel;
   if (accdate<(now- 30)) then output permlib.a;
```

```
run;

/* Now delete those observations from the master file. */

proc sql;
delete from datalib.fuel where (accdate<(today()- 30));
quit;

/* All done! Release the lock on the master file. */

lock datalib.fuel clear;
```

SQL Programming Considerations

The REMOTE engine supports the SQL procedure Pass-Through Facility (RSPT), which allows you to pass SQL statements to a remote SAS SQL processor or DBMS through a SAS/SHARE server or a SAS/CONNECT remote host.

You can use RSPT to reduce network traffic and to shift CPU load by sending queries for remote data to a remote server.

Note: If the server is a SAS/CONNECT remote host, you can also remotely submit queries with the RSUBMIT statement to achieve the same goals. △

For example, if you specify

```
select emptitle as title, avg(empyears), freq(empnum)
        from sql.employee
        group by title
        order by title;
```

where SQL is the libref for a remote SAS library that is accessed through a SAS/SHARE server or a SAS/CONNECT remote host, each row in the table EMPLOYEE must be returned to your local SAS session in order for the summary functions AVG() and FREQ() to be applied to them.

But, if you specify

```
select * from connection to remote
        (select emptitle as title,
             avg(empyears), freq(empnum)
             from sql.employee
             group by title
             order by title);
```

the query is passed through the SAS/SHARE server to the SAS SQL processor, which processes each row in the table and returns only the summary rows to your local SAS session.

You can also use RSPT to join remote data with local data. For example, if you specify

```
libname mylib 'c:\sales';

proc sql;
    connect to remote (server=mvs.shr1 dbms=db2
    dbmsarg=(ssid=db2p));

    select * from mylib.sales97,
```

```
                    connection to remote
                       (select qtr, division,sales, pct
                           from revenue.all97
                           where region = 'Southeast')
          where sales97.div = division;
```

the subquery against the DB2 data is sent through the SAS/SHARE server to the DB2 server and the rows for the divisions in the Southeast region are returned to your local SAS session, where they are joined with the corresponding rows from the local data set MYLIB.SALES97.

If your server is a SAS/CONNECT remote host, you can also use RSPT to send non-query SQL statements to a remote DBMS. For example,

```
proc sql;
   connect to remote (server=sunserv dbms=oracle);

   execute (delete from parts.inventory
           where part_bin_number = '093A6')
           by remote;
```

sends the SQL DELETE statement through the SAS/SHARE server to the remote ORACLE server.

SCL Programming Considerations

You can use SAS Component Language (SCL) with SAS/SHARE software to access data through a SAS/SHARE server. SCL has the ability to read and update SAS tables that are used concurrently by other clients or SCL applications. For complete information about SCL, see *SAS Component Language: Reference*.

A *concurrent SCL application* opens one SAS data table for update while other SAS operations (possibly in different SAS sessions) have the same data table open for update. These other opens for update can be done by other invocations of the first SCL application, by a different SAS application or SCL application, or even by a user who uses the FSEDIT or FSVIEW procedure on the table.

This section describes the following issues that you should consider when writing an SCL application that updates data concurrently:

- Locking rows in SAS tables
- Implications of row locking in SCL
- Programming in the FSEDIT and FSBROWSE procedures
- Locating and fetching control rows
- Unlocking a row.

Finally, see Appendix 4, "SAS Component Language Application," for an application that uses SAS tables that contain inventory and ordering information for each product in a store. The purpose of the application is to automate a system that develops orders and maintains the inventory list while sales representatives simultaneously write orders for products.

Locking Rows in SAS Tables

A row in a SAS table is locked implicitly when it is read by a SAS procedure, a DATA step, or an SCL application. A lock on a row is held until a different row is read or until the SCL application calls the UNLOCK function.

When a SAS table is opened for update, only one row can be locked at a time. When a SAS table is opened for update more than one time in the same SAS session or in different SAS sessions (through a server), a different row can be locked by each of the opens. For example, if two users are running an SCL application that calls the OPEN function to open a SAS table for update, row 7 can be locked under one of the opens while row 10 is locked under the other.

Implications of Row Locking in SCL

Row locking is a key consideration in concurrent SCL programming. After a lock on a row is released, your application can no longer be sure of the values that are contained in that row; another user might have already modified the values. Any data modifications that you make that are based on the old values may damage the data integrity of the system.

Therefore, you must never assume that the data values in a given row will not change in a shared table, even though only a very brief amount of time has elapsed between consecutive reads and locks of the row.

Row locking can give a programmer an important advantage. While an SCL application has a row locked, no other SAS operation (especially in another SAS session) can alter or delete that row. When each row in a SAS table can represent a specific instance of a resource that the application must govern, row locking provides a resource-specific, protected period of time in which the application can safely test and modify the state of the resource.

An example of a specific-resource instance is information about one of your customers or the number of items of a specific type that is currently in inventory. The sample SCL application in the appendix in this book applies locks to its inventory table to maintain the proper inventory count for each item, even if several sales representatives are simultaneously writing orders for those items.

Programming with the FSEDIT and the FSBROWSE Procedures

Unlike other SCL environments (such as SAS/AF software and the FSVIEW procedure), the FSEDIT and FSBROWSE procedures give the SCL programmer a number of labeled sections for structuring an SCL application. The sequence in which the FSEDIT and FSBROWSE procedures run some of these sections has several implications for concurrent SCL programming.

The INIT section is especially useful to applications that read and update shared data. The initial values of the columns in a row (as presently stored in the SAS table) can be preserved in SCL columns. Preservation of initial values in SCL columns is important for applications that update auxiliary tables that are based on the PROC FSEDIT user's modification or on the creation of a row in the *primary* table (that is, the table that is specified in the DATA= option in the PROC FSEDIT statement.) * These SCL applications typically need to perform the following tasks:

- determine if a modified window column actually contains a value different from its initial, validated value
- explicitly restore the initial values in case the user's modifications to the primary table's row are not allowed because they could not be validated
- modify the auxiliary tables because of the changes (from their initial values) made to the values in the primary table's row.

* In a concurrent SCL application, an *auxiliary table* is any table other than the one specified in the DATA= option of the PROC FSEDIT statement.

While the MAIN or TERM sections must validate the user's modifications to the primary table's row as well as update auxiliary tables, it is usually desirable that no row of an auxiliary table remain locked between executions of these sections. Such locks prevent other users or applications from modifying the row as long as the user is on the primary table's current row.

Programming with the Data Table and Data Form Classes

The Data Table and Data Form classes in SAS/AF FRAME entries allow you to specify an SCL entry to use for the model SCL. This SCL entry is separate from the frame's SCL entry. Model SCL is typically used to initialize computed columns and to perform error checking and data validation.

Like the FSEDIT procedure, the Data Table and the Data Form objects give the SCL programmer a number of labeled sections for structuring the order in which events will take place for each row in the table. These sections, which include INIT, MAIN, and TERM, operate in the same way as explained in "Programming with the FSEDIT and the FSBROWSE Procedures" on page 51.

If multiple instances of the Data Table or the Data Form objects are displayed within a single SAS/AF FRAME entry, the objects share data, then the model SCL for each data table or data form runs separately and the application developer must keep in mind whether a previous object has a lock on a row that the current object attempts to read or update. In addition, the frame SCL may also be operating on the shared data and timing within the frame could be critical. For more information about when SCL labels are run, see "Controlling Execution in the SCL" and "Summary of SCL Label Running" in the Data Set Data Model class under SAS/AF Component Reference in online help.

Locating and Fetching Control Rows

SCL provides a set of functions that are useful for locating and fetching the required auxiliary table observations in a data-concurrent SCL application. However, you should use caution with these functions in applications that access shared data. The return code, which is obtained directly from the called function or from the SYSRC function, must be checked to ensure that a lock was obtained on the observation or that an update was successful. The return value, which is generated by the macro invocation %SYSRC(_SWNOUPD), is generated when a fetch or update function fails to lock or update the observation because it is locked by another application.

The FETCHOBS table function is useful when the observation number can serve as the observation identifier. Remember that this function accepts a relative observation number by default that may or may not equate to the physical observation number. If you can delete observations in the auxiliary table, you probably want to use the ABS option of the FETCHOBS function for absolute observation numbering.

The LOCATEC and LOCATEN table functions may be useful for finding observations in small tables when the data can remain sorted by a unique identifier (column) and a binary search is specified. Beyond these limits, due to the overhead of searching with these SCL functions, you should use the WHERE and FETCH functions to find these observations. In a shared-data environment, each observation must be requested from the server and transmitted to the client's SAS session for the LOCATEC and LOCATEN functions to check.

The SYSRC function must be queried for warnings when the LOCATEC and LOCATEN functions find an observation because these functions return only a zero-positive return code for either condition: observation found or observation not found. The following SCL program example illustrates checking whether the located observation is locked by another task:

```
gotrec=locatec(data-set-id,var-num,search-string,
                sort-order);
if (gotrec<=0) then do;
   /* Handle observation not found */
 end;
else if (sysrc()=%sysrc(_swnoupd)) then do;
   /* Handle observation locked */
 end;
```

Note: The LOCATEC and LOCATEN functions cannot perform binary searches on compressed tables, SAS data views, or SAS data files with deleted observations. △

The more general and usually more efficient way to locate an observation is to use the WHERE function followed by a FETCH function call. The WHERE clause is evaluated in the server's SAS session, and only the observation that satisfies the WHERE clause is transmitted to the client's SAS session.

If the WHERE clause does not find the requested observation, the FETCH function returns a –1 return code indicating that the end of the table has been reached. If the WHERE clause is cleared by issuing a null WHERE function call, the next FETCH call that the application issues fetches the first observation in the table. The FETCH call, not the WHERE clause, locks the observation (if possible). Note that the WHERE function returns a harmless warning, %SYSRC(_SWWREP), when the WHERE clause is replaced.

The DATALISTC and DATALISTN selection-list functions help a client to select a valid observation. These functions actually fetch the entire selected observation into the Table Data Vector (DDV) and lock the observation (if possible). Because these functions do not return a system return code, the SYSRC function must be queried for warnings. The DATALISTC and DATALISTN functions may cause the entire SAS table to be read, where each observation read is transferred individually from the server to the client SAS session.

Unlocking an Observation

Besides releasing a lock on the current observation by reading another observation, an SCL application can also use the SCL function UNLOCK. UNLOCK leaves the read-pointer at its current position in the table and does not update the DDV.

Furthermore, the SCL OBSINFO function returns information about the primary table's current observation in an FSEDIT application. You can query whether the observation has been deleted, locked, or newly created. An observation does not attain deleted status until the client's DELETE command has run. Therefore, if you specified the CONTROL ENTER statement to force your MAIN section to run, the OBSINFO function will not return the deleted status when issued from the MAIN section (because the user's DELETE command has caused MAIN to be run.) However, it will return a deleted status when the MAIN statement or TERM section is run again.

SAS Data View Programming Considerations

This section describes SAS data views and how they are affected by the use of SAS/SHARE software. It contains a discussion of these topics:

- Defining SAS data views
- Interpreting SAS data views
- Deciding where to interpret SAS data views.

Defining SAS Data Views

Beginning with Version 6 of SAS software, a SAS data set can be of member type DATA (SAS data file) or type VIEW. A data set of type VIEW is called a *SAS data view*. It contains a definition or description of data that is stored elsewhere. Views can be created using the DATA step, the SQL procedure, or the ACCESS procedure, which is available through SAS/ACCESS software. In most SAS programs, whether the data comes from a SAS data view or a data file is unimportant.

Many SAS/ACCESS interface products, such as DB2 or ORACLE, enable you to update product data through a SAS/ACCESS view. However, for views that are interpreted in the server's session, whether you can update a view's underlying DBMS data depends on the particular SAS/ACCESS interface engine that you are using. For information about how to use SAS/ACCESS engines in a SAS/SHARE server session, see the SAS/ACCESS documentation.

Interpreting SAS Data Views

A SAS data view that is accessed through a server can be interpreted in either the server or client session. For complete information about view interpretation, see "Deciding Where to Interpret SAS Data Views" on page 55.

The client specifies where a SAS data view is interpreted by using the RMTVIEW= option to the LIBNAME statement. When the RMTVIEW= option is set to YES or the option is omitted, a data view is interpreted in the server session. When the option is set to NO, a data view is interpreted in the client session. For more information about the LIBNAME statement and its options, see Chapter 9, "The LIBNAME Statement," on page 103.

Interpreting a view consists of loading and calling the view engine to read the view's underlying data and present them as a SAS data set. When a view is interpreted in the client SAS session, the view engine is loaded and called by the client SAS session to read and present the underlying data. When a view is interpreted in the server session, the view engine is loaded and called by the server to read and present the underlying data.

Wherever a SAS data view is interpreted, the underlying data must be accessible to its view engine. The definition of accessible varies according to the type of view:

- PROC SQL
- DATA step
- SAS/ACCESS.

For PROC SQL views, *accessible* means that all librefs that are used in the view must be pre-defined in the SAS session in which the view is interpreted or included in the USING clause of the query that is stored in the view. Such a libref can be associated with a SAS data library that is accessed through a server, or it can be associated with a data library that is accessed locally. You do not have to specify a libref in a PROC SQL view for data sets in the same data library as the view itself.

For DATA step views, *accessible* means that any external file (or files) must be available and that any filerefs and librefs that are stored in the view must be defined in the SAS session in which the view is interpreted.

For PROC ACCESS views, *accessible* means that the interface product and its data, as well as the SAS/ACCESS interface view engine, must be available to the SAS session in which the view is interpreted.

Deciding Where to Interpret SAS Data Views

Deciding where SAS data views should be interpreted in a shared environment depends on the kind of view, on how you plan to use the view's data, and on certain site considerations.

The first consideration is whether the view's underlying data is accessible. If the data is accessible only from one of the sessions, the view must be interpreted there.

If the data is accessible from both the client session and the server session, performance is the next factor to consider. If interpreting the view requires the SAS session to read a large number of rows in order to select a small subset, having the SAS/SHARE server interpret the view greatly reduces the number of records that are transmitted to the client session. This method reduces network load and may be faster than having the client session interpret the view. However, this might put an undesirable processing load on the server (particularly if joins are involved) and adversely affect server performance for other clients. If the view selects most of the input rows or if the selection criteria are processed by a DBMS server, interpreting the view in the client session is probably preferred.

Using SAS Catalog Entries in Programs

You cannot access the following types of catalog entries through a SAS/SHARE server:

AFCBT
AFGO
AFPGM
AMETHOD
ENGINE
GEDIT
GLOBAL
GOPTIONS
MODEL
MSYMTAB
OLDMACRO
PROFILE
STATGRAP
TITLE
WSAVE

Many catalog entries of these types are stored in the library SASUSER. Because they cannot be accessed through a server, do not store them in a SAS library that is accessed through a server.

Full access to all other types of entries is supported through a SAS/SHARE server.

You can obtain exclusive access to a catalog or to individual entries (other than the types provided in the preceding list) by locking the catalog or catalog entries with the LOCK statement or the LOCK command, as described in Chapter 5, "Locking SAS Data Objects," on page 59 and Chapter 11, "The LOCK Statement and the LOCK Command," on page 113. You can also lock a catalog or catalog entry with the SCL function LOCK as described in *SAS Component Language: Reference*.

SAS/CONNECT Programming Considerations

You can use SAS/SHARE with SAS/CONNECT to extend your access to SAS files and hardware resources. With SAS/CONNECT, a user who is running a SAS session on

a *local host* (for example, UNIX or OS/2) can access files on one or more *remote hosts*, which can be any other operating system environment that supports a SAS session and SAS/SHARE.

The two major functions of SAS/CONNECT software are data transfer and remote processing. *Data transfer* enables a user to move data across hardware platforms as well as across releases of SAS software. *Remote processing* enables a user to send SAS statements to the remote SAS session for processing; all output and messages generated from that session are directed back to the local SAS session for display. However, remote processing has limitations. For example, you cannot perform interactive tasks, such as a PROC FSEDIT session, while using SAS/CONNECT.

You must use SAS/SHARE software with SAS/CONNECT software if SAS files that are used by the remote SAS session are accessed through a SAS/SHARE server. SAS/CONNECT software provides the connection between the hosts and the ability to run SAS statements in the remote environment; SAS/SHARE software allows concurrent use of the shared data.

The following example illustrates the necessity for SAS/SHARE in a SAS/CONNECT session. Say that Sasha uses the same data accessed by John and Maria in the example in Chapter 3. Sasha needs to create a report from DATALIB.FUEL. He typically uses SAS/CONNECT to establish the connection from his local workstation to the host system that clients John and Maria log onto directly, which is the host that the SAS/SHARE server runs on. Because the library that contains the data that he needs is accessed through a server, Sasha must also use SAS/SHARE to access that data from his remote SAS session. He connects to the remote host and uses the RSUBMIT command to remote-submit the following SAS statements to the remote SAS session:

```
options comamid=tcp;
%libdef(datalib);
proc print data=datalib.fuel;
   where area='TEX3' and profits<=0;
   title 'Losses in Texas, Area 3';
run;
```

In this example, Sasha uses the OPTIONS statement to re-set the communications access method to TCP/IP, which is used by the REMOTE engine to communicate with the server.

The LIBDEF macro generates a LIBNAME statement to access DATALIB through the same server that John and Maria use. Sasha's remote session connects to the server and then executes the PRINT procedure to produce his report. The report is displayed in his SAS session on the local workstation. Except for certain interactive limitations that are imposed by SAS/CONNECT, Sasha can remote-submit the same SAS program steps to read or write the data in DATALIB that other users use when logging onto the remote host directly.

CAUTION:

Do not remote-submit the SERVER procedure when using SAS/CONNECT software. Although the server runs in the remote session, the local session waits until the remote-submitted statements have been run, so that the local session is occupied until the server terminates. △

As a further example of using SAS/CONNECT and SAS/SHARE together, consider Sasha's next project. He needs to update a set of production costs that are maintained in SQL/DS tables by another division. The SAS/ACCESS views for these SQL/DS tables are stored in a data library on a CMS host that is accessible through a server. These views enable Sasha and other users on the remote host to process the SQL/DS tables in their SAS sessions. Because the views are accessed through a server, users can create new SAS/ACCESS views while the existing views are in use.

Sasha remote-submits a LIBDEF macro to access the library that contains the SAS/ACCESS views for the SQL/DS tables. He then uses a MODIFY statement in a DATA step to update the PRODCOST view's underlying SQL/DS table. For more information about the MODIFY statement, see "DATA Step Processing" on page 45.

```
%libdef(sqldsvws);
data sqldsvws.prodcost;
   modify sqldsvws.prodcost;
   if part='0420' and units<=10 then
      do;
         if _iorc_=0 then remove;
         else put 'Could not remove cost record '
            recid;
      end;
run;
```

Note that Sasha is able to update the view's underlying SQL/DS table because the LIBNAME statement that is generated by the LIBDEF macro specifies NO as the value of the RMTVIEW= option. Setting RMTVIEW to NO causes the SAS/ACCESS view to be interpreted in Sasha's remote SAS session rather than in the server session. For more information about the RMTVIEW= option and interpreting views, see "Interpreting SAS Data Views" on page 54.

For more information about using SAS/CONNECT software, see the *SAS/CONNECT User's Guide*.

Tuning Tips for Client-Server Applications

For information about programming techniques and SAS system option value adjustments that you may use to improve the performance of your application, see Appendix 3, "Tuning Tips for Applications That Use SAS/SHARE Software," on page 177.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/SHARE User's Guide, Version 8*, Cary, NC: SAS Institute Inc., 1999. pp. 247.

SAS/SHARE User's Guide, Version 8

ISBN 1–58025–478–0

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, September 1999