CHAPTER

11

Using SAS Tables

Introduction

SCL provides a group of features that can read or manipulate data stored in SAS tables. For example, you may want an SCL program to update one or more SAS tables, based on user transactions from a single user interface. For a data entry and retrieval

system, you may want to use a secondary table to supplement the primary table. You might use the secondary table as a lookup table for sophisticated error checking and field validation. In addition, you may want to manipulate SAS tables to perform tasks like the following:

- displaying table values in a window
- creating a new table
- copying, renaming, sorting, or deleting a table
- indexing a SAS table.

Many functions that perform SAS table operations return a SAS software return code, called sysrc. Chapter 14, "SAS System Return Codes," on page 227 contains a list of return codes with a section for operations that are most commonly performed on SAS tables. You can check for these codes to write sophisticated error checking for your SCL programs.

The following sections describe the tasks that SCL programs can perform on SAS tables, along with summary information about the SCL function or routine to use to perform that task. These functions and routines are described in Chapter 16, "SAS Component Language Dictionary," on page 249.

Accessing SAS Tables

Before an SCL program can access the values in a SAS table, a communication link must be established between SAS software, the SAS tables, and the SCL program. You link SAS software to the tables by assigning librefs to the data libraries in which the SAS tables are stored. You complete the communication by linking the SAS tables and the SCL program, using the OPEN function to open the SAS tables. (Some SCL routines, such as CALL FSEDIT and CALL FSVIEW automatically open the SAS table that they are displaying. Therefore, the OPEN function is not needed to open the specified table.)

Assigning Librefs

SCL provides the LIBNAME function for assigning a libref in an SCL program. You can also assign librefs outside an SCL program that works with SAS tables by putting the appropriate LIBNAME statement in the application's start-up file, the autoexec file. For more information about assigning librefs outside an SCL program, see the SAS software documentation for your host operating system.

If you have SAS/SHARE software or SAS/CONNECT software installed at your location, you can also use Remote Library Services (RLS) to assign librefs. RLS gives your SCL applications "read" or "write" access to SAS tables, views and catalogs across hardware platforms and SAS releases. Once an RLS libref is established, the RLS functionality is transparent to SAS tables and views in SCL programs. Catalog compatibility across platforms is architecture dependent. For further information, see *SAS/SHARE User's Guide.*

Opening SAS Tables

To open a SAS table, use the OPEN function. Opening a SAS table provides the gateway through which an SCL program and a SAS table can interact. This process

does not actually access the information in a SAS table, but it makes the table available for the program's use. To access the data in the SAS table, the program must perform "read" operations on the table. When you open a SAS table, the following actions take place:

- The SAS table data vector (TDV) for the table is created to store copies of the table's column values.
- A unique numeric identifier is assigned to the SAS table. This identifier is used by other functions that manipulate data.
- An access control level is assigned to the SAS table. This control level determines the level of access to the SAS table that is permitted to other users or applications that try to use the SAS table at the same time.

The identifier number identifies the table to the application, and you pass it to other SCL functions that manipulate the SAS table. Because this number is unique for each table that is currently open, it is useful for tracking multiple SAS tables that are open at the same time.

Note: If for some reason a SAS table cannot be opened, the OPEN function returns a value of 0 for the table identifier. Therefore, to determine whether a SAS table has been opened successfully, you should test the value of the return code for the OPEN function in your SCL program. Doing this ensures that you don't cause a program to halt by passing a 0 to another function that uses that SAS table identifier. To determine why the table could not be opened, use the SYSMSG function to retrieve the message that is associated with the return code. △

Number of Open SAS Tables Allowed

An application can have a maximum of 999 SAS tables open simultaneously. However, your operating system may impose other limits. For details, see the documentation provided by the vendor for your operating system.

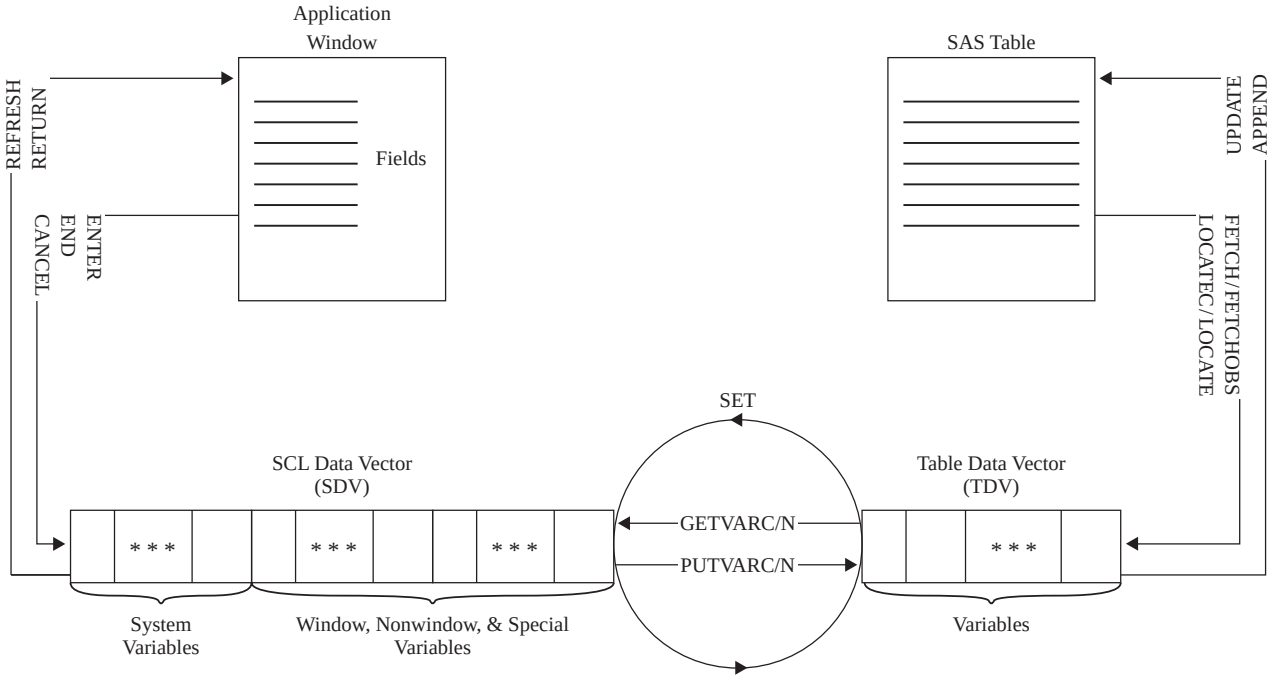
Although SCL permits you to have a large number of tables open simultaneously, be aware that memory is allocated for each SAS table from the time the SAS table is opened until it is closed. Therefore, try to minimize the number of tables that are open at the same time, and close them as soon as a program finishes with them for that session.

SAS Table and SCL Data Vectors

When an application opens a SAS table, its TDV is empty. However, to enable the program to work with the SAS table columns, SCL provides functions for copying table rows one at a time from the SAS table to the TDV. Once column values for a row are in the TDV, you can copy these values into the SCL data vector (SDV), and the application can manipulate the column values.

Before you can display or manipulate the values of SAS table columns, those columns must be linked to SCL variables through the TDV and the SDV. Special SCL functions and storage locations facilitate the transfer of values between SAS table columns and SCL variables. Figure 11.1 on page 168 illustrates the SDV and TDV that are created for an application that opens a SAS table. This figure shows the paths that rows take when they are read from the table, displayed in the window, processed, and then returned to the table.

Figure 11.1 Path of Data in SAS Table "Read" and "Write" Operations



Two steps are required in order to transfer data from an open SAS table to an SCL program:

1. The values of the columns in a row in the open SAS table are copied into the TDV.
2. The values of the columns in the TDV are copied to the SDV, which contains all of the SCL variables (window variables, nonwindow variables, system variables, and so on). The transfer of data from the TDV to the SDV can be either automatic (when the SET routine is used) or under program control (when the GETVARC or GETVARN functions are used).

Once the values are in the SDV, you can manipulate them by using SCL statements. Two steps are also required in order to transfer data from an SCL program to an open SAS table:

1. The column values in the SDV are transferred to the TDV. The transfer of data from the SDV to the TDV can be either automatic (when the SET routine is used) or under program control (when PUTVARC or PUTVARN is used).
2. The values in the TDV are written to the columns in a row in the open table.

Access Control Levels

When a SAS table is opened, SAS software determines the control level for the table. The control level determines the extent to which access to the table is restricted. For example, an application may be able to gain exclusive update access to the entire SAS table, or it may be able to gain exclusive update access to only the row that is currently in use. In either case, there are ramifications for any users or applications that need to access the SAS table at the same time. You can open a SAS table with one of the following control levels:

RECORD
: provides exclusive update access only to the SAS table row that is currently in the TDV (as with the FETCH and FETCHOBS functions). With this control level, the

same user can open the same SAS table multiple times (multiple concurrent access). In addition, if SAS/SHARE software is used, then multiple users can open the same SAS table simultaneously for browsing or for editing. For more information, see *SAS/SHARE User's Guide*.

MEMBER
provides exclusive update access to an entire SAS table. While this control level is in effect, no other user can open the table, and the same user cannot open the table multiple times.

Specifying a Control Level

When you use the OPEN function to open a SAS data set in UPDATE mode, by default the table is opened with RECORD-level control. However, in SCL you can use the OPEN function with the SAS data set option CNTLLEV= to set the control level when a SAS table opens. See "OPEN" on page 576 for more information.

Reading SAS Tables

You may want to use an SCL program to manipulate column values from SAS tables. For example, you may want to do one or more of the following:

- display data values in a window
- use the values in arithmetic calculations
- determine data values before taking certain actions.

Before a program can manipulate the data, it must read the data from the table. After column values are changed, the program can update the values of columns in the table. In addition to updating existing column values, programs also can add new rows or delete obsolete rows.

After a SAS table is open, you can access any column value for any row in the SAS table. The first step in accessing the data involves reading (or copying) a row from the SAS table to the TDV—for example, by using the FETCH function. By default, the FETCH function starts with the first row in the SAS table and reads the next row from the SAS table each time it executes.

Linking SAS Table Columns And SCL Variables

The next step in accessing the data is to link the SAS table columns in the TDV with the SCL window variables and nonwindow variables in the SDV. The function that you use depends on whether the SCL variables and SAS table columns have the same name and type. If an application has some SCL variables that match SAS table columns and others that do not, then you can use a combination of these techniques.

Matched Column and Variable Names

If columns of a SAS table and SCL variables have the same names and types, then you can use the SET routine to link all of them automatically with a single program statement. The SET routine is typically invoked immediately following the OPEN function.

Note: If you use the SET routine and then also use the PUTVARC or PUTVARN routine for an SCL variable that has a matching SAS table column, the SET routine

overrides the PUTVARC or PUTVARN routine. Doing this is inefficient because duplicate actions are performed. △

Unmatched Column and Variable Names

When the SCL variables do not have the same names or types as SAS table columns, you must use a GETVARC or GETVARN statement (for character and numeric values, respectively) for each unmatched column to link them from the TDV to the SDV. Once the columns have been manipulated, use an individual PUTVARC or PUTVARN routine to link each one from the SDV back to the TDV.

Note: The GETVARC and GETVARN functions establish only a temporary link between a SAS table column and an SCL variable. When the statement executes, the columns are linked. After the statement executes, the link is terminated. Therefore, you must use the GETVARC or GETVARN function one time for each SAS table column that you want to link. This is different from the SET routine, which establishes a permanent link between any matching SAS table and SCL variables until the open SAS table is closed. △

Determining a Column's Position in a SAS Table

Some functions, such as GETVARC, GETVARN, PUTVARC and PUTVARN, require the position of a column in the SAS table row. Use the VARNUM function to determine the position, and then use the position repeatedly throughout your program. The following example uses the VARNUM function to determine the position of several columns. After the column positions have been determined, the program links to a labeled section called GETVALUE to determine the column values.

```
INIT:
   control enter;
   houses=open('sasuser.houses','u');
   if (houses=0) then _msg_=sysmsg();
   else
      do;
         vtype=varnum(houses,'style');
         vsize=varnum(houses,'sqfeet');
         vbedrms=varnum(houses,'bedrooms');
         vbathrms=varnum(houses,'baths');
         vaddress=varnum(houses,'street');
         vcost=varnum(houses,'price');
         link getvalue;
      end;
return;

MAIN:
   ...more SCL statements...
return;

TERM:
   if (houses>0) then rc=close(houses);
return;

GETVALUE:
   rc=fetch(houses);
   type=getvarc(houses,vtype);
```

```
   size=getvarn(houses,vsize);
   bedrms=getvarn(houses,vbedrms);
   bathrms=getvarn(houses,vbathrms);
   address=getvarc(houses,vaddress);
   cost=getvarn(houses,vcost);
return;
```

Using Table-Lookup Techniques

Table lookup, the process of looking up data in a data structure, has several useful applications for data entry applications. For example, you may want to display certain information in a window based on a value that a user has entered. If this information is stored in another SAS table, then you can use table-lookup techniques to read and display this information. In addition, you can use table lookup to perform field validation by ensuring that a value entered by a user is a value that is contained in a specified SAS table.

To validate a field value, you can use the LOCATEC, LOCATEN, or WHERE function to search a secondary SAS table for a specific character or numeric value that has been entered by a user. For example, you might want to make sure that users enter names that exist in another SAS table. You also can use these techniques to display text from a secondary SAS table, based on values that users enter in the fields. For example, when a user enters a valid name in the **`Employee Name`** field, you can look up the associated sales region and sales to date in the secondary SAS table and then display this information in the window.

Controlling Access to SAS Table Rows

For many applications, you may want an SCL program to read from a SAS table only the rows that meet a set of search conditions. For example, if you have a SAS table that contains sales records, you may want to read just the subset of records for which the sales are greater than $300,000 but less than $600,000. To do this, you can use WHERE clause processing, which is a set of conditions that rows must meet in order to be processed. In WHERE clause processing, you can use either permanent or temporary WHERE clauses.

Permanently Subsetting Data

A permanent WHERE clause applies a set of search conditions that remain in effect until the SAS table is closed. You might use a permanent WHERE clause to improve the efficiency of a program by reading only a subset of the rows in a SAS table. You might also want to use a permanent WHERE clause in applications when you want to limit the SAS table rows that are accessible, or visible, to users. For example, if you are working with a large SAS table, users may not need access to all the rows to use your application. Or, for security reasons, you may want to restrict access to a set of rows that meet certain conditions.

SCL provides several features that enable you to subset a SAS table based on specified search conditions. To apply a permanent WHERE clause to a SAS table, you can use the SAS data set option WHERE= with the OPEN function. For example, the following WHERE clause selects only the records for which the sales are greater than $300,000 but less than $600,000:

```
   /* Open the SAS table and display a  */
   /* subset of the SAS table rows      */
salesid=open
   ("sample.testdata(where=((sales > 300000)"||
                 "and (sales < 600000)))",'i');
```

You can also use the WHERE= option in SCL with the FSEDIT and FSVIEW routines.

Temporarily Subsetting Data

In addition to restricting access to SAS table rows, you may want to enable users to subset the accessible records even further. In this case, you can use the WHERE function to apply a temporary WHERE clause. A temporary WHERE clause applies a set of search conditions that can be modified or canceled by subsequent SCL statements. For example, you could apply a temporary WHERE clause like the following:

```
rc=where(dsid,'SSN='||ssn);
```

When a SAS table is indexed, you can use the SETKEY function for subsetting. For example, if a SAS table is indexed on the column SSN, you could use:

```
rc=setkey(dsid,'SSN','eq');
```

Searching with WHERE versus LOCATEC or LOCATEN

You can search efficiently with the WHERE function if you are working with a large SAS table that is indexed by the column or columns for which you are searching. It is also appropriate to use the WHERE function when you are using an expression that involves several columns to locate rows.

However, you can use LOCATEC or LOCATEN to find a row when one or more of the following conditions are met:

- The SAS table is small.
- You are searching for one row that meets a single search condition (for example, the row that contains a particular name).
- You are looking for one row that meets a single search condition in a large SAS table, if the SAS table is sorted by the column for which you are searching, and if you are using the more efficient binary search. See the following section for more information.

Searching Efficiently

By default, LOCATEC and LOCATEN search a SAS table sequentially. However, a sequential search is not always the most efficient way to locate a particular row, especially if your SAS table has been sorted. If a SAS table has already been sorted by the column for which you want to search, you can specify a faster, more efficient binary search. For a binary search, use an additional optional argument with LOCATEC or LOCATEN to specify the order in which the SAS table has been sorted (A for ascending order or D for descending order). For example, assuming that the SAS table MYSCHOOL.CLASS has been sorted in ascending order by NAME, you can use the following statements to perform a binary search:

```
dsid=open('myschool.class');
vnum=varnum(dsid,'name');
```

```
sname='Gail';
val=locatec(dsid,vnum,sname,'a');
```

Undoing WHERE Clauses

WHERE clauses impose certain restrictions on other SCL functions that manipulate data. Therefore, in some cases, you may need to undo a WHERE clause in an SCL program before using other functions. When you specify a WHERE clause, the WHERE conditions replace the conditions that were specified in the previous WHERE clause. However, you can augment a WHERE condition with the ALSO keyword. For example, the following WHERE clause adds the condition of "age greater than 15" to an existing WHERE clause:

```
rc=where(dsid,'also age > 15');
```

To undo the condition that was added by the ALSO keyword, you could use the following statement:

```
rc=where(dsid,'undo');
```

To undo (or delete) a *temporary* WHERE clause, use the WHERE function and specify only the SAS table identifier argument. This process undoes all temporary WHERE clauses that are currently in effect.

Changing the Sequence of Reading Rows

When an application displays a subset of a SAS table, you may want to let users display and scroll through all rows that meet the search conditions. To do this, you can use a set of SCL functions that reread table rows. For example, when a program displays the first row that meets the conditions, SCL provides functions that you can use to mark the row. Then a user can continue to search the rest of the SAS table for any other rows that meet the search conditions, counting them along the way. After finding the last row that meets the search conditions, the user can redisplay the first row in the subset (the row that was marked earlier). The following sequence of steps implements this technique:

1. Use the NOTE function to mark a row in the subset for later reading.
2. Use the POINT function to return to the marked row after you have located all rows that meet the search conditions.
3. Use the DROPNOTE function to delete the NOTE marker and free the memory used to store the note after the program finishes using the noted row.

Updating SAS Tables

When a table row is read, its data follow a path from the SAS table through the TDV to the SDV, where finally they can be manipulated. After the data is manipulated, it must follow the reverse path from the SDV through the TDV back to the SAS table. If you use the SET routine to link the values from the TDV to the SDV, then any changed values are automatically linked from the SDV back to the TDV. If you do not use SET, then you must explicitly copy the value of each variable to the TDV. In either case, you use the UPDATE function to copy the values from the TDV to the SAS table.

Appending Rows

To add new rows to a SAS table rather than updating the existing rows, use the APPEND function. If the SCL variables have the same name and type as the SAS table columns and you use the SET routine to link them, then using the APPEND function is straightforward, and the values are automatically written from the TDV to the SAS table.

Note: If the program does not use the SET routine, or if the APPEND function is used with the NOSET option, a blank row is appended to the SAS table. This is a useful technique for appending rows when the SCL program or the window variables do not match the SAS table columns. For example, when the SET routine is not used, you would use a sequence of statements like those below to append a blank row and then update it with values. △

```
rc=append(dsid);
...PUTVARC or PUTVARN program statement(s)...
rc=update(dsid);
```

Deleting Rows

To delete rows from a SAS table, use the DELOBS function. In order to use this function, the SAS table must be open in UPDATE mode. The DELOBS function performs the following tasks:

- marks the row for deletion from the SAS table. However, the row is still physically in the SAS table.
- prevents any additional editing of the row. Once a row has been marked for deletion, it cannot be read.

Remaining Rows Not Renumbered

Although deleted rows are no longer accessible, all other rows in the SAS table retain their original physical row numbers. Therefore, it is important to remember that a row's physical number may not always coincide with its relative position in the SAS table. For example, the FETCHOBS function treats a row value as a relative row number. If row 2 is marked for deletion and you use FETCHOBS to read the third row, FETCHOBS reads the third *non-deleted* row—in this case, row 4. However, you can use FETCHOBS with the ABS option to count deleted rows.

Non-deleted rows are intentionally not renumbered so that you can continue to use row numbers as pointers. This is important when you are using the FSEDIT procedure or subsequent SAS statements that directly access table rows by number, such as the POINT= option in a SAS language SET statement.

You can control row renumbering if necessary. See the next section for details.

Renumbering Rows

To renumber accessible SAS table rows, an SCL program must use one of the following techniques to process the SAS table:

- Sort the table, using either the SORT function in SCL or the SORT procedure. If the SAS table is already in sorted order, then you must use the FORCE option.

Note: The SORT function and PROC SORT do not sort and replace an indexed SAS table unless you specify the FORCE option, because sorting destroys indexes for a SAS table. △

- Copy the table, using either the COPY function in SCL or the COPY procedure. In this case, the input and output tables must be different. The output table is the only one that is renumbered.
- Read the remaining data table rows, using the SAS language SET statement in a DATA step (not the SCL SET statement), and write these rows to a data table. To avoid exiting from SCL, you can use a submit block. For example:

```
houseid=open('sasuser.houses','u');
   ...SCL statements that read rows and delete rows...
submit continue;
   data sasuser.houses;
      set sasuser.houses;
   run;
endsubmit;
```

Closing SAS Tables

After an SCL program has finished using a SAS table, the program should close the table with the CLOSE function at the appropriate point in your program. If a SAS table is still open when an application ends, SAS software closes it automatically and displays a warning message. In general, the position of the CLOSE function should correspond to the position of the OPEN function, as follows:

- If the OPEN function is in the initialization section, then put the CLOSE function in the termination section.
- If the OPEN function is in MAIN, then put the CLOSE function in MAIN.

Note: If you're designing an application system in which more than one program uses a particular SAS table, and if the identifier for this table can be passed to subsequent programs, then close the SAS table in the termination section of the program that uses the table last. △

Determining Attributes of SAS Tables and Columns

SCL provides features for determining characteristics (or attributes) of the SAS table or columns with which a program is working. For example, one approach is to open a table, determine how many columns are in the table, and then set up a program loop that executes once for each column. The loop can query the attributes of each column. To do this, the program needs to determine how many columns are in the SAS table, as well as the name, type, length, format, informat, label, and position of each column.

Querying Attributes of SAS Tables

SAS tables have a variety of numeric and character attributes associated with them. These attributes can provide some basic information to your SCL program. For

example, to determine the number of columns in an existing SAS table, use the NVARS argument with the ATTRN function. For a list of other table attributes and how to retrieve them, see “ATTRC and ATTRN” on page 258.

Querying Attributes of SAS Table Columns

Columns in a SAS table also have several attributes that your program may need to query. Here is a list of column attributes and the SCL functions that you can use to retrieve those attributes:

name	VARNAME function
number	VARNUM function
data type	VARTYPE function
length	VARLEN function
label	VARLABLE function
format	VARFMT function
informat	VARINFMT function.

Defining New Columns

After determining the name, type, length, label, format, and informat of each column, you can add a new column that has these attributes to the column list for a new SAS table. To do this, first use the OPEN function with the **N** argument (for NEW mode), and then use the NEWVAR function.

CAUTION:

Your program should check to see whether the SAS table exists before opening it in NEW mode. When used with the **N** argument (for NEW mode), the OPEN function replaces an existing SAS table that has the same name. If you do not want to delete an existing SAS table by opening it in NEW mode, then use the EXIST function to confirm that the table does not exist before using OPEN to create a new SAS table. △

Performing Other SAS Table Operations

There are other SCL functions that you can use to perform operations on SAS tables. The tasks that you can perform, along with the function to use, are as follows:

- To copy a table, use the COPY function. By default, if the target file already exists, the COPY function replaces that file without warning. To avoid unintentionally overwriting existing files, your program should use the EXIST function to determine whether the target file exists before executing the COPY function. (You can use COPY with a WHERE clause to create a new table that contains a subset of the rows in the original table.)
- To create a new table, use the OPEN function with the **N** option. (The table must be closed and then reopened in UPDATE mode if the program will update it). Then, use NEWVAR to create columns.
- To enable users to create a new table interactively, use the NEW function.
- To enable users to create a new table interactively from an external file, use the IMPORT function or the IMPORT wizard.

- To delete a table, use the DELETE function. (The table must be closed).
- To rename a table, use the RENAME function. (The table must be closed.)
- To sort a table, use the SORT function. (The table must be open in UPDATE mode.)

Preserving the Integrity of Data

SCL provides a group of functions that specify and enforce integrity constraints for SAS tables. Integrity constraints preserve the consistency and correctness of stored data, and they are automatically enforced for each addition, update, and deletion activity for a SAS table to which the constraints have been assigned. For such a table, value changes must satisfy the conditions that have been specified with constraints.

There are two basic types of integrity constraints: general constraints and referential constraints. The following list shows the specific types of integrity restraints that you can apply through SCL. The first four items are general constraints, which control values in a single SAS table. The last item is a referential constraint, which establishes a parent-child relationship between columns in two or more SAS tables.

- A column can contain only non-null values.
- A column can contain only values that fall within a specific set, range, or list of values, or that duplicate a value in another column in the same row.
- A column can contain only values that are unique.
- A column that is a primary key can contain only values that are unique and that are not missing values.
- A column that is a foreign key (the child) can contain only values that are present in the associated primary key (the parent) or null values. A column that is a primary key can contain only values that cannot be deleted or changed unless the same deletions or changes have been made in values of the associated foreign key. Values of a foreign key can be set to null, but values cannot be added unless they also exist in the associated primary key.

SCL provides the following functions for creating and enforcing integrity constraints:

ICCREATE
creates and specifies integrity constraints for a SAS table.

ICDELETE
drops an integrity constraint.

ICTYPE
returns the type of constraint that is assigned to a SAS table.

ICVALUE
returns the varlist or WHERE clause that is associated with an integrity constraint.

For more information about integrity constrains, see *SAS/SHARE User's Guide*.

Manipulating SAS Table Indexes

When you develop an application that creates a SAS table, you may want to give users the option of creating an index for the table. An index, which provides fast access to rows, is an auxiliary data structure that specifies the location of rows, based on the values of one or more columns, known as key columns. Both compressed and

uncompressed SAS tables can be indexed by one or more columns to aid in the subsetting, grouping, or joining of rows. SAS table indexes are particularly useful for optimizing WHERE clause processing.

SCL provides a set of functions for creating and manipulating SAS table indexes. However, SCL functions are just one way of building and querying SAS table indexes. Other ways include:

- the DATASETS procedure in base SAS software
- the INDEX= option (when you are creating a SAS table)
- the SQL procedure in base SAS software.

There are two types of indexes: simple indexes and composite indexes. A simple index is an index on a single column, and a composite index is an index on more than one column. A SAS table can have multiple simple indexes, multiple composite indexes, or a combination of simple and composite indexes.

SCL provides the following functions for manipulating indexes:

ICREATE
creates an index for SAS tables that are open in UTILITY mode.

IVARLIST
returns a list of one or more columns that have been indexed for the specified key in the table.

ISINDEX
returns the type of index for a column in a SAS table, as follows:

BOTH	The column is a member of both simple and composite indexes.
COMP	The column is a member of a composite index.
REG	The column is a member of a regular (simple) index.
(blank)	No index has been created for the specified column.

IOPTION
returns a character string that consists of the options for the specified key and index columns. The options are separated by blanks.

IDELETE
deletes an index for a SAS table that is open in UTILITY mode. You can delete an index when a program finishes with it, or if you find that the index is not operating efficiently. Keep in mind that indexes are not always advantageous. Sometimes the costs outweigh the savings. For a detailed discussion of when to use indexes, see the information about SAS files in *SAS Language Reference: Concepts*.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS® Component Language: Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999.

SAS® Component Language: Reference, Version 8

ISBN 1–58025–495–0

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

1st printing, October 1999