Sometimes You Get What You Want: SAS® I/O Enhancements for Version 7

Steve Beatrous and Billy Clifford, SAS Institute Inc.

Abstract

This paper presents a high level overview of new database features added to Version 7. Some of the features presented in this overview have been top-vote getters when they appeared on the SASWare Ballot (such as long mixed-case column names and versioning). Some of the new features were added to make the SAS System more of a DBMS (such as integrity constraints).

Concatenation

SAS libraries and SAS catalogs may be logically combined through library and catalog concatenation. Concatenation provides a tool for establishing a search list for locating the information contained in a SAS library or catalog.

Library Concatenation

Concatenation allows you to reference two or more SAS libraries with a single libref. In Version 6, some platforms, such as Windows and HP-UX, supported a limited form of library concatenation. The limited support of V6 had several problems:

- It allowed only one form of the concatenation syntax -all the levels were specified as quoted physical names.
- It was not portable; SAS programs that used concatenation could not be easily moved from one platform to another.
- Concatenation was limited to the base engine.

In Version 7, all platforms support SAS library concatenation. The libraries to be combined may be specified as librefs or as quoted physical names. The following are valid library concatenations:

```
libname foo (’path1’ ’path2’ ’path3’);
libname foo (A B C);
libname foo (’path’ A B);
libname bar (foo C D);
```

The quoted names in the above examples are physical library names. The names that are not quoted are previously assigned SAS librefs.

Library concatenation allows you to combine libraries that are processed by different engines. For example, suppose there are some files in a Version 6 library and some other files in a Version 7 library. Further, suppose that the application needs to process the collection of files in both libraries. The following syntax can be used to establish a single libref ("MYLIB") that combines the Version 6 and Version 7 libraries:

- Libname v6 ’path-to-v6-library’;
- Libname v7 ’path-to-v7-library’;
- Libname mylib (v7 v6);

The preceding example allows you to leave some files used by an application in Version 6 format while converting others to a Version 7 format. (Note that converting a file from Version 6 format to Version 7 format is as simple as running a proc copy from a Version 6 liberf to a Version 7 libref.) Someone who wants to evolve his or her files and applications from Version 6 to Version 7 uses this construct.

SAS Catalog Concatenation

Catalog concatenation allows the combination of two or more catalogs into a single logical catalog. Version 7 offers two forms of catalog concatenation: implicit and explicit.

Implicit catalog concatenation results from a concatenation of libraries through a LIBNAME statement (see "Library Concatenation" above). In implicit concatenation, all catalogs in the concatenation with the same catalog name will be logically combined.

Explicit catalog concatenation is a concatenation specified by the global CATNAME statement. In explicit concatenation, catalogs in the concatenation may have different member (catalog) names. An explicit concatenation establishes a logical catalog name that may be used in any context accepting a physical catalog name.

The syntax of the CATNAME statement is:

```
CATNAME clib.cmem (lib1.cat1 <(ACCESS =
READONLY)>lib2.cat2 <(ACCESS = READONLY)>
...);
```

In the above example, you are establishing a logical catalog named CLIB.CMEM. It is assumed that there is a libref defined named CLIB. It is also assumed that there is not an existing physical catalog named CLIB.CMEM.CATALOG.

The CATNAME statement above will establish a logical in-memory catalog that is named CLIB.CMEM. The logical catalog name may be used in any context that accepts a physical catalog name. The in-memory logical catalog is a snapshot of the catalogs listed in the parentheses.

The ACCESS option allows you to restrict which levels of the concatenation may be written to.

The following CATNAME commands only apply to explicitly defined catalog concatenations:

```
CATNAME clibname.cmemname CLEAR;
```

Clears the concatenation definition for CLIBNAME.CMEMNAME.

```
CATNAME _ALL_ CLEAR;
```

Clears all currently defined catalog concatenations.

```
CATNAME clibname.cmemname LIST;
```

Lists the members of the concatenation CLIBNAME.CMEMNAME.

```
CATNAME _ALL_ LIST;
```

Lists all currently defined catalog concatenations.

Implicit and explicit catalog concatenations allow combinations of two or more catalogs into a logical unit. The rules for navigating through the concatenation are the same for implicit and explicit concatenations.

The term catref will be used here to refer to a catalog name. A catalog name consists of a libref and a SAS file name separated by a period. For example the catref SASHELP.BASE refers to the catalog named BASE in the SASHELP library.

Rules for Library and Catalog Concatenation

Once a library or a catalog concatenation is established, its libref or catref may be used in any context that accepts a simple (non-concatenated) libref or catref.

When a file or catalog entry is opened or when a library or a catalog directory is listed for a concatenation there must be rules for locating the items among the parts of the concatenation. The rules for searching through the libraries and the catalogs are the same. In order to define the rules only once, the term *item* is used *to* refer to a SAS file in a library or to a catalog entry. The rules for library and catalog concatenation are as follows:

- When an item is open for input or update, the parts will be searched and the first occurrence of the item will be used. This is important when there are items with the same name in more than one part of the concatenation.
- When an item is open for output, it will be created in the first part of the concatenation. This is true even if there is an item with the same name in another part of the concatenation.
- When you want to delete or rename an item, the concatenation will be searched and only the first occurrence of the item will be affected.
- Any time a list of items is displayed for you, only one occurrence of an item name will be shown. So again, if one item name exists in multiple levels, only the first one will be listed for the you to see.
- When there are logically connected files (for example A.DATA and A.INDEX), displaying a list of members only lists the subordinate file (A.INDEX) when the parent file (A.DATA) resides in the same library. For instance, if the concatenation included libraries LIB1 and LIB2 and both contained the file A.DATA but only LIB2 contained A.INDEX, then A.INDEX wouldn't be listed; only A.DATA in the first library will be displayed (see previous rule) and there isn't an index file associated with that data file.

Example: Library and Catalog Concatenation

Assume that you have two SAS libraries - path1 and path2. A LIBNAME statement allows you to concatenate the two libraries:

```
Libname both ('path1' 'path2');
```

If the two libraries contained the following:

Path 1	Path 2
MYCAT.CATALOG	MYCAT.CATALOG
TABLE1.DATA	MYCAT2.CATALOG
TABLE3.DATA	TABLE1.DATA
	TABLE1.INDEX
	TABLE2.DATA
	TABLE2.INDEX

The concatenated libref BOTH would have the following:

BOTH
MYCAT.CATALOG (from path1 and path2)
MYCAT2.CATALOG(from path2)
TABLE1.DATA (from path1)
TABLE2.DATA (from path2)
TABLE2.INDEX(from path2)
TABLE3.DATA (from path1)

Notice that TABLE1.INDEX does not show up in the concatenation while TABLE2.INDEX does. The system suppresses listing the index when its associated data file is not part of the concatenation.

The catref BOTH.MYCAT is an implicit catalog concatenation. The concatenation is made by logically combining the two physical files that reside in 'path1' and 'path2'. To understand the contents of the concatenation BOTH.MYCAT, first look at the contents of both parts of the concatenation. Assume that MYCAT.CATALOG contains the following in 'path1' and 'path2':

Path1	Path2
A.FRAME	A.GRSEG
C.FRAME	B.FRAME
	C.FRAME

The combined catalog BOTH.MYCAT will contain:

BOTH.MYCAT
A.GRSEG(from path2)
A.FRAME(from path1)
B.FRAME(from path2)
C.FRAME(from path1)

Bigger is Better

Column names and labels, catalog entry names, SAS file names and labels, and character column values are longer in Version 7 of the SAS System. Bigger names and labels allow you to be more descriptive in your naming. A longer column value implies that you no longer have to chop things up into 200 byte chunks.

SAS Column Names (AKA SAS Variable Names)

The size of a column name in a SAS data set has increased from 8 bytes to 32 bytes. The size of a column label has increased from 40 to 256 bytes.

Column names are not only longer, but are now stored in mixed case. The case that is used to define a column is retained for display purposes. For example,

```
Data Bands;
 Bass = ’Boom’;
 Electric_Guitar = ’Screech’;
 Drums = ’Thud’;
Proc print; run;
```

Produces:

```
                The SAS System          1
                 Monday, December 22, 1997
        Electric_
Bass     Guitar      Drums
Boom     Screech     Thud
```

The above example shows that the case of column names is preserved for presentation. For column name matching or look up, however, the SAS System ignores case. For example, the column names "Dolly", "dolly", "DOLLY", and "doLLy" all refer to the same column.

The following BY statements all have the same effect:

```
proc print data=bands; by Guitar;
proc print data=bands; by GUITAR;
proc print data=bands; by guItAr;
```

Version 7 will support long mixed-case column names by default. However, some sites may want to restrict column names. For this reason, the VALIDVARNAME system option has been introduced.

The VALIDVARNAME system option lets you control what type of column names will be allowed in a SAS session. The VALIDVARNAME option will restrict the column names on new files that are created within a SAS session and will restrict the names of columns on files that are being read by the SAS session.

The VALIDVARNAME option has four possible values:

- ANY No restrictions on column names. Only the DATA step and the SQL procedure will be certified to run with this setting in Version 7. You will get a warning message when VALIDVARNAME is set to "ANY".
- V7 (default) Names may be up to 32 bytes in length and mixed case. Names must begin with an alphabetic character or an underscore and may contain only alphanumerics and underscores in the 2nd through nth character.
- UPCASE Like VALIDVARNAME=V7 except all column names are uppercased.
- V6 Column names are limited to uppercased 8 byte SAS names (like Version 6). For most applications, the column names created on new files when VALIDVARNAME=V6 will be automatically truncated to 8 bytes.

The VALIDVARNAME=ANY setting is used with SAS/ACCESS® libname engines to allow SAS applications to process column names with embedded blanks or other special characters that are not normally allowed in SAS names. Please reference Vino Gona’s paper titled "Version 7 Enhancements to SAS/ACCESS Software" in the "Proceedings of the Twenty Third Annual SAS User’s Group International Conference" for more details on how SAS/ACCESS engines use VALIDVARNAME=ANY.

Entry Names

The size of a catalog entry name has increased from 8 to 32 bytes in Version 7. Entry names, unlike column names, are uppercased by the system.

SAS File Names

The size of a SAS file name has increased from 8 to 32 bytes in Version 7. The size of a SAS data set label has increased from 40 bytes to 256 bytes. For the native SAS engines (V5, V6, V7, etc) SAS file names will be normalized by the system. The normalization of the file name will be host dependent. For example, VMS and MVS will upper case SAS file names while UNIX and Windows implementations will lower case SAS file names. SAS/ACCESS libname engines, however, will support true mixed-case file names. . Please reference Vino Gona’s paper titled "Version 7 Enhancements to SAS/ACCESS Software" in the "Proceedings of the Twenty Third Annual SAS User’s Group International Conference" for more details on how SAS/ACCESS engines handle file (or table) names.

Character Values

The maximum length of a character column value has increased from 200 bytes to 32K bytes. In Version 6, applications that processed character data that was longer than 200 bytes had to chop the data up into 200 byte pieces. In Version 7, character columns may be as long as 32K.

Cross Environment Data Access (CEDA)

Version 7 recognizes that diverse and distributed computers have become more common than they were when Version 6 was introduced. It is no longer unusual to have a site where multiple CPU’s share access to a single disk or to a single networked file system. CEDA is the facility that allows any V7 SAS data file created on any directory-based host (for example, Solaris, Windows, HP-UX, VMS, MVS HFS, etc) to be read by the SAS System running on any other platform.

In practical terms, with CEDA you can:

- Process a file that has gone through a binary transfer from one host to another (with FTP or similar file transfer software).
- Directly access a foreign file (one made by another CPU) residing on a shared network.

In Version 6, the SAS System required you to either use the data transfer services of SAS/CONNECT® software (the UPLOAD and DOWNLOAD procedures) or to use SAS/SHARE® or SAS/CONNECT's Remote Library Services (RLS) to access a data file created by and residing on another host. Data transfer services require you to establish two SAS sessions with a SIGNON command. RLS requires that a SAS/SHARE server be running or that a SAS/CONNECT SIGNON be established in order to access foreign data with a LIBNAME statement. RLS and data transfer services require a client and server SAS session in order to process a foreign file.

The advantage of CEDA over data transfer services and RLS is that you can FTP your file from one host to another or NFS mount a disk from another host and automatically be able to access your data without any extra steps. CEDA eliminates the need to execute any other procedure, maintain a running server, or even SIGNON to the remote host.

It is important to note that CEDA does not replace data transfer services or RLS. There are some restrictions the CEDA user must be aware of:

- Limited to Version 7 SAS data files - views and utility files are not processed by CEDA.
- Update opens are not supported. (Input and Output opens are supported.)
- No WHERE expression optimization with an index.
- Limited to directory-based libraries (bound libraries on MVS and CMS files are not part of CEDA).

If your application can operate within the above restrictions, CEDA provides a simpler cross-platform strategy than the Version 6 data transfer services and remote library services. If your needs go beyond these restrictions then data transfer services and remote library services are still available in Version7.

A user on a different platform cannot use CEDA software to reference an MVS bound library. However, the Version 7 SAS System for MVS supports unbound (or directory-based) libraries processed with Hierarchical File System (HFS). As an MVS user, you can use the IBM® NFS Client to get to UNIX files from an MVS SAS session. As a UNIX user, you can use an IBM NFS Server to reference MVS HFS files from a UNIX platform. CEDA performs the necessary translations so that these cross-platform references seem like local references.

CEDA technology has spawned two new options -- OUTREP and TRANTAB. These options may appear as data set or LIBNAME options. As LIBNAME options they provide defaults for the library. As data set options they apply to individual data set opens.

The OUTREP option is used on OUTPUT opens to determine the new file's data representation.

By default, the SAS System creates new files using the native representation of the CPU running the SAS System. In other words, a PC user creates a file with ASCII characters and byte-swapped integers.

The OUTREP= option allows the creator of a SAS data file to decide how the data should be represented on that file. This is useful when the readers of a file will be using a different CPU than the creator of the file. For example, an administrator running on MVS may wish to create a file on an NFS system. The readers of this file will all be running HP-UX. The creator can force the data representation to be in the reader's format by specifying OUTREP= HP—UX. The readers will get better performance because reading the file does not require any data conversions.

The list of values allowed for OUTREP is:

- ALPHA_VMS
- ALPHA_OSF
- HP_UX
- MAC
- OS2
- MVS
- RS_6000_AIX
- SOLARIS
- VAX_VMS
- WINDOWS

The TRANTAB option is used to provide a translation table for character conversions. For example:

```
libname foo '.' trantab=mytable;

proc print data=foo.a;
```

If FOO.A is a foreign data set, the translate table named MYTABLE will be used to translate the characters from foreign encoding to local encoding. Note that the system searches for translate tables in SASUSER.PROFILE.CATALOG and in SASHELP.BASE.CATALOG.

The TRANTAB and OUTREP options may be used together:

```
data stuff.three(outrep=HP_UX trantab=mytab2);
```

The file STUFF.THREE will be created in HP-UX format. The translate table MYTAB2 will be used for converting from local format to ASCII.

CEDA is licensed as part of SAS/CONNECT in Version 7.

SAS Data Set Versioning (Generations)

Generations were added as a feature in Version 7 to allow you to keep multiple copies of the same file. Multiple copies represent distinct versions or historical snapshots of a particular SAS file. Generations are supported for SAS data files (member type DATA) and for SAS data views (member type VIEW).

A generation group is a collection of files with the same name but different version numbers. Two new data set options have been added to support processing generation groups: GENMAX and GENNUM.

GENMAX is an OUTPUT data set option that specifies how many versions you want the system to maintain. The GENMAX option value ranges from 1 to 999. The SAS System cannot keep more than 999 versions at one time. A generation group, however, can support version numbers from 1 to 32,767.

For example, "data a(GENMAX=10);" will establish a file named "A" and will initialize a generation group. The next nine replacements of the file named "A" will retain the older versions as back copies. The 10th replacement of the file named "A" will delete the oldest (version number 1) while retaining the 2nd through 10th versions in the generation group. As time passes, the system will always maintain the last 10 copies of the file named A. After 3000 replacements, the version numbers will range from 2,991-3,000.

Note that file names in a generation group are limited to 28 (rather than 32) bytes. The last four bytes of the name in a generation group will be used to hold an escape character and a version number.

GENNUM is an input dataset option that specifies which historical version you wish to process. A positive GENNUM value is used to reference a specific generation number. For example, "proc print data=a(gennum=2999);" will print the 2,999th version of the file named A. A negative generation number is used to reference a version relative to the top of the generation group. For example, "proc print data=a(gennum=-3)" asks for three versions back from the current version. After 3,000 replacements of A the above proc print would yield the 2998th version. A GENNUM=0 is used to reference the current (or most recently created) version of a file.

SAS utilities (like the Datasets procedure) have added support for managing generation groups. For example:

- Version numbers are displayed in directory listings
- Deleting of a SAS file takes an optional version number argument. For example, "delete a(gennum=2);" deletes the 2nd version of the file named A while leaving the other versions intact. To delete the whole generation group enter "delete a(gennum=all). To delete only the top copy and restore the most recent version as the top copy enter: "delete a;"
- Renaming of a SAS file allows you to rename the entire group "change a=newa" ; or to rename a single historical version ("change a(gennum=2)=newa);"
- Support has been added for raising or lowering the GENMAX value for a generation group. For example, to change the GENMAX of a group from the current value to 5 enter "modify a(genmax=5)". Note that lowering the GENMAX value will have the effect of deleting enough of the eldest versions so as not to exceed the new maximum.

Integrity Constraints

Integrity Constraints is a new Version 7 feature that allows you to guarantee the correctness and consistency of your data. The design adheres to the SQL ANSI standards. An integrity constraint stored in a SAS data set restricts the data values that can be updated or inserted into a data set. They can be specified at data set creation time or after data already exist in the data set. In the latter situation, all data are checked to verify that they satisfy the candidate constraints before the constraints are added to the data set. Integrity constraints are enforced automatically by the SAS System for each add, update, and delete of data to the data set containing the constraints.

There are five basic types of integrity constraints:

- Not NULL—NULL (that is, missing) values are not allowed for the column
- Check—any valid WHERE expression
- Unique—all values for the column must be unique
- Primary Key—all data values must be unique and not NULL. A Primary Key may or may not be linked to a Foreign Key.
- Foreign Key—Foreign keys link one or more records in a data set to a specific record in another data set (containing a Primary Key). This linkage ensures the integrity of the parent-child relationship between a Primary Key (parent) record and its Foreign Key (child) records. This link exists when a Foreign Key value in one data set matches a Primary Key value in another data set. This relationship limits modifications to both the Primary Key and the Foreign Key as follows:

  - a Primary Key value cannot be changed without accounting for all matching values in the Foreign Key.

  - a Foreign Key value can only be changed to NULL or to a value found in the referenced Primary Key.

Not Null, Check, and Unique constraints are called 'general' constraints. A Primary Key that is not linked to any Foreign Key is a 'general' constraint also. A Foreign Key and the Primary Key it is linked to constitute a 'referential' constraint.

General constraints are limited in scope to a single data set. That is, they operate on data within one data set. Referential constraints associate columns and their values across data set boundaries.

You can create and delete integrity constraints using the DATASETS procedure, the SQL procedure, and Screen Control Language (SCL). The CONTENTS procedure lists constraints on a data set.

Given a data set about students and their grades, here is an example:

```
  /*------------------------------------*/
  /* Use the DATASETS procedure to add   */
  /* integrity constraints.             */
  /*------------------------------------*/
  proc datasets lib=work nolist;
    modify grades;
      ic create ck_grade   = check (where=(grade in
('A' 'B' 'C' 'D' 'F')));
NOTE: Integrity constraint ck_grade defined.
      ic create nn_testid  = not null (testid);
NOTE: Integrity constraint nn_testid defined.
      ic create nn_student = not null (student);
NOTE: Integrity constraint nn_student defined.
  run;
  /*------------------------------------*/

  /*Use the CONTENTS procedure to list the*/
  /*constraints                          */
  /*-------------------------------------*/
  proc contents data=grades; run;

Alphabetic List of Variables and Attributes

                                             (continued)
```

```
#    Variable    Type    Len    Pos
-----------------------------------
1    grade       Char      1      8
2    student     Char     10      9
3    testid      Num       8      0


-----Alphabetic List of Integrity Constraints-----


   Integrity                          Where
#  Constraint  Type      Variables    Clause
----------------------------------------------------
1  ck_grade    Check     grade      in ('A','B','C',
                                        'D','F')
2  nn_student  Not Null  student
3  nn_testid   Not Null  testid


  /*----------------------------------------------*/
  /* Insert an invalid value for GRADE.           */
  /*----------------------------------------------*/
  proc sql;
   insert into work.grades
    set student = "Fred",
        testid  = 130,
        grade   = "x";
ERROR: Data value(s) do not comply with integrity
constraint ck_grade for file GRADES.
```

Addressing Compressed Files by Observation Number

Data sets can be compressed in Version 6, but they cannot be addressed by observation number. This means you cannot use POINT= and FIRSTOBS= with compressed data sets. Version 7 removes these limitations.

Observation number (or record number) addressability allows you to access observations by their relative physical position within the data set. For example, the first observation in the data set is referred to as observation 1, the second is 2, and so on. This is illustrated by the use of the POINT= option of the DATA step SET statement and the use of observation numbers on the command line of the FSEDIT procedure.

An uncompressed data set contains fixed length records. Thus, randomly accessing an observation by its number is accomplished by using a relatively simple algorithm. In contrast, a compressed data set is a data set containing variable length records, each with a data-dependent size. Without some auxiliary data, locating the beginning of a given observation can only be accomplished by sequentially reading observations until the desired one is found.

In Version 7, an index is embedded in the compressed data set. This index provides rapid translation of an observation number into its disk address so that POINT= and FIRSTOBS= are fully supported for compressed data sets with negligible performance penalty.

The REUSE= option is not compatible with the internal index because it can cause observations to be stored in the 'middle' of the file. Thus, accessing such a data set could provide observations from a sequential pass in a different order than from a random pass. If REUSE= is specified, the internal index is not created and the data set is not addressable by observation number.

User-Specified Compression for Data Sets

In Version 6, compressed data sets were introduced with a single compression algorithm that compressed adjacent identical bytes. This compression algorithm works well for character data, but it may not provide much compression for numeric data. Users have complained about the limitations of the current algorithm and have asked for other algorithms, including the ability to supply their own compression function.

The Version 6 data set compression routine compresses identical consecutive bytes into a maximum of three bytes using RLE (Run Length Encoding) technology.

- 3 to 129 blanks are compressed into 2 bytes
- 3 to 66 binary zeros are compressed into 2 bytes
- 3 to 63 occurrences of any other character are compressed into 3 bytes

The RLE algorithm is good for character data.

In Version 7, an additional compression routine, RDC (Ross Data Compression) is supported. RDC is good for compressing binary (for example, numeric) data and is most effective when the size of the data exceeds several hundred bytes. Since compression operates on a single data set record at a time, the RDC algorithm may not work well on small records.

Some applications may be able to benefit from specialized compression algorithms. The SAS System will allow the use of a user-written compression function in place of the Institute-supplied functions. SAS/TOOLKIT® will be used to install the user's compression function. Further details of user-written compression can be found in the New Features, Changes, and Enhancements document for SAS/TOOLKIT.

In Version 6, compression is specified using the COMPRESS= global or data set option with values of YES or NO. In Version 7, the old syntax as well as new values are valid.

- CHAR | YES - RLE algorithm
- BINARY - RDC algorithm

The CONTENTS procedure will display the name of the compression routine you specify, including the user-written routine.

Use of YES | CHAR in Version 7 is backward compatible with Version 6. Use of BINARY in Version 7 is not backward compatible with Version 6.

In general, RLE has an advantage over RDC in two areas: CPU resources and compressed size. However, the CPU difference is not large, and the compressed size advantage of RLE changes as the record size grows. RDC is likely to provide better compression on records larger than 1000 bytes.

As with most data-dependent performance enhancements, you will need to try them with your data to see which one works best for you.

Miscellaneous Performance Improvements

Creating an Index Honors Sort Assertion

The SORT procedure stores the sort order, called the sort assertion, in the sorted data set. Certain procedures can save resources by knowing that the data are sorted.

In Version 7, when an index is created with the base engine, the software examines the sort assertion and does not invoke the sort if the data are already sorted as needed. No new syntax is required.

Centiles

A primary use of an index is to optimize a WHERE expression. The base engine must decide if it is cheaper (that is, faster) to satisfy the WHERE expression by reading all the records sequentially or using an index to randomly access only a subset of the records. The cost of using an index in Version 6 is based upon two data values: the minimum and maximum values for the indexed column. If the data are uniformly distributed, the cost analysis is quite accurate. However, data distributed according to some other scheme can cause the base engine to make the wrong decision. The Version 6 cost estimate could be off by 100% in extreme cases.

The error can be reduced by storing additional statistics with each index. These are conventional statistics called cumulative percentiles, or sometimes "quantiles" or "centiles" for short. What exactly are these new statistics, in everyday terms? Here is an intuitive definition: the nth centile is the value that is greater than or equal to the values in n% of all the records. The 100th centile is the maximum value; the 50th is the median. It turns out to be handy to have the minimum value also, which we may think of as the zero'th percentile. So the software stores a list of twenty-one values from the index: centiles 0, 5, 10, 15,... 95, 100. This allows the cost estimate for the number of records qualified by the WHERE expression to be accurate within 5%.

Let's take an example using the age of college students. Say the youngest is 16 and the oldest is 75, with most of them in the 19-25 range. In the tables below, the first row of numbers is the centiles, the second is the index values.

<u>Version 6 statistics (percentiles)</u>

Min	Max
0	100
16	75

<u>Version 7 statistics (percentiles)</u>

Min																				Max
0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
16	18	18	19	19	19	20	20	20	21	21	21	22	23	24	24	25	28	30	40	75

For the expression "where age > 24", the Version 6 cost estimate (of the number of qualified records) will be computed as (1-((24-16) / (75-16))) *100 or 86%. This is due to the (incorrect) assumption that the data are distributed uniformly between 16 and 75. The index would not be chosen.

In Version 7, the centiles show that the percent of selected records will really be around 20% and the index would likely be used.

While this change does result in much more accurate estimates for the number of qualified records, it is only one of several numbers that must be considered in the overall cost analysis. There are still cases when the base engine may make the wrong choice, but the number of such cases has been reduced substantially. See "Control over Index Usage in Where Expression" for syntax to override the base engine's choice.

Greater Use of Composite Indexes

Data sets often have multiple indexes, and some of them may be composite (that is, composed of multiple columns). Compound optimization is the process of optimizing multiple WHERE expression conditions with a single composite index.

In Version 6, the base engine makes some use of compound optimization. Simple WHERE expressions with EQ conditions ANDed together are supported. For example, "where frstname eq 'JOHN' and lastname eq 'SMITH'" can be optimized with a composite index on the columns LASTNAME and FRSTNAME. The IN operator and fully bounded range conditions (for example, 19<AGE <=25) are supported also.

Version 7 expands the list of WHERE expressions that can be compound optimized to include these examples:

Inequalities and Nots	Truncated Comparisons
i ^= 5	ch =: 'abc'
i not in (5,10)	ch ^=: 'abc'
not 1 < i < 5	ch >=: 'abc'
not 1 <= i < 5	ch <=: 'abc'
not 1 <= i <= 5	ch >: 'abc'
not 1 < i <= 5	ch <: 'abc'
i > 5	
i >= 5	
i < 5	
i <= 5	

For compound optimization to occur, the following must be true:

- Starting at the left side of the index description (that is, the list of columns in the composite index), at least the first two columns must be used in suitable WHERE expression conditions, specifically, conditions that use EQ or IN, a range operation, inequalities, nots, or truncated comparisons on literals.
- At least one of the matching WHERE expression conditions must be EQ or IN. You cannot have, for example, all range conditions.
- The conditions in the WHERE expression must be connected with AND and can occur in any order.

The MSGLEVEL=I option directs the software to log a message identifying which index is selected for optimization.

Let’s look at some examples. Assume a composite index on columns I, J, and CH:

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

This WHERE expression can be compound optimized because every condition specifies a column in the composite index, and each condition uses one of the supported operators. The base engine will position the composite index to the first entry that meets all three conditions and retrieve only records that exactly match the WHERE expression.

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

This WHERE expression can be compound optimized, but only on the columns I and J. Only records where "I in (1,4)" and "J = 5" will be retrieved, and then the base engine will reject those that fail to satisfy the pattern-matching comparison "K like ’%c’".

where J = 1 and K = 2;

This WHERE expression cannot be compound optimized because neither J nor K is the leftmost column in the index.

PROC APPEND With an Index

Adding records to a data set requires additional processing when the data set has one or more indexes. The base engine automatically keeps the values in the index consistent with the values in the data set. When many records are added, as with the APPEND procedure, the index overhead may be substantial. Changes to the base engine and the APPEND procedure in Version 7 have reduced this overhead in two areas.

Inserting Multiple Occurrences

A non-UNIQUE index stores multiple occurrences of the same value as one occurrence of the value followed by the RID (Record IDentifier) for each of the occurrences of the value. For large data sets with many multiple occurrences, the list of RIDs for a given value may require several pages in the index file. Since the RIDs are stored in physical order, any new record added to the data set with the given value will be stored at the end of the list of RIDs. Navigating through the index structures to find the end of the RID list can cause many I/O operations.

The base engine was modified to remember the previous position in the index tree so that when inserting more occurrences of the same value, the end of the RID list will be found quickly.

As an example, a test case based upon a customer-reported problem that appends 2000 new occurrences of a value required 4,494 I/O operations on the index file using Version 6. In Version 7, this test requires only 76 I/O operations, less than 2% of the cost in Version 6.

The APPEND Procedure

Adding records to a data set usually means appending them to the end of the file; they are added sequentially in the order they are received. Updating the index with the new values requires navigating through the index to find the correct physical location. If the data are sorted in the same order as the index, then index updates are more efficient because the navigation is performed in a sequential manner. For example, all multiple occurrences will be sorted together and the RID list for a given value can be updated once, rather than having to revisit the RID list multiple times.

Sorting the data to be appended prior to using the APPEND procedure can reduce the index update overhead. However, this technique may not work for you if you don’t want the data records sorted in this manner, or if you have multiple indexes. In the case of multiple indexes, you can only sort the data to help one index.

In Version 7, the APPEND procedure and the base engine cooperate to perform the sorting for you. The base engine delays the index updates until all the records have been appended to the data set. Then it sorts the data going into each index before updating the index. It does not sort the data appended to the data set.

Since the index is not updated until all the records have been appended, it is possible that errors may occur during the delayed index update. For example, a column with a UNIQUE index will not have its uniqueness validated until index update time. If a non-unique value is detected, the offending record will be deleted from the data set. This may cause deleted records in the data set after the append operation. In Version 6, these records are rejected before they are added to the data set.

There are situations that prevent the SAS System from using the new faster algorithm. You can enable the display of INFO messages to determine if the fast algorithm is being used, and if not, why not. The MSGLEVEL= option controls the display of these messages.

```
    options msglevel=i;
```

If the fast algorithm is being used, you should see a message like this:

- INFO: Engine’s fast-append process in use.

If the fast algorithm cannot be used, you may see any combination of the messages below:

- INFO: Engine’s fast-append process cannot be used because
- INFO: - There is no member level locking
- INFO: - Referential Integrity Constraints exist
- INFO: - Cross Environment Data Access is being used
- INFO: - There is a where clause present (on the BASE data set)

The default processing in Version 7 is to use the faster algorithm described here. However, you may want to use the Version 6 algorithm. Use the APPENDVER=V6 option on the APPEND procedure statement to obtain that behaviour. Here’s an example:

```
 proc append base=a data=b appendver=v6;
         run;
```

Control Over Index Usage in a WHERE Expression

As mentioned earlier, the base engine’s ability to select between using an index for optimization of a WHERE expression and doing a sequential pass of the data set has been improved in Version 7. However, there may still be cases when you want to override the base engine’s decision.

Two new data set options are available to control the use of indexes for optimization. Note that these are not global options.

IDXWHERE=YES

This option instructs the base engine to use the best available index to process the WHERE expression, even if a sequential pass is faster.

IDXWHERE=NO

This option instructs the base engine to perform a sequential pass of the data set, regardless of any indexes that could be used.

IDXNAME=<name>

This option specifies that existing index <name> is to be used regardless of performance considerations.

Absence of the IDXWHERE= option means the base engine will make the choice to use an index or not for optimization.

This example forces the use of index SIZE to process the WHERE expression.

```
data foo.houses;
   set bar.materials (where=(size > 3)
                      idxname=size);
    run;
```

Conclusions

This paper has presented a wide variety of new data features that have been added to Version 7 of the SAS System.   The variety of topics prevents the authors from going into great detail on any one feature. This paper is meant to give you an overview of some of the new features that will be in store for you with Version 7.

When Version 7 is in the field we hope to present more in-depth papers on these and other enhancements.

Acknowledgments

Many people were involved in the development of the features described in this paper. They include:

Lisa Brown	Library concatenation.
Barbara Foster	CEDA.
Jim Craig	Where clause optimization and centiles
Greg Dunbar	Testing and verification.
Gary Franklin	Integrity constraints.
Cynthia Grant	Index performance.
Art Jensen	Radix addressable compressed files, long variable names, and fast append.
Kevin Mosman	Generations, testing, and verification.
Diane Olson	Integrity constraints and generations.
Rebecca Perry	Catalog concatenation.
Kanthi Yedavalli	CEDA and long variable names.

Authors

Steve Beatrous
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
(919) 677-8000
sassmb@wnt.sas.com (Steve Beatrous)

Billy Clifford
SAS Institute Inc.
11920 Wilson Parke Ave.
Austin, TX 78720-0075
(512) 258-5171
saswdc@unx.sas.com (Billy Clifford)