

MULTIPLE ENGINE ARCHITECTURE AND THE BASE ENGINE FOR THE SAS® SYSTEM

William D. Clifford, SAS Institute Inc., Austin, TX

ABSTRACT

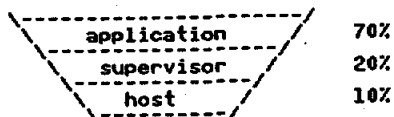
The SAS System supports access to data in a wide variety of formats with its Multiple Engine Architecture (MEA). The base engine for SAS software defines the format of data for all applications and is responsible for creating and accessing SAS data sets. This paper examines important features provided by the original base engine in Release 6.06 and describes enhancements made for Release 6.07. Included are feature and performance comparisons among Releases 5.18, 6.06, and 6.07, of the SAS System.

INTRODUCTION

MultiVendor Architecture (MVA)[™] is a term used to describe the structural design of the SAS System. First conceived in 1984, MVA provides a software architecture that maximizes the SAS System's ease of migration from one operating environment to another. With MVA, the SAS System is a vendor-independent system that runs the same across all environments it supports.

The benefits of MVA are shared by the user and SAS Institute. The interface to the SAS System is consistent across all supported operating environments. This consistency makes SAS System applications portable and eliminates the need to retrain users to use the SAS System on a new machine. The Institute is able to develop products once, in a portable manner, rather than develop the same product for each operating environment.

In its simplest form, the MVA can be visualized in three layers.

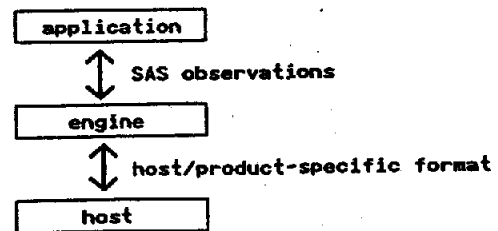


The application layer contains the DATA step and all procedures. The supervisor layer contains the engine supervisor and all engines. The host layer is at the bottom, and is the only part of the SAS System that is not portable. The host layer represents about 10% of the source code. This means that 90% of the source code is portable

and, therefore, is designed and developed once for all supported hosts.



The supervisor layer is responsible for managing access to data through a software subsystem called an engine. An engine is responsible for the transfer and translation of data between the host layer and the application. Below the engine, the data resides on mass storage devices managed by the host in a host or product-specific format. Above the engine, the application sees data as SAS observations. It is the engine's responsibility to translate the host or product-specific data into a SAS observation. This translation is transparent to the application.



There is an engine for each data format supported by the SAS System. A native engine is an engine that accesses data in a format created and maintained by SAS Institute. Examples of native engines are the tape, base, and compatibility engines. The other type of engine is an interface engine. Interface engines access data in a non-SAS format. Some examples of interface engines are IMS, DATACOM/DB, ADABAS, and ORACLE. Allowing applications to access data transparently in a variety of formats is the primary purpose of the Multiple Engine Architecture (MEA).

THE BASE ENGINE

The base engine defines the Version 6 SAS data set format for disk files. By default it is the engine used if no engine specification is made. Thus the

base engine is the most commonly-used engine.

To meet the growing needs of the SAS user community, the base engine contains a number of advanced features found in commercially available database management products. Two major benefits to be derived from this new functionality are ease of use and improved performance.

The remainder of this paper focuses on various aspects of the Release 6.07 base engine, comparing and contrasting its features with the Release 5.18 System and the Release 6.06 base engine.

DATA SET COMPRESSION

The base engine stores SAS observations in units called pages. A page is composed of host-specific disk blocks and contains multiple observations. Each observation is completely stored in a given page.

Release 5.18 of the SAS System supports a single observation type: uncompressed, or fixed-length. Version 6 supports uncompressed observations as well as a new optional data set format called compressed, or variable-length observations. A given data set contains only one type of observation, either compressed or uncompressed.

Uncompressed observations are stored in an unmodified format: numerics are stored as fixed-length floating-point, and characters are stored in fixed-length fields padded with blanks according to their length specification. Each uncompressed observation of a given data set occupies exactly the same number of bytes of storage.

Compressed observations start with the uncompressed format from the application layer but are compressed by the engine to remove redundancy. Thus, a compressed observation usually requires fewer bytes of storage than does its uncompressed counterpart. Each compressed observation in a given data set may be a different size.

Each page of an uncompressed data set contains the same number of observations. Pages of a compressed data set contain a variable number of observations. The significance of this is discussed later in the paper.

In Release 5.18, the only data compression feature available was the use of short numeric variables.

Version 6 adds the capability of compressing the entire observation regardless of its constituent variables.

The compression function supplied by the SAS System treats the entire observation as a single string of bytes. Variable types and variable boundaries are ignored. The compression algorithm compresses up to 1041 identical bytes into a maximum of 3 bytes.

Creating a Compressed Data Set

The COMPRESS option is used to specify the data set observation type, either uncompressed (the default) or compressed. This option is a permanent attribute of the data set and has meaning only when a data set is being created. Example syntax is:

```
option compress = yes;
```

```
data a (compress = yes);
```

Compress Tradeoffs

- In general, a compressed data set requires less disk storage than the same data set with uncompressed observations. Thus, fewer I/Os are required to process a compressed data set.
- As with Release 5.18, a Version 6 data set with uncompressed observations is addressable via the observation number. Since a compressed data set has a variable number of observations per page, it is not possible to directly calculate the address of a given observation from its observation number. Thus, compressed data sets are not currently addressable via the observation number.
- Observation compression/decompression and the additional bookkeeping necessary to manage variable-length records will require more CPU time. The reduced I/O may save CPU cycles on some machines so the net effect on CPU time for compressed files is host dependent.
- An updated observation in a compressed data set may not fit into its original location, and there may not be enough extra space on the page. In this case, the larger observation is stored on another page with a

pointer to it in the original location. Accessing this observation in the future may cause an extra I/O, but no more than one since such an observation is never more than one pointer away from its original location.

Compress Example

To demonstrate the impact of using the COMPRESS option, a data set used by SAS Institute and distributed to customers was converted to a compressed format. Several tests were run to measure the CPU and I/O impact of this compressed data set. The results presented here are not necessarily the same as you will have with your data set. Such results are dependent upon the data set characteristics. The data set used for these tests, the SAS Usage Notes data set, has 6176 observations, 61 variables, and an observation size of 2395 bytes. The performance analysis was done on the MVS operating system, but similar results with the same data can be expected on other hosts.

The performance of six typical applications was measured. These applications included interactive queries using the FSEDIT procedure, report generation, and batch updates. These performance numbers were generated with Release 6.06.

- ADD - Use the APPEND procedure to add 1000 new observations.
- REPORT - Use the TABULATE procedure to generate a report subsetting with a WHERE clause.
- BATCH - Use a series of DATA steps and SORT procedures to update a master file from a transaction file.
- FSED2 - Use the FSEDIT procedure to browse a subset with a WHERE clause.
- FSED4 - Use the FSEDIT procedure to ask for an observation (via a WHERE clause) that does not exist.
- FSED5 - Use the FSEDIT procedure to update 25 observations with a WHERE clause.

The numbers in the following table represent the ratio of resource utilization increase or decrease when using a compressed data set. Positive numbers indicate a decrease in resources required, and negative numbers indicate an increase in resources required. For example, 3.00 means 3 times fewer resources were used for a compressed data set.

Impact of using a compressed data set:

Application	Change in CPU Time	Change in Elapsed Time	Change in I/O (EXCP count)
ADD	-3.00	1.06	1.41
REPORT	-2.80	3.45	3.57
BATCH	-3.40	1.92	3.45
FSED2	-1.50	2.00	3.23
FSED4	-2.70	3.45	3.45
FSED5	-1.10	1.10	2.56

The data set size was reduced by compression from 1256 pages to 349 pages, or a 3.6-fold decrease. As can be seen from the performance data, the number of I/Os decreased in proportion to the decrease in data set size.

DATA SET INDEXES

An index is an auxiliary data structure used to assist in the location (selection) of a subset of the observations in a data set specified by the value of a data set variable. For example, all the observations with AGE > 65,

An index is called a single index if it contains the values of only one variable. A composite index contains the values for more than one variable concatenated to form a single value. A given data set can have multiple single and composite indexes. The indexes are stored in a separate file associated with the data set.

In addition, a given index can be specified to contain only unique values. The creation of such an index also prohibits duplicate values for its variable(s) from being stored into the data set. This option is useful for preventing duplicate values from incorrectly getting into a data set for variables such as account number and employee number. By default, duplicate values are allowed in the index and, thus, in the data set.

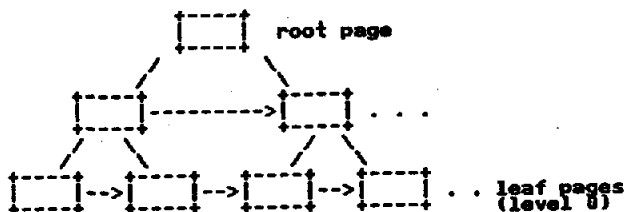
Another permanent index attribute is an option to prevent missing values from being stored in an index. Unlike the unique index option, the missing option does not prevent missing values from being stored in the data set. This option is useful if the variable(s) contain many missing values that would make the index large and, thus, slower to access than if they were excluded. By default, missing values are stored in an index.

Index Architecture

The index storage structure can be viewed as a

collection of objects (or pages) connected by pointers in the form of a tree. The tree has a single root, branches, and leaves at the end of the branches. By convention, the root of the tree is the highest level in the tree, and the leaves are at the lowest level (level 0).

Here is an example of a two level index:



Each level 0 or leaf node, contains a set of VALUE/RID pairs ordered by VALUE. VALUE is a value for the indexed data set variable, and RID is the record identifier for the observation in the data set containing VALUE. All of the leaf nodes taken together form an ordered set of VALUE/RID pairs for all values of the indexed variable. The first VALUE in the left-most leaf node is the lowest value for the indexed variable. The last VALUE in the right-most leaf node is the highest value.

Each non-leaf node (root and branches) contains a set of VALUE/NID pairs ordered by VALUE. VALUE is a value for the indexed variable, and NID is a child node identifier. VALUE is the highest value contained in the child node pointed to by NID. All of the non-leaf nodes taken together form a multi-level directory used to find a given VALUE/RID pair in a leaf node.

Thus the search for a requested value begins at the root node and moves down through one node at each level until the leaf node that should contain VALUE is found. If VALUE is not found in this node, then VALUE is not in the index. If VALUE is found, its RID can be used to read the observation containing VALUE. Subsequent VALUE/RID pairs higher (greater) than the requested value are found by reading the remaining VALUE/RID pairs in the node and then following the right pointer to leaf nodes containing the higher values.

Index Tradeoffs

- An index provides an extremely fast method of locating records by value. Without an index the entire data set must be searched

to find the desired observation(s).

- Indexes are stored on a separate file and, thus, require disk space.
- Every index that is used requires I/O buffers in memory.
- CPU and I/O resources are required initially to create an index.
- Whenever the data set is updated, the affected indexes also must be updated. The index updates are performed automatically by the engine whenever an observation is added, deleted, or modified.

Index Creation

SCL programs, the DATA step, and the DATASETS, SQL, and IML procedures can be used to create an index. Here is an example using PROC DATASETS to create a unique index, named NAMES, on variable ACCOUNT and a non-unique composite index on variables LNAME and FNAME for data set x:

```
proc datasets;
  modify x;
  index create account/unique;
  index create names = (lname fname);
run;
```

When Is an Index Used ?

In Version 6, indexes are used for three operations: WHERE-clause optimization, BY-clause processing, and optimization of the SQL procedure joins. The discussion here will be limited to WHERE-clause optimization.

The base engine automatically tries to use an index to optimize a WHERE clause. Variables specified in a WHERE clause are checked for being indexed. The WHERE-clause optimizer selects a single index to provide a coarse filter for the observations returned to the supervisor and then to the application. The supervisor may do a final filtering of the observations if the base engine returns observations that only partially satisfy the WHERE clause.

An index is selected for optimization only if both of the following conditions are true. First, the selected index must be the cheapest; that is, select the fewest observations among all indexes that

could be used. And second, reading the estimated number of observations from the candidate index must be cheaper than a sequential pass of the entire data set.

The global option MSGLEVEL can be used to determine if an index was used to optimize a WHERE clause. An informative message giving the name of the index will be written to the log when an index is used. Here is an example assuming there is an index on variable ACCOUNT:

```
23 options msglevel = i;
24 proc print;
25 where account = 36252;
   INFO: Index ACCOUNT selected for
       WHERE clause optimization.
26 run;
```

The cost effectiveness of an index depends upon several factors:

- the ratio of retrievals to updates. Updates to the data set cause updates to the index(es) and, therefore, consume more resources. The number of retrieval operations performed should be much higher than the number of update operations.
- data distribution. The index cost algorithms are based upon a uniform distribution of data values for the indexed variable. Thus, the closer the variable's distribution is to being uniform the more accurate the cost estimates are.
- discriminating values. The index is best at selecting small subsets of values. Indexing a variable with a small number of unique values is not likely to create an index that will be used very often. An index on the variable for an employee's sex would be a poor choice. An index on an employee's last name would be a good choice.
- subset size. Accessing the index to get to the data set requires some additional overhead so the size of the subset you request is important. An index should not be used (or be expected to be used) to select more than 25% of the data set. Much smaller subsets are more cost effective.

Index Example

A copy of the Usage Notes data set was made with

an index on one variable to compare the performance of indexed and unindexed versions of the same data set. The applications used are the same as for the COMPRESS analysis except the WHERE-clause variable is indexed.

The numbers in the table below represent the ratio of resource utilization increase or decrease for use of an index. Positive numbers indicate a decrease in resources required and negative numbers indicate an increase in resources required. For example, -6.80 means 6.80 times more resources were used with an index.

Impact of using an index:

Application	Change in CPU Time	Change in Elapsed Time	Change in I/O (EXCP count)
ADD	-6.80	-11.00	-15.00
REPORT	-1.47	1.72	1.49
FSED2	-1.26	4.55	9.09
FSED4	153.85	714.29	312.50
FSED5	-1.10	-1.80	-2.30

All test applications retrieving data using a WHERE clause (REPORT, FSED2, and FSED4) showed improved I/O performance due to optimization via the index. Updates (ADD and FSED5) however, are more costly. In addition, extra disk storage is required for the 16-page index file.

FEATURE COMPARISON OF RELEASES 5.18, 6.06, AND 6.07

Sequential Read Performance

To compare the sequential reading of a data set, another in-house data set was used. This data set is used by SAS Institute to maintain SUGI registration data. It has the following characteristics:

- 11,383 observations
- 1345 bytes per observation
- 107 variables (36 numeric, 71 character)

The test was "data _null_ set x.sugireg; run;". This test measures the resources used to get observations from the host, through the engine and engine supervisor, and to the application. It does not include any resources used in the analysis of the observations by the application.

The tests were run under MVS using the default page and block sizes, but similar results can be

expected on other hosts.

Version	Compress	Page Size	#Pages	CPU Seconds	I/O (excps)
5.18	no	22937	672	.61	n/a
6.06	no	12288	1266	.54	453
6.07	no	43008	368	.37	147
6.06	yes	18432	302	2.79	112
6.07	yes	43008	130	1.70	52

The numbers indicate that each succeeding version of the SAS System requires fewer resources for the same type of data set. The compressed data set requires substantial increases in CPU resources, but also requires significantly fewer I/O resources. The overall cost to run an application with a compressed data set is, of course, host dependent.

Note also that the compressed data set is 2.8 times smaller than the uncompressed one, saving disk storage and I/O resources.

Subsetting

In Release 5.18, subsetting a data set is a two-step process. First, the subset is extracted into a new data set, and then the new data set is processed by a procedure. Here is an example:

```
data a; set accounts;
  if (acctnum eq 1234) then output;
run;
proc print data=a;
run;
```

All observations in data set accounts are read, a new data set, a, is created, and finally the PRINT procedure reads all the observations in the data set a.

In Version 6, you can eliminate the DATA step subsetting part by using a WHERE clause.

```
proc print data=accounts;
  where acctnum eq 1234;
run;
```

In this case all observations in the accounts data set are read but the processing needed to create and read the intermediate data set is eliminated.

If there is an index on variable ACCTNUM, then the SAS program to print the report is exactly the same since the SAS System automatically uses an index.

```
proc print data=accounts;
  where acctnum eq 1234;
run;
```

But the resources used to produce the report are likely to be substantially reduced because the index will supply the addresses of the specified observations so that the entire data set does not have to be read.

Subsetting Performance

Again the SUGI registration data set described above was used to generate comparisons for subsetting tests. The following tests were used to select six observations from the data set:

```
T1  data a; set x.sugireg;
     if lname eq 'Clifford';
     run;
     proc print;
     var lname fname;
     run;
T2  proc print data=x.sugireg;
     where lname eq 'Clifford';
     var lname fname;
     run;
```

Version	Test	Compress	Index	CPU Seconds	I/O (excps)
5.18	T1	no	no	.62	n/a
6.06	T2	no	no	.55	452
6.07	T2	no	no	.24	147
6.06	T2	yes	no	2.83	115
6.07	T2	yes	no	1.60	53
6.06	T2	no	yes	.82	13
6.07	T2	no	yes	.82	11

Test T1 is the standard way of subsetting in Release 5.18. In Version 6, the WHERE clause (test T2) is easier to use and faster. The test results again show the performance of the SAS System improving with each new version. This series of tests demonstrates the I/O performance gains when using a compressed data set, and especially when using an index.

Types of WHERE Clauses Optimized

Release 6.06 can use an index to optimize several basic types of WHERE clauses:

- simple condition
where name = 'smith';

- multiple conditions
`where lname = 'smith' and salary > 20000;`
- other operators
`where age in (25, 35, 45);`
`where age between (25, 45);`
- can't optimize a WHERE clause with an OR because a single index will not supply all of the observations.
- only the first variable of a composite index is used.

In Release 6.07, more types of WHERE clauses can be optimized:

- SAS functions TRIM and SUBSTR
`where`
`where trim(lname) = 'smith';`
`where substr(lname, 1, 5) = 'smith';`
- LIKE, truncated comparison, and CONTAINS operators
`where lname like '%smith';`
`where lname = 'smith';`
`where lname contains 'smith';`
- a composite index will optimize multiple EQ conditions combined with AND
`where lname = 'smith' and fname = 'frank';`

Data Set File Format

The SAS System stores some bookkeeping, or overhead, information with each observation in a SAS data set. For example, a deleted flag is used to mark deleted observations. The amount of storage required for this information has varied from release to release.

- Release 5.18: 4 bytes (IBM mainframes)
- Release 6.06: 12 bytes
- Release 6.07: 1 bit for uncompressed
12 bytes for compressed

For small observations, the overhead of Release 5.18, and especially Release 6.06, can be a significant percentage of the overall data set size. In Release 6.07, the one-bit overhead for non compressed data sets virtually will eliminate the need for storage of overhead information. The 12-byte overhead is still needed for compressed data sets to locate an observation within a page.

Data Set Sort Order Assertion

Having sorted data is an important part of some applications. Internally, the SAS System has algorithms that can take advantage of the fact that data are being processed in sorted order. In Release 6.07, a new feature was added to maintain information, called the sort assertion, in the data set describing how it is sorted. Setting the sort assertion and using it is normally handled automatically by the SAS System. However, the user can override the sort assertion, if necessary.

PROC SORT sets the sort assertion whenever it sorts a data set, even if another vendor's sort is used. If the data set is already sorted as requested, PROC SORT does not sort the data set. Any updates to a data set, for example with PROC FSEDIT, that change the value of any of the sorted variables will cause the base engine to cancel the sort assertion. Adding observations to a data set also will cause the sort assertion to be canceled.

In Release 6.07, there are three internal uses of the sort assertion. PROC SORT, as previously described, will avoid the sort operation if the data set is already sorted. PROC SQL can optimize some join operations by eliminating sorts if it knows that the data set is sorted. And the base engine will not use an index to provide the requested BY ordering if the sort assertion specifies that the data set ordering is sufficient.

Index Creation Performance Improvements

The base engine index creation code in Release 6.06, makes a sequential pass of the data set extracting the indexed variable's values and inserts them into the index. In Release 6.07, this code was changed to sort the values to be indexed before inserting them in the index. In addition, a special fast insert algorithm is used that takes advantage of the fact that the data are being inserted in sorted order.

The new algorithms used for index creation have several benefits: a 50% reduction in CPU time, a 50% reduction in I/Os, and a 20% to 40% smaller index file. The index file is smaller because each leaf page can be filled to capacity before creating the next leaf page. When the values being inserted are not sorted, the leaf pages may not be completely full.

Data Set Passwords

Passwords are used to provide security for data. In Release 5.18, you can permit read-only access to a data set by specifying the READ password. Write privileges plus the ability to delete the data set are granted by the PROTECT password.

Release 6.06, unfortunately, did not support any passwords.

Release 6.07 supports READ, WRITE, and ALTER passwords for data sets, views, access and program files. As in Release 5.18, the READ password grants read-only access to data. The WRITE password provides READ privileges plus the ability to modify, delete, and add records and observations. The ALTER password is roughly equivalent to the Release 5.18 PROTECT password. It allows WRITE privileges plus the ability to delete the file, rename variables, and modify indexes. PW is an alias for the READ/WRITE/ALTER passwords.

Here are some examples:

```
Given that the data set was created with  
  READ = foo, WRITE = bar, then  
  proc fsedit data=a(pw=bar); is equivalent to  
  proc fsedit data=a(write=bar);  
And  
  data a(pw=stuff); is equivalent to  
  data a(read=stuff write=stuff alter=stuff);
```

SUMMARY

Through its MultiVendor Architecture, Version 6 of the SAS System provides a consistent interface across heterogeneous hardware platforms. The Multiple Engine Architecture supports direct access to data in a wide variety of formats. Data do not have to be converted to a SAS format for the SAS System to use them.

The Release 6.07 of the SAS System fills functionality gaps left out of Release 6.06, and it provides performance improvements over Release 6.06 making it faster than Release 5.18. There are no excuses left to avoid moving your applications to Version 6.

SAS is a registered trademark of SAS Institute Inc., Cary, NC, USA.

MultiVendor Architecture is a trademark of SAS Institute Inc., Cary, NC, USA.

IBM is a registered trademark of International Business Machines Corporation.

Other brand and product names are registered trademarks or trademarks of their respective companies.