SAS Publishing

SAS® 9.1 Scalable Performance Data Engine

Reference

The Power to Know®

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2004. *SAS® 9.1 Scalable Performance Data Engine: Reference*, Cary, NC: SAS Institute Inc.

SAS® 9.1 Scalable Performance Data Engine: Reference

Copyright © 2004, SAS Institute Inc., Cary, NC, USA

1-59047-156-3

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

1st printing, January 2004

Contents

What's New ***v***

Overview **v**

PART 1 Usage 1

Chapter 1 △ Overview: The SPD Engine 3

Introduction to the SPD Engine **3**

Using the SMP Machine **4**

How the SPD Engine Organizes SAS Data **4**

Comparing the Base SAS Engine and the SPD Engine **5**

Interoperability of the Base SAS Engine and the SPD Engine Data Sets **7**

Interoperability of the SPD Engine and the SPD Server Data Sets **8**

Sharing the SPD Engine Files **8**

Features That Enhance I/O Performance **8**

Features That Boost Processing Performance **9**

The SPD Engine Options **9**

Chapter 2 △ Creating and Loading SPD Engine Files 11

Introduction for Creating and Loading SPD Engine Files **11**

Allocating the Library Space **11**

Efficiency Using Disk Striping and Large Disk Arrays **14**

Converting Base SAS Engine Data Sets to SPD Engine Data Sets **14**

Creating and Loading New SPD Engine Data Sets **16**

SPD Engine Component File Naming Conventions **16**

Efficient Indexing in the SPD Engine **17**

PART 2 Reference 21

Chapter 3 △ SPD Engine LIBNAME Statement Options 23

Introduction to the SPD Engine LIBNAME Statement **23**

SPD Engine LIBNAME Statement Syntax **23**

SPD Engine LIBNAME Statement Options List **33**

Chapter 4 △ SPD Engine Data Set Options 35

Introduction to SPD Engine Data Set Options **35**

Syntax **35**

SPD Engine Data Set Options List **60**

SAS Data Set Options That Behave Differently with the SPD Engine Than with the Base SAS Engine **61**

SAS Data Set Options Not Supported by the SPD Engine **61**

Chapter 5 △ SPD Engine System Options 63

Introduction to SPD Engine System Options **63**
Syntax **63**
SPD Engine System Options List **71**
SAS System Options That Behave Differently with SPD Engine **71**

PART 3 Appendix 73

Appendix 1 △ Quick Guide to the SPD Engine Disk-I/O Set-Up 75
SPD Engine Disk-I/O Set-Up **75**
Disk Striping and RAIDs **76**
Metadata Area Configuration **77**
Data Area Configuration **77**
Index Area Configuration **81**
Work Area Configuration **84**
Configuration Validation Program **85**

Appendix 2 △ Recommended Reading 89
Recommended Reading **89**

Glossary 91

Index 95

What's New

Overview

The SAS Scalable Performance Data Engine (SPD Engine) is a SAS LIBNAME engine that provides rapid data delivery to applications by using multiple CPUs to provide parallel data I/O. The SPD Engine is not intended to replace the default Base SAS engine. Rather it is intended for rapid processing of very large data sets that are stored in partitions across multiple disk volumes.

Note: z/OS is the successor to the OS/390 operating system. SAS 9.1 for z/OS is supported on both OS/390 and z/OS operating systems and, throughout this document, any reference to z/OS also applies to OS/390, unless otherwise stated. △

PART 1

Usage

Chapter ***1*** . . . . **Overview: The SPD Engine** *3*

Chapter ***2*** . . . . **Creating and Loading SPD Engine Files** *11*

CHAPTER 1 Overview: The SPD Engine

Introduction to the SPD Engine **3**
Using the SMP Machine **4**
How the SPD Engine Organizes SAS Data **4**
  Metadata Component Files **5**
  Index Component Files **5**
  Data Component Files **5**
Comparing the Base SAS Engine and the SPD Engine **5**
  The SPD Engine Libraries and File Systems **6**
  Utility File Workspace **6**
  Temporary Storage of Interim Data Sets **6**
  Other Similarities/Differences between the Base SAS Engine Data Sets and the SPD Engine Data Sets **6**
Interoperability of the Base SAS Engine and the SPD Engine Data Sets **7**
Interoperability of the SPD Engine and the SPD Server Data Sets **8**
Sharing the SPD Engine Files **8**
Features That Enhance I/O Performance **8**
  Multiple Directory Paths **8**
  Physical Separation of the Data File and the Associated Indexes **8**
  WHERE Optimization **9**
Features That Boost Processing Performance **9**
  Implicit Sort Capabilities **9**
  Queries Using Indexes **9**
  Parallel Index Creation **9**
The SPD Engine Options **9**

Introduction to the SPD Engine

The Scalable Performance Data Engine (SPD Engine) is designed for high-performance data delivery. It enables rapid access to SAS data for processing by the application. The SPD Engine delivers data to applications rapidly because it organizes the data into a streamlined file format that takes advantage of multiple CPUs to perform parallel input/output functions.

The SPD Engine uses *threads* to read blocks of data very rapidly and in parallel. The software tasks are performed in conjunction with an operating system that enables threads to execute on any of the machine's available CPUs. Although threaded I/O is an important part of the SPD Engine functionality, the real power of the SPD Engine comes from the way that the software structures SAS data. The SPD Engine organizes data into a new file format that includes partitioning of the data. This data structure permits threads, running in parallel, to perform I/O tasks efficiently.

Although it is not intended to replace the default Base SAS engine, the SPD Engine is a high-speed alternative for processing very large data sets. It reads and writes data sets that contain millions of observations, data sets that expand beyond the 2-gigabyte size limit imposed by some operating systems, and data sets that SAS analytic software and procedures must process faster.

The SPD Engine performance is boosted in these ways:

- support for gigabytes of data
- scalability on symmetric multiprocessor (SMP) machines
- parallel WHERE selections
- parallel loads
- parallel index creation
- parallel I/O data delivery to applications
- implicit sorting on BY statements.

The SPD Engine runs on UNIX, Windows, z/OS (zFS file system only), and OpenVMS Alpha (on ODS-5 file systems only).

All operating environment-specific information is included in this documentation and not in the SAS Companion documentation.

Note: Be sure to visit the Scalability Community Web site at **`support.sas.com/rnd/scalability`** for more information about scalability in SAS 9.1 △

Using the SMP Machine

The SPD Engine exploits a hardware and software architecture known as symmetric multiprocessing (SMP). An SMP machine has multiple central processing units (CPUs) and an operating system that supports threads. An SMP machine is usually configured with multiple controllers and multiple disk drives per controller. When the SPD Engine reads a data file, it launches one or more threads for each CPU; these threads then read data in parallel from multiple disk drives, driven by one or more controllers per CPU. The SPD Engine running on an SMP machine provides the capability to read and deliver much more data to an application in a given elapsed time.

For example, in a perfectly tuned system, reading a data set with an SMP machine that has 5 CPUs and 10 disk drives could be as much as 5 times faster than I/O on a single-CPU machine. In addition to threaded I/O, an SMP machine enables threading of application processes, for example, threaded sorting in the SORT procedure in SAS 9.1

The exact number of CPUs on an SMP machine varies by manufacturer and model. The operating system of the machine is also specialized; it must be capable of scheduling code segments so that they execute in parallel. If the operating system kernel is threaded, performance is further enhanced because it prevents contention between the executing threads.

As threads run on the SMP machine, managed by a threaded operating system, the available CPUs work together. The synergy between the CPUs and threads enables the software to scale processing performance. The scalability, in turn, significantly increases overall processing speed for tasks such as creating data sets, appending data, and querying the data by using WHERE statements.

How the SPD Engine Organizes SAS Data

Because the SPD Engine organizes data for high-performance processing, an SPD Engine data set is physically different from a Base SAS engine data set. The Base SAS engine stores data in a single data file that contains both data and data descriptors for the file (metadata). The SPD Engine creates separate files for the data and data descriptors. In addition, if the data set is indexed, two index files are created for each index. Each of these four types of files is called an SPD Engine *component file* and each has an identifying file extension.

In addition, each of these components can comprise one or more physical files so that the component can span volumes but can be referenced as one logical file. For example, the SPD Engine can create many physical files containing data, but reference it as a single data component in an SPD Engine data set. The *metadata* and index components differ from the data component in two ways:

1 You can specify a fixed length partition size for data component files using the PARTSIZE= option. However, you have little or no control over the size of the metadata or index partitions.

2 The data component files are created in a cyclical fashion across all defined paths. The metadata and index components are created in a single path until that path is full; then the next path is used.

Metadata Component Files

The SPD Engine data set stores the descriptive metadata in a file with the file extension .mdf. Usually an SPD Engine data set has only one .mdf file.

Index Component Files

If the file is indexed, the SPD Engine creates two index component files for each index. Each of these files contains a particular view of the index, so both are present for each data set.

- The index file with the .hbx file extension contains the global index.
- The index file with the .idx file extension contains the segment index.

Data Component Files

The data component of an SPD Engine data set can be several or many files (partitions) per path or device, rather than just one. Each of these partitions is a fixed length, specified by you when you create the SPD Engine data set.

Specifying a partition size for the data component files allows you to tune the performance of your applications because the partitions are the threadable units, that is, each partition (file) is read in one thread. Chapter 2, “Creating and Loading SPD Engine Files,” on page 11 provides details on how the SPD Engine stores data, metadata, and indexes.

Comparing the Base SAS Engine and the SPD Engine

Base SAS engine data sets and SPD Engine data sets have many similarities. They both store data in a SAS data library, which is a collection of files that reside in one or more directories. However, since the SPD Engine data libraries can span devices and file systems, the SPD Engine is ideal for use with very large data sets. Also, the SPD Engine enables you to specify separate directories, or devices, for each component on the LIBNAME statement. Chapter 2, “Creating and Loading SPD Engine Files,” on page 11 provides details on designing and setting up the SPD Engine data libraries.

The SPD Engine Libraries and File Systems

An SPD Engine library can contain data files, metadata files, and index files. The SPD Engine does not support catalogs, SAS VIEWS, MDDBs, or other utility (byte) files.

The SPD Engine uses the zFS file system for OS/390 and z/OS and the ODS-5 file system for OpenVMS Alpha. This means that some functionality might be slightly different on these platforms. For example, for OS/390 and z/OS, the user must have a home directory on zFS.

Utility File Workspace

Utility files are generated during the SPD Engine operations that need extra space, for example, when creating parallel indexes or when sorting very large files. Default locations exist for all platforms but, if you have large amounts of data to process, the default location may not be large enough. The SPD Engine system option SPDEUTILLOC= lets you specify a set of file locations in which to store utility scratch files. See "SPDEUTILLOC= System Option" on page 68 for details.

Temporary Storage of Interim Data Sets

To create a library to store interim data sets, specify the SPD Engine option TEMP= on the LIBNAME statement. If you want current applications to refer to these interim files using one-level names, specify that library on the USER= system option.

This example code creates a user libref for interim data sets. It is deleted at the end of the session.

```
libname user spde '/mydata' temp=yes;
data a; x=1;
run;
proc print data=a;
```

The USER= option can be set in the configuration file so that applications that reference interim data sets with one-level names can run in the SPD Engine.

Other Similarities/Differences between the Base SAS Engine Data Sets and the SPD Engine Data Sets

The following chart compares the SPD Engine capabilities to Base SAS engine capabilities.

Table 1.1 Comparing the Base SAS Engine Data Sets and the SPD Engine Data Sets

Feature	SPD Engine	Base SAS Engine
Partitioned data sets	yes	no
Parallel WHERE optimization	yes	no
Lowest locking level	member	record
Concurrent access from multiple SAS sessions on a given data set	READ (INPUT open mode)	READ/WRITE (all open modes)
Support in SAS/SHARE	no	yes

Feature	SPD Engine	Base SAS Engine
Implicit sort for SAS BY processing (sort a temporary copy of the data to support BY processing)	yes	no
User-defined formats and informats	yes, except in WHERE[1]	yes
Catalogs	no	yes
Views	no	yes
MDDB	no	yes
Integrity constraints	no	yes
Data sets generations	no	yes
CEDA	no	yes
Audit trail	no	yes
NLS transcoding	no	yes
DBCS support	no	yes
Ability to change/modify passwords using PROC DATASETS	no	yes
Number of variables	more than 32,767	more than 32,767
Number of observations	2^{63}-1 (all hosts)	2^{31}-1 (on 32-bit hosts) 2^{63}-1 (on 64-bit hosts)
COMPRESS=	YES | NO	YES | NO | CHAR | BINARY
Functions and call routines	yes, with some exceptions[2]	yes
Encryption	yes	yes
Observations returned in physical order	no by default; yes via THREADNUM=1	yes

1 In WHERE processing, user-defined formats and informats are passed to the supervisor for handling; therefore, they are not processed in parallel.

2 In WHERE processing, functions and call routines introduced in SAS 9 or later are passed to the supervisor for handling; therefore, they are not processed in parallel.

Interoperability of the Base SAS Engine and the SPD Engine Data Sets

Base SAS engine data sets must be converted to the SPD Engine format in order for the SPD Engine to access them. You can convert the Base SAS engine data sets easily using the COPY procedure, the APPEND procedure, or a DATA step. (PROC MIGRATE cannot be used.) In addition, most of your existing SAS programs can run on the SPD Engine files with little modification other than the LIBNAME statement. Chapter 2, “Creating and Loading SPD Engine Files,” on page 11 provides details of converting Base SAS engine data sets to the SPD Engine.

Interoperability of the SPD Engine and the SPD Server Data Sets

UNIX and Windows only support the interoperability that the SAS 9.1 SPD Engine can read the SPD Server 4.0 files and the SPD Server 4.0 product can read the SPD Engine files created in SAS 9.1 unless they have more than 32,767 variables.

There are certain access requirements to enable this interoperability.

- The OS permissions must be set for access of the files.
- The SPD Engine file must be accessed via the server product using USER=ANONYMOUS on the LIBNAME statement.
- The server product files, to be usable by the SPD Engine, must not have an ACL (access control list) owner; they must be stored with USER=ANONYMOUS.

However, some features of the SPD Server cannot be used with the SPD Engine files.

- The SPD Server backup and restore process cannot be used on SPD Engine files.
- The SPD Server index reorganization utility cannot be used. If the SPD Engine indexes become skewed, delete and then re-create them.
- The enhanced SQL capabilities of the SPD Server, for example, parallel BY group processing and SQL pass-through, are not supported in the SPD Engine.

Sharing the SPD Engine Files

The SPD Engine supports member-level locking, which means that multiple users can have the same SPD Engine data set open for INPUT (read only). However, if an SPD Engine data set has been opened for update, then only that user can access it. If you want to share the SPD Engine data sets among multiple users who could be both reading and updating the data set, you must use the SPD Server product.

Features That Enhance I/O Performance

The SPD Engine introduces several new features that enhance I/O performance. These features can dramatically increase the performance of I/O bound applications, in which large amounts of data must be delivered to the application for processing.

Multiple Directory Paths

You can specify multiple directory paths and devices for each component type, because the SPD Engine can reference multiple physical files across volumes as a single logical file. For very large data sets, this feature circumvents any file size limits that the operating system might impose.

Physical Separation of the Data File and the Associated Indexes

Because each component file type can be stored in a different location, file dependencies are not a concern when deciding where to store the component files. Only cost, performance, and availability of disk space need to be considered.

WHERE Optimization

The SPD Engine uses a WHERE optimization process, developed for the SAS SPD Server product. The SPD Engine automatically determines the optimal process to use to evaluate observations for qualifying criteria specified in a WHERE statement. WHERE statement efficiency depends on such factors as whether the variables in the expression are indexed. A WHERE evaluation planner is included in the SPD Engine which can choose the best method to use to evaluate WHERE expressions that use indexes to optimize evaluation.

Features That Boost Processing Performance

The SPD Engine also provides several features that can boost processing performance of CPU-bound applications.

Implicit Sort Capabilities

The SPD Engine's implicit sort capabilities save time and resources for SAS applications that process large data sets. With the SPD Engine, you do not need to invoke the SORT procedure before you submit a SAS statement with a BY clause. When the SPD Engine encounters a BY clause, if the data is not already sorted or indexed on the BY variable, the SPD Engine automatically sorts the data without affecting the permanent data set or producing a new output data set.

Queries Using Indexes

Large data sets can be indexed to maximize performance. Indexes permit rapid WHERE-expression evaluations for indexed variables. The SPD Engine takes advantage of multiple CPUs to search the index component file efficiently.

Parallel Index Creation

In addition, the SPD Engine supports parallel index creation so that indexing large data sets is not time consuming. The SPD Engine decomposes data set append or insert operations into a set of steps that can be performed in parallel. The level of parallelism depends on the number of indexes present in the data set. The more indexes you have, the greater the exploitation of parallelism during index creation. However index creation requires utility file space and memory resources.

The SPD Engine Options

The SPD Engine works with many Base SAS engine options. In addition, the SPD Engine-specific options allow you to further manage the SPD Engine libraries and processing.

See

- Chapter 3, "SPD Engine LIBNAME Statement Options," on page 23

- Chapter 4, “SPD Engine Data Set Options,” on page 35
- Chapter 5, “SPD Engine System Options,” on page 63.

CHAPTER 2
Creating and Loading SPD Engine Files

Introduction for Creating and Loading SPD Engine Files **11**
Allocating the Library Space **11**
Configuring Space for All Components in a Single Path **12**
Configuring Separate Library Space for Each Component File **12**
Anticipating the Space for Each Component File **12**
Storage of the Metadata Component Files **13**
Example: Initial Set of Paths **13**
Example: Adding Subsequent Paths **13**
Storage of the Index Component Files **14**
Storage of the Data Partitions **14**
Efficiency Using Disk Striping and Large Disk Arrays **14**
Converting Base SAS Engine Data Sets to SPD Engine Data Sets **14**
Converting Base SAS Engine Data Sets Using PROC COPY **15**
Converting Base SAS Engine Data Sets Using PROC APPEND **15**
Creating and Loading New SPD Engine Data Sets **16**
SPD Engine Component File Naming Conventions **16**
Efficient Indexing in the SPD Engine **17**
Parallel Index Creation **18**
Parallel Index Updates **18**

Introduction for Creating and Loading SPD Engine Files

This section provides details on allocating SPD Engine libraries and creating and loading SPD Engine data and indexes. Performance considerations related to these tasks are also discussed.

Allocating the Library Space

To realize performance gains through SPD Engine's partitioned data I/O and threading capabilities, the SPD Engine libraries must be properly configured and managed. Optimally, a SAS system administrator will perform these tasks.

An SPD Engine data set requires a file system with enough space to store the various component files. Often that file system includes multiple directories for these components. Usually, a single directory path (part of a given file system) is constrained by a volume limit for the file system as a whole. This limit is the maximum amount of disk space configured for the file system to use.

Within this maximum file space, you must allocate enough space for all of the SPD Engine component files. Understanding how each component file is handled is critical to estimating the amount of storage you will need in each library.

Configuring Space for All Components in a Single Path

In the simplest SPD Engine library configuration, all of the SPD Engine component files (data files, metadata files, and index files) can reside in a single path called the *primary path*. The primary path is the default path specification in the LIBNAME statement. The following LIBNAME statement sets up the primary file system for the MYLIB library:

```
libname mylib spde '/disk1/spdedata';
```

Because there are no other path options specified, all component files will be created in this primary path. While storing all component file types in the primary path is simple and works for very small data sets, it doesn't take advantage of the performance boost that storing components separately achieves nor does it take advantage of multiple CPUs.

Note: The SPD Engine requires fully qualified pathnames to be specified. △

Configuring Separate Library Space for Each Component File

Most sites use the SPD Engine to manage very large amounts of data, which can have thousands of variables, some of them indexed. At these sites, separate storage paths are usually defined for the various component types. In addition, using disk-striping and RAIDS can be very efficient. Refer to Appendix 1, "Quick Guide to the SPD Engine Disk-I/O Set-Up," on page 75 for additional information.

All metadata component files must begin in the primary path, even if they span devices. In addition, specifying separate paths for the data and index components provides further performance gains. This is because the I/O load is distributed across disk drives and because separating the data and index components helps prevent disk contention and increases the level of *parallelism* that can be achieved, especially in complex WHERE evaluations. The following example code specifies a primary path for the metadata and uses the "DATAPATH= LIBNAME Statement Option" on page 26 and "INDEXPATH= LIBNAME Statement Option" on page 28 to specify additional, separate paths for the data and index component files:

```
libname all_users spde '/disk1/metadata'
   datapath= ('/disk2/userdata' '/disk3/userdata')
   indexpath= ('/disk4/userindexes' '/disk5/userindexes');
```

The metadata is stored on disk1, which is the primary path. The data is on disk2 and disk3, and the indexes are on disk4 and disk5. For all path specifications you must specify the fully qualified pathname.

CAUTION:

The primary path must be unique for each library. If two librefs are created with the same primary path but with differences in the other paths, data can be lost. △

Anticipating the Space for Each Component File

In order to properly configure the SPD Engine library space, you need to understand the relative sizes of the SPD Engine component files. The following information provides a general overview. More details are located in Appendix 1, “Quick Guide to the SPD Engine Disk-I/O Set-Up,” on page 75.

Metadata component files are relatively small files, so the primary path might be large enough to contain all the metadata files for the library.

Index component files (both .idx and .hbx) can be medium to large depending on the number of distinct values in each index and whether the indexes are single or composite indexes. When an index component file grows beyond the space available in the current file path, a new component file is created in the next path.

Data component files can be quite numerous, depending on the amount of data and the partition size specified for the data set. Each data partition is stored as a separate data component file. The size of the data partitions is specified in the “PARTSIZE= LIBNAME Statement Option” on page 29. Data files are the only component files for which you can specify a partition size.

Storage of the Metadata Component Files

Because much of the information that the SPD Engine needs in order to efficiently read and write partitioned data is stored in the metadata component, the SPD Engine must be able to rapidly access that metadata. By design, the SPD Engine expects every data set’s metadata component to begin in the primary path. These metadata component files can overflow into other paths (specified in the “METAPATH= LIBNAME Statement Option” on page 28) but they must always begin in the primary path. This is a very important concept to understand because it directly affects whether you can add data sets (with their associated metadata files) to the library.

When the space in the primary path is full, and a new data set for that library is created, the SPD Engine cannot begin the metadata component file in that primary path as required. The create operation will fail with an appropriate error message. To successfully create a new data set in this case, you must either free space in the primary path or assign a new library so the metadata component file can begin in the primary path for the new library. You cannot use the METAPATH= option to create space for a new data set’s first metadata partition. METAPATH= only specifies overflow space for a metadata component that begins in the primary path but has expanded to fill the space reserved in the primary path. Therefore, if you anticipate that your metadata component will grow to exceed the file size or library space limitations, and you want to ensure you have space in the primary path for additional data sets, specify an overflow path for metadata in the METAPATH= option when you first create the library.

For data and index component files, however, you can specify additional space at a later time, even if you specified separate paths for data and index component files in the initial LIBNAME statement.

Example: Initial Set of Paths

In this example, the LIBNAME statement specifies the MYLIB directory for the primary path. By default, this path is used to store initial metadata partitions. Other devices and directories are specified to store the data and index partitions.

```
libname myref spde 'mylib'
   datapath=('/mydisk30/siteuser')
   indexpath=('/mydisk31/siteuser');
```

Example: Adding Subsequent Paths

Later, if more space is needed, for example for appending large amounts of data, additional devices are added for the data and indexes. For example,

```
libname myref spde 'mylib'
   datapath=('/mydisk30/siteuser' '/mydisk32/siteuser' '/mydisk33/siteuser')
   indexpath=('/mydisk31/siteuser' '/mydisk34/siteuser');
```

Storage of the Index Component Files

Index component files are also stored based on overflow space. When several file paths are specified with the INDEXPATH=option, index files are started in the first available space and then overflow to the next file path when the previous space is filled. Unlike metadata components, index component files do not have to begin in the primary path.

Storage of the Data Partitions

The data component partitions are the only files for which you can specify the size. Partitioned data can be processed in threads easily, thereby taking full advantage of multiple CPUs on your machine.

The partition size for the data component is fixed and it is set at the time the data set is created. The default is 16 megabytes, but you can specify a different partition size using the PARTSIZE= option. Performance depends on appropriate partition sizes. You are responsible for knowing the size and uses of the data so that SPD Engine data sets can be created with a partition size that results in a balanced number of observations. (See “PARTSIZE= Data Set Option” on page 48 for details.)

Unlike index and metadata files, many data partitions can be created in each data path for a given data set. The SPD Engine uses the collection of file paths that you specify with the DATAPATH= option to distribute partitions in a cyclic fashion. The SPD Engine creates the first data partition in the first path, the second partition in the next path, and so on. The software continues to cycle among the file paths, as many times as needed, until all data partitions for the data set are stored.

For example, assume that you specify the following in your LIBNAME statement:

```
datapath=('/data1''/data2')
```

The SPD Engine stores the first partition in /DATA1, the second partition in /DATA2, the third partition in /DATA1, and so on. Cyclical distribution of the data partitions creates disk striping, which can be highly efficient. Disk striping is discussed in detail in Appendix 1, “Quick Guide to the SPD Engine Disk-I/O Set-Up,” on page 75.

Efficiency Using Disk Striping and Large Disk Arrays

If your system has a file creation utility that enables you to override the file system limitations and create file systems (volumes) greater than the space on a single disk, you can use the utility to allocate SPD Engine libraries that span multiple disk devices, such as redundant arrays of independent disks (RAIDs). RAID configurations employ a technique called disk striping that can significantly enhance I/O. For more information on disk striping and RAIDS, see Appendix 1, “Quick Guide to the SPD Engine Disk-I/O Set-Up,” on page 75.

Converting Base SAS Engine Data Sets to SPD Engine Data Sets

You can convert existing Base SAS engine data sets to SPD Engine data sets using these methods:

- PROC COPY
- PROC APPEND.

Some limitations apply. For example, if your Base SAS engine data has integrity constraints, then the integrity constraints are dropped when the file is created in the SPD Engine format. The following chart of file characteristics indicates whether that characteristic will be retained, dropped, or result in error when converted.

Table 2.1 Conversion Results for Base SAS File Characteristics

Base SAS File Characteristic	Conversion Result
Indexes	Rebuilt in SPD Engine (in parallel if ASYNCINDEX=YES)
Base SAS engine COMPRESS=YES*	Converts with SPD Engine compression (COMPRESS=YES)
Base SAS engine ENCRYPT=YES	Encryption retained
CHAR compression	Changed to SPD Engine compression (COMPRESS=YES)
BINARY compression	Results in ERROR
User defined compression	Results in ERROR
Integrity constraints	Dropped without ERROR
Audit file	Dropped without ERROR
Generations file	Dropped without ERROR

* If the Base SAS engine file has both compression and encryption, the compression is dropped but the encryption is retained.

Converting Base SAS Engine Data Sets Using PROC COPY

To create an SPD Engine data set from an existing Base SAS engine data set you can simply use the COPY procedure as shown in this example. The PROC COPY statement copies the Base SAS engine-formatted data set LOCAL.RACQUETS to a new SPD Engine-formatted data set SPORT.RACQUETS.

```
libname sport spde 'conversion_area';

proc copy in=local out=sport;
   select racquets;
run;
```

Even though the indexes on the Base SAS engine data set are automatically regenerated as the SPD Engine indexes (both .hdx and .idx files), they are not created in parallel because the data set option ASYNCINDEX=NO is the default. The default partition size is 16 megabytes.

Converting Base SAS Engine Data Sets Using PROC APPEND

Use the APPEND procedure when you need to specify data set options for a new SPD Engine data set.

This example creates an SPD Engine data set from a Base SAS engine data set using PROC APPEND. The ASYNCINDEX=YES data set option specifies to build the indexes in parallel. The PARTSIZE= option specifies to create partitions of 100 megabytes.

```
libname spdelib spde 'old_data';
libname somelib 'old_data';
proc append base=spdelib.cars (asyncindex=yes partsize=100)
  data=somelib.cars;
run;
```

Creating and Loading New SPD Engine Data Sets

To create a new SPD Engine data set, you can use the DATA step, any PROC statement* with the OUT= option, or PROC SQL with the CREATE TABLE= option.

The following example uses the DATA step to create a new SPD Engine data set, CARDATA.OLD_AUTOS in the report_area directory. The options ENCRYPT= and PW= are Base SAS engine options; COMPRESS= is the SPD Engine-specific option.

```
libname cardata spde '/report_area'(compress=yes encrypt=yes pw=secret);

data cardata.old_autos;
   infile 'old_cars';
   input year $4. @6 manufacturer $12. @18 model $12. @31 body_style $5. @37
   engine_liters @42 transmission_type $1. @45 exterior_color
   $10. @55 mileage @62 condition;

datalines;

1966 Ford        Mustang      conv  3.5  M  white     143000 2
1967 Chevrolet   Corvair      sedan 2.2  M  burgundy   70000 3
1975 Volkswagen  Beetle       2door 1.8  M  yellow     80000 4
1987 BMW         325is        2door 2.5  A  black     110000 3
1962 Nash        Metropolitan conv  1.3  M  red       125000 3
;
```

SPD Engine Component File Naming Conventions

When you create an SPD Engine data set, many component files can result. SPD Engine component files are stored with the following naming conventions:

```
filename.mdf.0.p#.v#.spds9
filename.dpf.fuid.p#.v#.spds9
filename.idxsuffix.fuid.p#.v#.spds9
filename.hbxsuffix.fuid.p#.v#.spds9
```

filename
   a valid SAS file name.

mdf

* except PROC MIGRATE.

identifies the metadata component file.

dpf
: identifies the partitioned data component files.

p#
: is the partition number.

v#
: is the version number.

fuid
: is the unique file id, which is set to the primary (metadata) path.

idx*suffix*
: identifies the segmented view of an index, where suffix is the name of the index.

hbx*suffix*
: identifies the global view of an index, where suffix is the name of the index.

spds9
: denotes a SAS 9 SPD Engine component file.

Table 2.2 shows the data set component files that are created when you use this LIBNAME statement and DATA step:

```
libname sample spde '/DATA01/mydir'
   datapath=('/DATA01/mydir' '/DATA02/mydir')
   indexpath=('/IDX1/mydir');
data sample.mine(index=(ssn));
   do i=1 to 100000;
   ssn=ranuni(0);
   end;
run;
```

Table 2.2 Data Set Component Files

mine.mdf.0.0.0.spde	metadata file
mine.dpf._DATA01_mydir.0.1.spds9	data file partition #1
mine.dpf._DATA02_mydir.1.1.spds9	data file partition #2
mine.dpf._DATA01_mydir.*n*-1.spds9	data file partition #*n*
mine.dpf._DATA02_mydir.*n*.1.1.spds9	data file partition #*n*+1
mine.hbxssn._DATA01_mydir.0.1.spds9	global index data set for variable SSN
mine.idxssn._DATA01_mydir.0.1.spds9	segmented index data set for variable SSN

Efficient Indexing in the SPD Engine

Indexes can improve the performance of WHERE expression processing and BY expression processing. The SPD Engine enables the rapid creation and update of indexes because it can process these in parallel.

The SPD Engine's indexes are especially suited for data sets of varying sizes and data distributions. These indexes contain both a segment view and a global view of indexed variables' values. This feature allows the SPD Engine to optimally support both queries that require global data views, such as BY expression processing, and queries that require segment views, such as parallel processing of WHERE expressions.

Parallel Index Creation

You can create indexes on your SPD Engine data in parallel, asynchronously. To enable asynchronous parallel index creation, use the "ASYNCINDEX= Data Set Option" on page 36.

Use this option with the DATA step INDEX= option, with PROC DATASETS INDEX CREATE commands, or on the PROC APPEND statement when creating an SPD Engine data set from a Base SAS engine data set that has an index. Each method allows all of the declared indexes to be populated from a single scan of the data set.

The following example shows indexes created in parallel using the DATA step. A simple index is created on variable X and a composite index is created on variables A and B.

```
data foo.mine(index=(x y=(a b)) asyncindex=yes);
   x=1;
   a="Doe";
   b=20;
run;
```

To create multiple indexes in parallel, you must allocate enough utility disk space to create all of the key sorts at the same time. You must also allocate enough memory space. Use "SPDEUTILLOC= System Option" on page 68 to allocate disk space and "SPDEINDEXSORTSIZE= System Option" on page 66 in the configuration file or at invocation to allocate additional memory.

The DATASETS procedure has the flexibility to enable batched parallel index creation by using multiple MODIFY groups. Instead of creating all of the indexes at once, which would require a significant amount of space, you can create the indexes in groups as shown in this example:

```
proc datasets lib=main;
   modify patients(asyncindex=yes);
      index create number;
      index create class;
   run;
   modify patients(asyncindex=yes)'
      index create lastname firstname;
   run;
   modify patients(asyncindex=yes);
      index create fullname=(lastname firstname);
      index create class_sex=(class sex);
   run;
quit;
```

Indexes NUMBER and CLASS are created in parallel, indexes LASTNAME and FIRSTNAME are created in parallel, and indexes FULLNAME and CLASS_SEX are created in parallel.

Parallel Index Updates

The SPD Engine software also supports parallel index updating during data set append operations. Multiple threads enable updates of the data store and index files.

The SPD Engine decomposes a data set append or insert operations into a set of steps that can be performed in parallel. The level of parallelism attained depends on the number of indexes present on the data set. As with parallel index creation, this operation uses memory and disk space for the key sorts that are part of the index append processing. Use system options SPDEINDEXSORTSIZE= to allocate memory and SPDEUTILLOC= to allocate disk space.

PART 2

Reference

Chapter **3**. . . . . **SPD Engine LIBNAME Statement Options** *23*

Chapter **4**. . . . . **SPD Engine Data Set Options** *35*

Chapter **5**. . . . . **SPD Engine System Options** *63*

CHAPTER 3 SPD Engine LIBNAME Statement Options

Introduction to the SPD Engine LIBNAME Statement **23**
SPD Engine LIBNAME Statement Syntax **23**
BYSORT= LIBNAME Statement Option **24**
DATAPATH= LIBNAME Statement Option **26**
ENDOBS= LIBNAME Statement Option **26**
INDEXPATH= LIBNAME Statement Option **28**
METAPATH= LIBNAME Statement Option **28**
PARTSIZE= LIBNAME Statement Option **29**
STARTOBS= LIBNAME Statement Option **31**
TEMP= LIBNAME Statement Option **32**
SPD Engine LIBNAME Statement Options List **33**

Introduction to the SPD Engine LIBNAME Statement

This section contains reference information for all LIBNAME options that are valid for the SPD Engine LIBNAME statement. Some of these LIBNAME options are also data set options. As in the Base SAS engine, data set options take precedence over corresponding LIBNAME options if both options are set.

SPD Engine LIBNAME Statement Syntax

LIBNAME *libref* SPDE '*full-primary-path*' <*options*> ;

libref
: a name that is up to eight characters long and that conforms to the rules for SAS names. You cannot specify TEMP as a libref for a SPD Engine library unless TEMP is not used as an environment variable.

'*full-primary-path*'
: the fully qualified pathname of the primary path for the SPD Engine library. The name must be recognized by the operating environment. Enclose the name in single or double quotation marks. Unless the DATAPATH= and INDEXPATH= options are also specified, the index and data components will also be stored in the same location. The primary path must be unique for each library. Librefs that are different but that reference the same primary path are interpreted to be the same library and can result in lost data.

options

one or more SPD Engine LIBNAME statement options.

Operating Environment Information: A valid library specification and its syntax are specific to your operating environment. For details, see the SAS documentation for your operating environment.

BYSORT= LIBNAME Statement Option

Specifies for the SPD Engine to perform an automatic implicit sort when it encounters a BY statement.

Corresponding data set option: BYSORT=

Affected by data set option: BYNOEQUALS=

Default: YES

Syntax

BYSORT=YES | NO

YES
specifies to implicitly sort the data based on the BY variables whenever a BY statement is encountered, rather than explicitly invoking PROC SORT prior to a BY statement.

NO
specifies not to sort the data based on the BY variables.

Details

DATA or PROC step processing using the default Base SAS engine requires that if there is no index or if the observations are not in order, the data set must be sorted before a BY statement is issued. In contrast, by default the SPD Engine sorts the data returned to the application if the observations are not in order. Unlike PROC SORT, which creates a new sorted data set, the SPD Engine's implicit sort does not change the permanent data set and does not create a new data set. However, utility file space is used. See "SPDEUTILLOC= System Option" on page 68.

The default is BYSORT=YES. A BYSORT=YES argument allows the implicit sort, which outputs the observations in BY group order. If the data set option BYNOEQUALS=YES, then the observations within a group might be output in a different order from the order in the data set. Set BYNOEQUALS=NO to preserve data set order.

The BYSORT=NO argument means that the data must already be ordered on the specified BY variables. This can be the result of a previous explicit sort, an index on the specified variable(s), or the data set having been created in BY variable order. When BYSORT=NO, grouped data is delivered to the application in data set order. The data set option BYNOEQUALS= has no effect when BYSORT=NO.

If you specify the BYSORT= option in the LIBNAME statement, it can be overridden by specifying BYSORT= in the PROC or DATA steps. Therefore, if you set BYSORT=NO in the LIBNAME statement and subsequently a BY statement is encountered, unless your data has been explicitly sorted already, an error will occur.

Set BYSORT=YES in the DATA or PROC step, for input or update opens, to override BYSORT=NO in the LIBNAME statement.

Examples

Example 1: Group Formatting with BYSORT=YES by Default

```
libname growth spde 'D:\SchoolAge';
data growth.teens;
   input Name $ Sex $ Age Height Weight;
datalines;
Alfred M 14 69.0 112.5
Carol F 14 62.8 102.5
James M 13 57.3 83.0
Janet F 15 62.5 112.5
Judy F 14 64.3 90.0
Philip M 16 72.0 150.0
William M 15 66.5 112.0
;

proc print data=growth.teens; by sex;run;
```

Even though the data was not explicitly sorted, no error occurred because BYSORT=YES is the default. The output is shown below.

Output 3.1 Group Formatting with BYSORT=YES by Default

```
                    The SAS System

                         Sex=F

          Obs     Name      Age    Height    Weight

           2      Carol      14     62.8      102.5
           4      Janet      15     62.5      112.5
           5      Judy       14     64.3       90.0


                          Sex=M

          Obs      Name       Age    Height    Weight

           1      Alfred      14     69.0      112.5
           3      James       13     57.3       83.0
           6      Philip      16     72.0      150.0
           7      William     15     66.5      112.0
```

Example 2: Using BYSORT=NO in the LIBNAME Statement In this example, SAS returns an error because BYSORT=YES was not specified on the DATA or PROC steps to override the BYSORT=NO specification on the LIBNAME statement. Whenever implicit sorting is suppressed (BYSORT=NO), the data must be sorted on the BY variable prior to the BY statement, for example by using PROC SORT.

```
libname growth spde 'D:\SchoolAge' bysort=no;
proc print data=growth.teens; by sex;run;

ERROR: Data set GROWTH.TEENS is not sorted in ascending sequence.
       The current by-group has Sex = M and the next by-group has Sex = F.
NOTE: The SAS System stopped processing this step because of errors.
```

DATAPATH= LIBNAME Statement Option

Specifies a list of paths in which to store data partitions (.dpf) for an SPD Engine data set.

Affected by: PARTSIZE= data set or LIBNAME option

Default: the primary path specified on the LIBNAME statement.

Syntax

DATAPATH= ('path1' 'path2'...)

'pathn'
is a fully qualified pathname in single or double quotation marks within parentheses. Separate multiple arguments with spaces.

Details

The SPD Engine creates as many partitions as are needed to store all the data. The size of the partitions is set using the PARTSIZE= option and partitions are created in the paths specified using the DATAPATH= option in a cyclic fashion.

Example

Data partitions are created by cycling through the paths specified on the LIBNAME statement. The first partition is created in /DISK1/DATAFLOW1. The second partition is created in /DISK1/DATAFLOW2. The third partition is created in /DISK1/ DATAFLOW1, and so on.

```
libname mylib spde '/metadisk/metadata'
   datapath=('/disk1/dataflow1' '/disk2/dataflow2');
```

ENDOBS= LIBNAME Statement Option

Specifies the end observation number in a user-defined range of observations to be processed.

Used with: STARTOBS=

Corresponding data set option: ENDOBS=

Default: the last observation in the data set

Restriction: Use ENDOBS= with input data sets only

Syntax

ENDOBS=*n*

n
is the number of the end observation.

Details

By default, the SPD Engine processes all the observations in the data set unless you specify a range of observations using STARTOBS= and ENDOBS= values. If the STARTOBS= option is used without the ENDOBS= option, the implied value of ENDOBS= is the end of the data set. When both options are used together, the value of ENDOBS= must be greater than the value of STARTOBS=.

In contrast to the Base SAS Engine options FIRSTOBS= and OBS=, the STARTOBS= and ENDOBS= SPD Engine options can be used on the LIBNAME statement. (Refer to Chapter 4, "SPD Engine Data Set Options," on page 35 for information on using the ENDOBS= data set option in WHERE processing.)

Example

This example shows that the STARTOBS and ENDOBS options subset the data before the WHERE clause executes. The example prints the four observations that were qualified by the WHERE expression (age >13 in the PRINT procedure) out of the five observations processed from the input data set:

```
libname growth spde 'D:\SchoolAge' endobs=5;
data growth.teens;
   input Name $ Sex $ Age Height Weight;
datalines;
Alfred M 14 69.0 112.5
Carol F 14 62.8 102.5
James M 13 57.3 83.0
Janet F 15 62.5 112.5
Judy F 14 64.3 90.0
Philip M 16 72.0 150.0
William M 15 66.5 112.0
;

proc print data=growth.teens;
   where age >13;
run;
```

```
                      The SAS System

          Obs     Name      Sex     Age     Height     Weight

            1     Alfred     M       14      69.0       112.5
            2     Carol      F       14      62.8       102.5
            4     Janet      F       15      62.5       112.5
            5     Judy       F       14      64.3        90.0
```

INDEXPATH= LIBNAME Statement Option

Specifies a path or list of paths in which to store the two types of index component files (.hbx and .idx) associated with an SPD Engine data set.

Default: the primary path specified on the LIBNAME statement

Syntax

INDEXPATH=(*'path1' 'path2'...*)

'pathn'
is a fully qualified pathname, enclosed in single or double quotation marks, within parentheses. Separate multiple arguments with spaces.

Details

The SPD Engine creates the two index component files in the location specified. When there is not enough space, the index component files overflow into the second file path specified, and so on.

Example

This example creates index component files that span the paths /DISK1/IDXFLOW1 and /DISK1/IDXFLOW2. The index component files will start in /DISK1/IDXFLOW1 and, when that location is full, the index files will overflow to /DISK2/IDXFLOW2.

```
libname mylib spde '/spdedata'
   indexpath=('/disk1/idxflow1' '/disk2/idxflow2');
```

METAPATH= LIBNAME Statement Option

Specifies a list of overflow paths to store metadata (.mdf) component files for an SPD Engine data set.

Default: the primary path specified on the libname statement

Syntax

METAPATH=(*'path1' 'path2'...*)

'pathn'
is a fully qualified pathname, enclosed in single or double quotation marks, within parentheses. Separate multiple arguments with spaces.

Details

The METAPATH= option is specified for space that is exclusively overflow space for the metadata component file. The metadata component file for each data set must begin in the primary path, and overflow occurs to the METAPATH= location when that primary path is full.

Example

This example creates overflow metadata file partitions as needed using the path /DISK1/METAFLOW1.

When /DISK1SPDE is full, the metadata overflows to /DISK1/METAFLOW1.

```
libname mylib spde '/disk1spde'
   metapath=('/disk1/metaflow1');
```

PARTSIZE= LIBNAME Statement Option

Specifies, when an SPD Engine data set is created, the largest size (in megabytes) that the data component partitions must be. This is a fixed size. This specification applies only to the data component files.

Corresponding data set option: PARTSIZE=

Used in conjunction with system option: MINPARTSIZE=

Affected by LIBNAME option: DATAPATH=

Default: 128

Syntax

PARTSIZE=*n*

n
is the size of the partition in megabytes. The maximum value is 2047.

Details

SPD Engine data must be stored in multiple partitions in order for it to be subsequently processed in parallel. Specifying PARTSIZE= forces the software to partition SPD Engine data files at the given size. The actual size of the partition is computed to accommodate the largest number of observations that will fit in the specified size of *n* megabytes. ˇIf you have a table with an observation length greater than 65K, you may find that the PARTSIZE= you specify and the actual partition size do not match. To get these numbers to match, specify a PARTSIZE= that is a multiple of 32 and the observation length.

By splitting (partitioning) the data portion of an SPD Engine data set into fixed-sized files, the software can introduce a high degree of scalability for some operations. The SPD Engine can spawn threads in parallel, up to one thread per partition for WHERE evaluations, for example. Separate data partitions also allow the SPD Engine to process the data without the overhead of file access contention between the threads. Because each partition is one file, the trade-off for small partition size is that an increased number of files (for example, UNIX i-nodes) are required to store the observations.

Scalability limitations using PARTSIZE= depend on how you configure and spread the file systems specified in the DATAPATH= option across striped volumes. (You should spread each individual volume's striping configuration across multiple disk controllers/SCSI channels in the disk storage array.) The goal for the configuration is to maximize parallelism during data retrieval. Refer to Appendix 1, "Quick Guide to the SPD Engine Disk-I/O Set-Up," on page 75 for information on disk striping.

The PARTSIZE= specification is limited by the SPD Engine system option MINPARTSIZE=, which is usually set and maintained by the system administrator. MINPARTSIZE= ensures that an inexperienced user does not arbitrarily create small partitions, thereby generating a large number of files.

The partition size determines a unit of work for many of the parallel operations that require full data set scans, but more partitions does not always mean faster processing. The trade-offs involve balancing the increased number of physical files (partitions) required to store the data set against the amount of work that can be done in parallel by having more partitions. More partitions means more open files in order to process the data set, but a smaller number of observations in each partition. A general rule is to have 10 or fewer partitions per data path, and 3 to 4 partitions per CPU.

To determine an adequate partition size for a new SPD Engine data set, you should be aware of the following:

- the types of applications that will run against the data
- how much data you have
- how many CPUs will be available to the applications
- which disks are available for storing the partitions
- the relationship of these disks to the CPUs.

For example, if each CPU controls only one disk, then an appropriate partition size would be one in which each disk contains approximately the same amount of data. If each CPU controls two disks, then an appropriate partition size would be one in which the load is balanced so that each CPU does approximately the same amount of work. Refer to Appendix 1, "Quick Guide to the SPD Engine Disk-I/O Set-Up," on page 75 for more information on specifying a partition size.

Note: The PARTSIZE= value for a data set cannot be changed after a data set is created. To change the PARTSIZE=, you must re-create the data set and specify a different PARTSIZE= value on the LIBNAME statement or on the new (output) data set. △

Note: Setting PARTSIZE=0 is not recommended. When PARTSIZE=0, the SPD Engine uses the DATAPATH= file systems strictly as overflow space. That is, the SPD Engine creates one partition in the first path and when that file is full, the SPD Engine proceeds to the second path, and so on. △

Example

Using the COPY procedure, extract a set of observations from an existing data set to create a non-indexed data set with a partition size of 32 megabytes:

```
libname sport spde 'conversion_area' partsize=32;
proc copy in=local out=sport;
   select racquets;
run;
```

You have 100 megabytes of data, four CPUs, and one disk per CPU.

Solution: Set the partition size to 8 megabytes. This creates 12.5 partitions (100/8=12.5). Three partitions will be stored on each disk plus a 4-megabyte partition on the first disk. (Remember, partitions are created in cyclical fashion as explained in Chapter 2, "Creating and Loading SPD Engine Files," on page 11.)

You have 100 megabytes of data, four CPUs, and two disks for each CPU as follows: CPU1 controls disk1a and disk1b; CPU2 controls disk2a and disk2b; CPU3 controls disk3a and disk3b; and CPU4 controls disk4a and disk4b.

Solution 1: Set partition size to 8 megabytes. Use the four "a" disks to store the data. This creates three partitions on each disk, plus a 4-megabyte partition on the first disk.

Solution 2: Set partition size to 4 megabytes. Use all eight disks so that each partition is 25 megabytes (100/4=25). This will place four partitions on the first disk and three partitions on the other disks.

STARTOBS= LIBNAME Statement Option

Specifies the starting observation number in a user-defined range of observations to be processed.

Used with: ENDOBS=

Corresponding data set option: STARTOBS=

Default: the first observation in the data set

Restriction: Use STARTOBS= with input data sets only

Syntax

STARTOBS=*n*

n
is the number of the starting observation.

Details

By default, the SPD Engine processes the entire data set unless you specify a range of observations with the STARTOBS= and ENDOBS= options. If the ENDOBS= option is used without the STARTOBS= option, the implied value of STARTOBS= is 1. When both options are used together, the value of STARTOBS= must be less than the value of ENDOBS=.

In contrast to the Base SAS engine options FIRSTOBS= and OBS=, the STARTOBS= and ENDOBS= SPD Engine options can be used on the LIBNAME statement. (Refer to

Chapter 4, “SPD Engine Data Set Options,” on page 35 for information on using the STARTOBS= data set option in WHERE processing.)

Example

This example prints the five observations that were qualified by the WHERE expression (age >13 in PROC PRINT) out of the six observations that were processed starting with the second observation in the data set:

```
libname growth spde 'D:\SchoolAge' startobs=2;
data growth.teens;
   input Name $ Sex $ Age Height Weight;
datalines;
Alfred M 14 69.0 112.5
Carol F 14 62.8 102.5
James M 13 57.3 83.0
Janet F 15 62.5 112.5
Judy F 14 64.3 90.0
Philip M 16 72.0 150.0
William M 15 66.5 112.0
;

proc print data=growth.teens;
   where age >13;
run;
```

Output 3.2 STARTOBS=

```
                          The SAS System

          Obs    Name       Sex    Age    Height    Weight

            2    Carol       F      14     62.8      102.5
            4    Janet       F      15     62.5      112.5
            5    Judy        F      14     64.3       90.0
            6    Philip      M      16     72.0      150.0
            7    William     M      15     66.5      112.0
```

TEMP= LIBNAME Statement Option

Specifies to store the library in a temporary subdirectory of the primary directory.

Default: NO

Syntax

TEMP=YES | NO

YES

specifies to create the temporary subdirectory.

NO
specifies not to create a temporary subdirectory.

Details

The TEMP= option creates a temporary subdirectory of the primary directory named in the LIBNAME statement. The subdirectory and all files contained in it are deleted at the end of the session.

You can use TEMP= in conjunction with the SAS option USER= to create a temporary directory to store interim data sets that can be referenced with a single-level name.

Example

This example illustrates two features: the use of the TEMP= libname option to create a temporary library and the use of the USER= system option to allow the use of single-level table names for SPD Engine tables. A directory is created under **`mydata`**. The MASTERCOPY table has its metadata file stored there. The data and index for MASTERCOPY will be created in the locations specified in the DATAPATH= and INDEXPATH= options, respectively.

```
libname perm <masterdata>
libname mywork spde 'mydata'
   datapath=('/data01/mypath' '/data02/mypath' '/data03/mypath' '/data04/mypath')
   indexpath=('index/mypath') TEMP=YES;

option user=mywork;

data mastercopy (index=(lastname));
   set perm.customer;
   where region='W';

run;
```

SPD Engine LIBNAME Statement Options List

"BYSORT= LIBNAME Statement Option" on page 24
specifies for the SPD Engine to perform an automatic implicit sort when it encounters a BY statement.

"DATAPATH= LIBNAME Statement Option" on page 26
specifies a list of paths in which to store data partitions (.dpf) for an SPD Engine data set.

"ENDOBS= LIBNAME Statement Option" on page 26
specifies the end observation number in a user-defined range of observations to be processed.

"INDEXPATH= LIBNAME Statement Option" on page 28
specifies a path or list of paths in which to store the two index component files (.hbx and .idx) associated with an SPD Engine data set.

"METAPATH= LIBNAME Statement Option" on page 28 specifies a list of overflow paths to store metadata (.mdf) component files for an SPD Engine data set.

"PARTSIZE= LIBNAME Statement Option" on page 29
: specifies, when an SPD Engine data set is created, the size (in megabytes) that the data component partitions must be. This is a fixed-length size. This specification applies only to partitions in the data component files.

"STARTOBS= LIBNAME Statement Option" on page 31
: specifies the starting observation number in a user-defined range of observations to be processed.

"TEMP= LIBNAME Statement Option" on page 32
: specifies to store the library in a temporary subdirectory of the primary directory.

CHAPTER
4

SPD Engine Data Set Options

Introduction to SPD Engine Data Set Options **35**
Syntax **35**
ASYNCINDEX= Data Set Option **36**
BYNOEQUALS= Data Set Option **37**
BYSORT= Data Set Option **39**
COMPRESS= Data Set Option **41**
ENDOBS= Data Set Option **42**
IDXWHERE= Data Set Option **45**
IOBLOCKSIZE= Data Set Option **46**
PADCOMPRESS= Data Set Option **47**
PARTSIZE= Data Set Option **48**
SEGSIZE= Data Set Option **50**
STARTOBS= Data Set Option **51**
SYNCADD= Data Set Option **53**
THREADNUM= Data Set Option **55**
UNIQUESAVE= Data Set Option **56**
WHERENOINDEX= Data Set Option **59**
SPD Engine Data Set Options List **60**
SAS Data Set Options That Behave Differently with the SPD Engine Than with the Base SAS Engine **61**
SAS Data Set Options Not Supported by the SPD Engine **61**

Introduction to SPD Engine Data Set Options

Specifying data set options for the SPD Engine is the same as specifying data set options for the Base SAS engine or SAS/ACCESS engines. This section provides details on SPD Engine-specific data set options. Base SAS engine data set options that affect the SPD Engine are also listed.

When using the options, remember that if a data set option is used subsequent to a LIBNAME option of the same name, the value of the data set option takes precedence.

Syntax

(option-1=value-1 ... option-n=value-n)

Specify a data set option in parentheses after a SAS data set name. To specify several data set options, separate them with spaces.

ASYNCINDEX= Data Set Option

When creating multiple indexes on an SPD Engine data set, specifies to create the indexes in parallel.

Valid in: DATA step and PROC step

Default: NO

Syntax

ASYNCINDEX=YES | NO

YES
  creates the indexes in parallel (asynchronously).

NO
  creates one index at a time (synchronously).

Details

The SPD Engine can create multiple indexes for a data set at the same time. To do this, the SPD Engine spawns a single thread for each index created, then processes the threads simultaneously. Although creating indexes in parallel is much faster than creating one index at a time, the default for this option is NO because parallel creation requires additional utility work space and additional memory, which might not be available. If the index creation fails due to insufficient resources, set the system option to MEMSIZE=0* or increase the size of the utility file space using the SPDEUTILLOC= system option. You increase the memory space used for index sorting using the SPDEINDEXSORTSIZE= system option. If you specify to create indexes in parallel, specify large enough space using the SPDEUTILLOC= system option. See "Space Requirement for Index Creation" on page 83.

Example

The DATASETS procedure has the flexibility to allow batched parallel index creation by using multiple MODIFY groups. Instead of creating all of the indexes at once, which would require a significant amount of space, you can create the indexes in groups as shown in the next example:

```
proc datasets lib=main;
   modify patients(asyncindex=yes);
      index create PatientNo PatientClass;
   run;
   modify patients(asyncindex=yes);
      index create LastName FirstName;
   run;
   modify patients(asyncindex=no);
      index create FullName=(LastName FirstName)
        ClassSex=(PatientClass PatientSex);
```

* for OpenVMS Alpha, increase the paging file quota (PGFLQUO); for OS/390 or z/OS, increase the REGION size.

```
   run;
quit;
```

BYNOEQUALS= Data Set Option

Specifies whether the output order of data set observations with identical values for the BY variable are guaranteed to be in data set order.

Valid in: DATA step and PROC step

Used with: Data set option BYSORT=YES

Default: NO

Syntax

BYNOEQUALS=YES | NO

YES
: does not guarantee that the output order of data set observations with identical values for a BY variable will be in data set order.

NO
: guarantees that the output order of data set observations with identical values for a BY variable will be in data set order.

Details

When a group of observations with identical values in the BY statement is output, the order of the observations in the output will be the same as the data set order, because the default is BYNOEQUALS=NO. By specifying YES, the processing time is decreased, but the observations are not guaranteed to be output in data set order.

The data set or LIBNAME option BYSORT= must be YES (the default), because the BYNOEQUALS= option has no effect when BYSORT=NO.

The following table shows when the SPD Engine preserves physical order in the output.

Table 4.1 SPD Engines Preserves Physical Order

Condition:	Data Set Order Preserved?
If BY is present	YES (BYNOEQUALS=NO and BYSORT=YES by default
If BY is present and BYNOEQUALS=YES	NO
If BY is present and BYSORT=NO	YES (because no implicit sort occurs)

Condition:	Data Set Order Preserved?
If neither BY nor WHERE is present	YES
If WHERE is present	NO

Examples

Example 1: BYNOEQUALS=YES

In this example, the observations with identical BY values on the key variable are output in unpredictable order, because BYNOEQUALS=YES:

```
title 'With BYNOEQUALS=YES'
proc print data=tempdata.housreps(bynoequals=yes);
   by state;
   where state in ('CA' 'TX');
run;
```

The output is shown below.

```
                          With BYNOEQUALS=YES
                               State=CA

      Obs    Representative                   District
       26    Berman, Howard L.                26th
       55    Calvert, Ken                     43d
       60    Capps, Lois                      22d
       76    Cardoza, Dennis                  18th
       22    Becerra, Xavier                  30th
        9    Baca, Joe                        42d
       80    Cox, Christopher                 47th
       38    Bono, Mary                       44th
       89    Cunningham, Randy "Duke"         50th


                              State=TX

      Obs    Representative                   District

       87    Culberson, John Abney            7th
       20    Barton, Joe                      6th
       75    Combest, Larry                   19th
       36    Bonilla, Henry                   23d
        8    Armey, Richard K.                26th
       23    Bentsen, Ken                     25th
       44    Brady, Kevin                     8th
```

Example 2: BYNOEQUALS=NO

This example shows the output with BYNOEQUALS=NO:

```
title 'With BYNOEQUALS=NO';
proc print data=tempdata.housreps(bynoequals=no);
   by state;
   where state in ('CA' 'TX');
run;
```

The output is shown below.

```
                           With BYNOEQUALS=NO

                                 State=CA

         Obs    Representative                    District

           9    Baca, Joe                         42d
          22    Becerra, Xavier                   30th
          26    Berman, Howard L.                 26th
          38    Bono, Mary                        44th
          55    Calvert, Ken                      43d
          60    Capps, Lois                       22d
          76    Cardoza, Dennis                   18th
          80    Cox, Christopher                  47th
          89    Cunningham, Randy "Duke"          50th


                                 State=TX

         Obs    Representative                    District

           8    Armey, Richard K.                 26th
          20    Barton, Joe                       6th
          23    Bentsen, Ken                      25th
          36    Bonilla, Henry                    23d
          44    Brady, Kevin                      8th
          75    Combest, Larry                    19th
          87    Culberson, John Abney             7th
```

BYSORT= Data Set Option

Specifies for the SPD Engine to perform an automatic implicit sort when it encounters a BY statement.

Valid in: DATA step and PROC step

Affects data set option: BYNOEQUALS=

Default: YES

Syntax

BYSORT=YES | NO

YES
specifies to implicitly sort the data based on the BY variables whenever a BY statement is encountered, rather than explicitly invoking the SORT procedure prior to a BY statement.

NO
specifies not to sort the data based on the BY variables. Specifying NO means that the data must already be sorted prior to the BY statement.

Details

DATA or PROC step processing using the default Base SAS engine requires that if there is no index or if the observations are not in order, the data set must be sorted before a BY statement is issued. In contrast, by default the SPD Engine sorts the data returned to the application if the observations are not in order. Unlike PROC SORT, which creates a new sorted data set, the SPD Engine's implicit sort does not change the permanent data set and does not create a new data set. However, utility file space is used. See SPDEUTILLOC= system option in Chapter 5, "SPD Engine System Options," on page 63.

The default is BYSORT=YES. A BYSORT=YES argument allows the implicit sort, which outputs the observations in BY group order. If the data set option BYNOEQUALS=YES, then the observations within a group might possibly be output in a different order from the order in the data set. Set BYNOEQUALS=NO to preserve data set order.

The BYSORT=NO argument means that the data must already be ordered on the specified BY variables. This can be the result of a previous explicit sort, an index on the specified variable(s), or the data set having been created in BY variable order. When BYSORT=NO, grouped data is delivered to the application in data set order. The data set option BYNOEQUALS= has no effect when BYSORT=NO.

If you specify the BYSORT= option in the LIBNAME statement, it can be overridden by specifying BYSORT= in the PROC or DATA steps. Therefore, if you set BYSORT=NO in the LIBNAME statement and subsequently a BY statement is encountered, unless your data has been explicitly sorted already, an error will occur. Set BYSORT=YES in the DATA or PROC step, for input or update opens, to override BYSORT=NO in the LIBNAME statement.

Examples

Example 1: BYSORT=YES by default

Group formatting with BYSORT= YES by default:

```
libname growth spde 'D:\SchoolAge';
data growth.teens;
   input Name $ Sex $ Age Height Weight;
datalines;
Alfred M 14 69.0 112.5
Carol F 14 62.8 102.5
James M 13 57.3 83.0
Janet F 15 62.5 112.5
Judy F 14 64.3 90.0
Philip M 16 72.0 150.0
William M 15 66.5 112.0
;

proc print data=growth.teens; by sex;
run;
```

Even though the data was not explicitly sorted, no error occurred because BYSORT=YES is the default. The output is shown below.

```
                      The SAS System

                          Sex=F
          Obs     Name     Age    Height    Weight

           2      Carol     14     62.8      102.5
           4      Janet     15     62.5      112.5
           5      Judy      14     64.3       90.0


                          Sex=M
          Obs      Name      Age    Height    Weight

           1      Alfred      14     69.0      112.5
           3      James       13     57.3       83.0
           6      Philip      16     72.0      150.0
           7      William     15     66.5      112.0
```

Example 2: BYSORT=NO With BYSORT=NO in the PROC PRINT statement, SAS returns an error whenever implicit sorting is suppressed (BYSORT=NO), the data must be sorted on the BY variable prior to the BY statement, for example by using PROC SORT.

```
libname growth spde 'D:\SchoolAge';
proc print data=growth.teens (bysort=no); by sex;run;

ERROR: Data set GROWTH.TEENS is not sorted in ascending sequence.
       The current by-group has Sex = M and the next by-group has Sex = F.
NOTE: SAS stopped processing this step because of errors.
```

COMPRESS= Data Set Option

Specifies to compress SPD Engine data sets on disk as they are being created.

Valid in: DATA step and PROC step

Related data set options: IOBLOCKSIZE=, PADCOMPRESS=

Default: NO

Syntax

COMPRESS= YES|NO

YES
performs the run-length compression on the data set.

NO
performs no data set compression.

Details

When COMPRESS=YES, the SPD Engine compresses by blocks the data component file as it is created. To specify the number of observations that you want to store in a compressed block, use the data set option IOBLOCKSIZE= when you create the data set. To add padding to the compressed block, specify PADCOMPRESS= when updating the compressed file or when creating the data set.

Unlike the default Base SAS engine, the SPD Engine does not support BINARY or user-specified compression. In addition, if you are migrating a Base SAS engine data set that is both compressed and encrypted, the encryption is preserved but the compression is dropped.

Note: Once a compressed data set is created, you cannot change its block size. To resize the block, you must copy the data set to a new data set, setting IOBLOCKSIZE= to the block size desired for the output data set. Therefore, if the size of the data set is expected to increase, it is recommended that you use the PADCOMPRESS= option if updates in place will occur. △

Using COMPRESS= when creating an SPD Engine data set from a Base SAS engine data set. If you are creating an SPD Engine data set from a compressed Base SAS engine data set, the COPY procedure preserves the compression if the Base SAS engine data set was compressed with YES or CHAR. If the Base SAS engine data set was compressed with BINARY, you must create a new SPD Engine data set using either PROC APPEND or the DATA step and specifying COMPRESS=YES.

Examples

Example 1: Using the DATA Step

```
libname v9lib v9 '.';
libname spdelib spde '.';
data v9lib.a(compress=binary); y=1; run;

data spdelib.a(compress=yes); set v9lib.a; run;
```

Example 2: Using PROC APPEND

```
libname v9lib v9 '.';
libname spdelib spde '.';
   data v9lib.a(compress=binary); y=1; run;

proc append base=spdelib.a(compress=yes); data=v9lib.a; run;
```

ENDOBS= Data Set Option

Specifies the end observation number in a user-defined range of observations to be processed.

Valid in: DATA step and PROC step

Used with data set option: STARTOBS=

Default: the last observation in the data set

Restriction: Use ENDOBS= with input data sets only

Syntax

ENDOBS=*n*

n

is the number of the end observation.

Details

By default, the SPD Engine processes the entire data set unless you specify a range of observations with STARTOBS= or ENDOBS=. If the STARTOBS= option is used without the ENDOBS= option, the implied value of ENDOBS= is the end of the data set. When both options are used together, the value of ENDOBS= must be greater than the value of STARTOBS=.

The ENDOBS data set option in the SPD Engine works the same way as the OBS= data set option in the default Base SAS engine except when specified for a WHERE expression.

Using ENDOBS= with a WHERE Expression In contrast to the Base SAS engine option OBS=, when ENDOBS= is used with WHERE, the ENDOBS= value represents the last observation to process rather than the number of observations to return. The following examples show the difference.

Examples

Example 1: ENDOBS= with SPD Engine A data set is created and processed by the SPD Engine with ENDOBS=5 specified. The WHERE expression is applied to the data set ending with observation number 5. The PRINT procedure prints four observations, which is the set of all observations qualified by the WHERE expression.

```
libname growth spde 'c:\temp';
data growth.teens;
   input Name $ Sex $ Age Height Weight;
   list;
datalines;
Alfred M 14 69.0 112.5
Carol F 14 62.8 102.5
James M 13 57.3 83.0
Janet F 15 62.5 112.5
Judy F 14 64.3 90.0
Philip M 16 72.0 150.0
Zeke M 14 71.1 105.0
Alice F 14 65.1 91.0
William M 15 66.5 112.0
;

proc print data=growth.teens (endobs=5);
   where age >13;
   title 'WHERE age>13 using SPD Engine';
run;
```

Output 4.1 Four Observations Printed

```
                  WHERE age>13 using SPD Engine

     Obs      Name      Sex     Age     Height    Weight

      1      Alfred      M       14      69.0      112.5
      2      Carol       F       14      62.8      102.5
      4      Janet       F       15      62.5      112.5
      5      Judy        F       14      64.3       90.0
```

Example 2: OBS= with the V9 Base SAS Engine The same data set as in Example 1 is processed by the default Base SAS engine with OBS=5 specified. PROC PRINT prints five observations from the set of all observations qualified by the WHERE expression, ending with the 5th qualified observation.

```
libname growth v9 'c:\temp';
data growth.teens;
   input Name $ Sex $ Age Height Weight;
   list;
datalines;
Alfred M 14 69.0 112.5
Carol F 14 62.8 102.5
James M 13 57.3 83.0
Janet F 15 62.5 112.5
Judy F 14 64.3 90.0
Philip M 16 72.0 150.0
Zeke M 14 71.1 105.1
Alice F 14 65.1 91.0
William M 15 66.5 112.0
;

proc print data=growth.teens (obs=5);
   where age >13;
   title 'WHERE age>13 using V9';
run;
```

Output 4.2 Five Observations Printed

```
                  WHERE age >13 using V9

     Obs      Name      Sex     Age     Height    Weight

      1      Alfred      M       14      69.0      112.5
      2      Carol       F       14      62.8      102.5
      4      Janet       F       15      62.5      112.5
      5      Judy        F       14      64.3       90.0
      6      Philip      M       16      72.0      150.0
```

IDXWHERE= Data Set Option

Specifies to use indexes when processing WHERE expressions in the SPD Engine.

Valid in: DATA step and PROC step

Default: YES

Syntax

IDXWHERE=YES | NO

YES
uses indexes when processing WHERE expressions.

NO
ignores indexes when processing WHERE expressions.

Details

IDXWHERE= is primarily a tool to use along with the SPD Engine's WHERE expression planning software called WHINIT to test the performance of index use with WHERE processing in various applications. Set the SAS system option MSGLEVEL=I so the WHERE processing information will be output to the SAS log.

The SPD Engine supports four WHERE-expression evaluation strategies. Three of these, strategies 1, 3, and 4, use available indexes and execute the indexed part of the WHERE expression. Evaluation strategy 2 executes the non-indexed part of the expression.

The first example below, shows that the evaluation strategy 2 is used in the WHERE evaluation because IDXWHERE=NO was specified. The second example shows that the evaluation strategy 1 was used because IDXWHERE=YES was specified.

Examples: WHINIT Log Output (MSGLEVEL=I)

Output 4.3 IDXWHERE=NO

```
34   options msglevel=i;
35   proc means data=permdata.customer(idxwhere=no);
36      var sales;
37      where state="CA";
38   run;


whinit: WHERE (sstate='CA')
whinit returns: ALL EVAL2
NOTE: There were 2981 observations read from the data set PERMDATA.CUSTOMER.
      WHERE state='CA';
```

Output 4.4 IDXWHERE=YES

```
39   proc means data=permdata.customer(idxwhere=yes);
40      var sales;
41      where state="CA";
42   run;


whinit: WHERE (sstate='CA')
 --
whinit: SBM-INDEX STATE uses 45% of segs (WITHIN maxsegratio 75%)
whinit returns: ALL EVAL1(w/SEGLIST)
NOTE: There were 2981 observations read from the data set PERMDATA.CUSTOMER.
      WHERE state='CA';
```

CAUTION:

Do not arbitrarily suppress index use when using both WHERE and BY statements in combination. When you include both a WHERE expression to filter the observations from an SPD Engine data set and a BY expression to order them in a desired way, the filtered observations qualified by the WHERE expression are fed directly into a sort step as part of the parallel WHERE expression evaluation and the final ordered observation set is produced as the result. Index use for WHERE processing greatly improves the filtering performance feeding into the sort step. △

IOBLOCKSIZE= Data Set Option

Specifies the number of observations in a block to be stored in or read from an SPD Engine data component file that is compressed.

Valid in: DATA step and PROC step

Affects data set options: COMPRESS=, PADCOMPRESS=

Default: 4096

Syntax

IOBLOCKSIZE=*n*

n

is the number of observations in the block. This number must be a multiple of 1024.

Details

The software reads and stores compressed observations in a data component file in blocks. IOBLOCKSIZE= specifies how many observations are in the compressed blocks. When you create the SPD Engine data set with COMPRESS=YES specified, the SPD Engine compresses the data component file by blocks as it creates it. To specify the number of observations that you want to store in a compressed block, use the data set option IOBLOCKSIZE= when you create the data set.

The IOBLOCKSIZE= value affects the physical organization of the compressed data component file on disk. Once a compressed data set is created, you cannot change its block size. To resize the block, you must copy the data set to a new data set, setting IOBLOCKSIZE= to the block size desired for the output data set. Since compression is retained when a Base SAS Engine data set is copied to the SPD Engine, you don't have to specify COMPRESS= or IOBLOCKSIZE= unless you want to specify a block size other than the default.

The default is 4096 observations. Specify an IOBLOCKSIZE= value that complements the data to be accessed: access to data that is randomly distributed favors a smaller block size, say 4096 observations, because accessing many smaller blocks is faster than accessing many larger blocks. In contrast, access to data that is uniformly or sequentially distributed or that requires a full data set scan favors a large block size, for example 65,536 observations.

Note: See the PADCOMPRESS= option to add pad space to compressed data sets without changing the compression block size. △

Example

```
/*IOBLOCKSIZE set to 64K */
data sport.maillist(ioblocksize=65536 compress=yes);


/*IOBLOCKSIZE set to 1K */
data sport.maillist(ioblocksize=1024 compress=yes);
```

PADCOMPRESS= Data Set Option

Specifies a number of bytes to add to compression blocks in a data set opened for UPDATE.

Valid in: DATA step and PROC step

Related to data set option: COMPRESS=, IOBLOCKSIZE=

Default: 0

Syntax

PADCOMPRESS= *n*

n
is the number of bytes to add.

Details

Compressed SPD Engine data sets occupy blocks of space on the disk. The number of observations in a block is specified when the data set is created using the IOBLOCKSIZE= data set option. When the data set is updated, it is possible that a new block fragment will need to be created to hold the update. More updates might then create new fragments, which in turn increases the number of I/O operations needed to read a data set.

By increasing the block padding in certain situations where many updates to the data set are expected, fragmentation can be kept to a minimum. However, adding padding can also waste space if you do not update the data set.

You must weigh the cost of padding all compression blocks against the cost of possible fragmentation of some compression blocks.

Specifying the PADCOMPRESS= data set option when you create or update a data set adds space to all of the blocks as they are written back to the disk. The PADCOMPRESS setting is not retained in the data set's metadata.

PARTSIZE= Data Set Option

When an SPD Engine data set is created, specifies the largest size (in megabytes) that the data component partitions must be. This is a fixed size. This specification applies only to the data component files.

Valid in: DATA step and PROC step

Used in conjunction with system option: MINPARTSIZE=

Corresponding LIBNAME Option: PARTSIZE=

Affected by LIBNAME option: DATAPATH=

Default: 128

Syntax

PARTSIZE=*n*

n

is the size of the partition in megabytes. The maximum value is 2047.

Details

Multiple partitions are necessary to read the data in parallel. The option PARTSIZE= forces the software to partition SPD Engine data files at the specified size. The actual size is computed to accommodate the largest number of observations that will fit in the specified size of *n* megabytes. If you have a table with an observation length greater than 65K, you may find that the PARTSIZE= you specify and the actual partition size do not match. To get these numbers to match, specify a PARTSIZE= that is a multiple of 32 and the observation length.

By splitting (partitioning) the data portion of an SPD Engine data set into fixed-sized files, the software can introduce a high degree of scalability for some operations. The SPD Engine can spawn threads in parallel, up to one thread per partition for WHERE evaluations, for example. Separate data partitions also allow the SPD Engine to process the data without the overhead of file access contention between the threads. Because each partition is one file, the trade-off for small partition size is that an increased number of files (for example, UNIX i-nodes) are required to store the observations.

Scalability limitations using PARTSIZE= depend on how you configure and spread the file systems specified in the DATAPATH= option across striped volumes. (You should spread each individual volume's striping configuration across multiple disk controllers/SCSI channels in the disk storage array.) The goal for the configuration, at the hardware level, is to maximize parallelism during data retrieval. For more information about disk-striping, see Appendix 1, "Quick Guide to the SPD Engine Disk-I/O Set-Up," on page 75.

The PARTSIZE= specification is limited by the SPD Engine system option MINPARTSIZE=, which is usually maintained by the system administrator. MINPARTSIZE= ensures that an inexperienced user does not arbitrarily create small partitions, thereby generating a large number of data files.

The partition size determines a unit of work for many of the parallel operations that require full data set scans. But more partitions does not always mean faster processing. The trade-offs involve balancing the increased number of physical files (partitions) required to store the data set versus the amount of work that can be done in parallel by having more partitions. More partitions means more open files in order to process the data set, but a smaller number of observations in each partition. A general rule is to have 10 or fewer partitions per data path, and 3 to 4 partitions per CPU. (Some operating systems have a limit on the number of open files allowed.)

To determine an adequate partition size for a new SPD Engine data set, you should be aware of the following:

- the types of applications that will run against the data
- how much data you have
- how many CPUs will be available to the applications
- which disks are available for storing the partitions
- the relationship of these disks to the CPUs.

For example, if each CPU controls only one disk, then an appropriate partition size would be one in which each disk contains approximately the same amount of data. If each CPU controls two disks, then an appropriate partition size would be one in which the load is balanced so that each CPU does approximately the same amount of work. Refer to Appendix 1, "Quick Guide to the SPD Engine Disk-I/O Set-Up," on page 75 for more information on specifying a partition sizes.

Note: The PARTSIZE= value for a data set cannot be changed after a data set is created. To change the PARTSIZE=, you must re-create the data set and specify a different PARTSIZE= value on the LIBNAME statement or on the new (output) data set. △

Note: Setting PARTSIZE=0 is not recommended. When PARTSIZE=0, the SPD Engine uses the DATAPATH= file systems strictly as overflow space. That is, the SPD Engine creates one partition in the first path and when that file is full, the SPD Engine proceeds to the second path, and so on. △

Example: Using PROC SQL

Using the COPY procedure, extract a set of observations from an existing data set to create a non-indexed data set with a partition size of 32 megabytes:

```
libname spdecen spde 'D:\CensusData';
proc sql;
   create data set spdecen.hr80spde (partsize=32)
   as
   select state,age,sex,hour89,industry,occup
   from spde cen.precs where hour89 > 40;
quit;
```

You have 100 megabytes of data, four CPUs, and one disk per CPU.

Solution: Set the partition size to 8 megabytes. This creates 12.5 partitions (100/8=12.5). Three partitions are stored on each disk plus a 4-megabyte partition on the first disk. (Remember, partitions are created in cyclical fashion as explained in Chapter 2, "Creating and Loading SPD Engine Files," on page 11.)

You have 100 megabytes of data, four CPUs, and two disks per CPU as follows: CPU1 controls disk1a and disk1b; CPU2 controls disk2a and disk2b; CPU3 controls disk3a and disk3b; and CPU4 controls disk4a and disk4b.

Solution 1: Set partition size to 8 megabytes. Use the four "a" disks to store the data. This creates three partitions on each disk, plus a 4-megabyte partition on the first disk.

Solution 2: Set partition size to 4 megabytes. Use all eight disks so each partition is 25 megabytes (100/4=25). This creates four partitions on the first disk and three on the other disks.

SEGSIZE= Data Set Option

Specifies the number of observations to use as the segment size for indexes in an SPD Engine data set.

Valid in: DATA step and PROC step

Default: 8192

Syntax

SEGSIZE=*n*

n

is the number of observations to include in an index file segment; *n* must be a multiple of 1024. The minimum SEGSIZE= value is 1024 observations.

Details

The segment is the logical portion of a table that is accessible to a WHERE thread as a unit of work. For example, a segment size of 8192 will logically divide the table into segments containing rows 1–8192, 8193–16384, and so on.

The size of the index segment determines the structure of the index file and cannot be changed after the SPD Engine data set is created.

Note: Tests show that modifying the size of the index segment does not significantly increase performance. △

Example

This DATA statement specifies a segment size of 65,536 observations for the index component of the data set MYLIB.MYDATA:

```
data mylib.mydata (segsize=65536);
```

STARTOBS= Data Set Option

Specifies the starting observation number in a user-defined range of observations to be processed.

Valid in: DATA step and PROC step

Default: the first observation in the data set

Restriction: STARTOBS= should not be used with the OBS= Base SAS engine data set option

Restriction: Use STARTOBS= with input data sets only

Syntax

STARTOBS=*n*

n
is the number of the starting observation.

Details

By default, the SPD Engine processes the entire data set unless you specify a range of observations with the STARTOBS= and ENDOBS= options. If the ENDOBS= option is used without the STARTOBS= option, the implied value of STARTOBS= is 1. When both options are used together, the value of STARTOBS= must be less than the value of ENDOBS=.

The STARTOBS= data set option used in the SPD Engine works the same way as the FIRSTOBS= data set option used with the default Base SAS engine except when specified for a WHERE expression.

Using STARTOBS= with a WHERE Expression When STARTOBS= is used with WHERE, the STARTOBS= value represents the first observation on which to apply the WHERE expression. Compare this to the default Base SAS engine data set option FIRSTOBS=, which specifies the starting observation number within the subset of data qualified by the WHERE expression.

Examples

Example 1: STARTOBS= with SPD Engine A data set is created and processed by the SPD Engine with STARTOBS=5 specified. The WHERE expression is applied to the data set beginning with observation number 5. PROC PRINT prints six observations, which is the set of observations qualified by the WHERE expression.

```
libname growth spde 'c:\temp';
data growth.teens;
```

```
   input Name $ Sex $ Age Height Weight;
   list;
datalines;
Alfred M 14 69.0 112.5
Carol F 14 62.8 102.5
James M 13 57.3 83.0
Janet F 15 62.5 112.5
Judy F 14 64.3 90.0
Philip M 16 72.0 150.0
Zeke M 14 71.1 105.1
Alice F 14 65.1 91.0
William M 15 66.5 112.0
Mike M 16 67.0 105.1
;

proc print data=growth.teens (startobs=5);
   where age >13;
   title 'WHERE age>13 using SPD Engine';
run;
```

Output 4.5 Six Observations Printed

```
                WHERE age>13 using SPD Engine

     Obs    Name       Sex    Age    Height    Weight

       5    Judy        F      14     64.3       90.0
       6    Philip      M      16     72.0      150.0
       7    Zeke        M      14     71.1      105.1
       8    Alice       F      14     65.1       91.0
       9    William     M      15     66.5      112.0
      10    Mike        M      16     67.0      105.1
```

Example 2: FIRSTOBS= with the Default Base SAS Engine The same data set as in Example 1 is processed by the default Base SAS engine with FIRSTOBS=5 specified. PROC PRINT prints five observations from the set of all observations qualified by the WHERE expression, starting with the 5th qualified observation. FIRSTOBS= is not supported in the SPD Engine.

```
libname growth v9 'c:\temp';
data growth.teens;
   input Name $ Sex $ Age Height Weight;
   list;
datalines;
Alfred M 14 69.0 112.5
Carol F 14 62.8 102.5
James M 13 57.3 83.0
Janet F 15 62.5 112.5
Judy F 14 64.3 90.0
Philip M 16 72.0 150.0
Zeke M 14 71.1 105.1
Alice F 14 65.1 91.0
William M 15 66.5 112.0
Mike M 16 67.0 105.1
;
```

```
proc print data=growth.teens (firstobs=5);
   where age >13;
   title 'WHERE age>13 using the V9 Engine';
run;
```

Output 4.6 Six Observations Printed

```
                  WHERE age>13 using the V9 Engine

     Obs    Name       Sex    Age    Height    Weight

       5    Judy        F      14     64.3       90.0
       6    Philip      M      16     72.0      150.0
       7    Zeke        M      14     71.1      105.1
       8    Alice       F      14     65.1       91.0
       9    William     M      15     66.5      112.0
      10    Mike        M      16     67.0      105.1
```

SYNCADD= Data Set Option

Specifies to process one observation at a time or multiple observations at a time.

Valid in: PROC SQL

Affects the data set option: UNIQUESAVE=

Default: NO

Syntax

SYNCADD=YES | NO

YES
processes a single observation at a time (synchronously).

NO
processes multiple observations at a time (asynchronously).

Details

When SYNCADD=YES, observations are processed one at a time. With PROC SQL, if you are adding observations to a data set with a unique index, then when the SPD Engine encounters an observation with a non-unique value, the add operation is aborted, all transactions just added are backed out, and the original data set on disk is unchanged.

When SYNCADD=NO, observations are added in blocks (pipelining), which is usually faster. If you are adding observations to a data set with a unique index and the SPD Engine encounters a observation with a duplicate index value, the SPD Engine rejects the observation but continues processing. A status code is issued only at the end of the append or insert operation.

To save the rejected observations in a separate data set, set the UNIQUESAVE= data set option to YES.

Example

In this example, two data sets, UQ01A and UQ01B, are created. On UQ01A, PROC SQL creates a unique composite index and then inserts new values into the data set with SYNCADD=NO (inserting blocks of data). Duplicates are stored in a separate file because UNIQUESAVE= is set to YES.

Then PROC SQL creates a unique composite index on UQ01B and inserts new values with SYNCADD=YES. SQL stops when duplicate values are encountered and restores the data set. (Version 8 behavior). Even though UNIQUESAVE=YES, it is ignored. The SAS log is shown below:

```
1097  libname userfile spde 'c:\temp';
NOTE: Libref SPDS USERFILE was successfully assigned as follows:
      Engine:        SPD Engine
      Physical Name: d3727.na.sas.com:528c:\temp\
1098
1099  data uq01a uq01b;
1100    input z $ 1-20 x y;
1101    list;
1102    datalines;

RULE:----+----1----+----2----+----3----+----4----+----5----+----6----+----7
1103        one                  1 10
1104        two                  2 20
1105        three                3 30
1106        four                 4 40
1107        five                 5 50
NOTE: The data set USER.UQ01A has 5 observations and 3 variables.
NOTE: The data set USER.UQ01B has 5 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.51 seconds
      cpu time            0.06 seconds


1108  ;
1109
1110
1111  proc sql    sortseq=ascii exec noerrorstop;
1112  create unique index comp
1113    on uq01a  (x, y);
NOTE: Composite index comp has been defined.
1114  insert into uq01a(syncadd=no,uniquesave=yes)
1115    values('rollback1', -80, -80)
1116    values('rollback2',-90, -90)
1117    values('nonunique', 2, 20)
1118   ;
NOTE: 3 observations were inserted into USER.UQ01A.

WARNING: Duplicate values not allowed on index comp for file USER.UQ01A.
         (Occurred 1 times.)
NOTE: Duplicate records have been stored in file USER._D2DAAF7.
```

```
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.99 seconds
      cpu time            0.05 seconds


1119  proc sql    sortseq=ascii exec noerrorstop;
1120  create unique index comp
1121    on uq01b  (x, y);
NOTE: Composite index comp has been defined.
1122  insert into uq01b(syncadd=yes,uniquesave=yes)
1123    set z='rollback3', x=-60, y=-60
1124    set z='rollback4', x=-70, y=-70
1125    set z='nonunique', x=2, y=20;
ERROR: Duplicate values not allowed on index comp for file UQ01B.
NOTE: Deleting the successful inserts before error noted above to restore
      data set to a consistent state.
1126
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.26 seconds
      cpu time            0.17 seconds


1127  proc compare data=uq01a compare=uq01b;run;

NOTE: There were 7 observations read from the data set USER.UQ01A.
NOTE: There were 5 observations read from the data set USER.UQ01B.
NOTE: PROCEDURE COMPARE used (Total process time):
      real time           0.51 seconds
      cpu time            0.05 seconds
```

THREADNUM= Data Set Option

Specifies the number of I/O threads the SPD Engine can spawn for processing an SPD Engine data set.

Valid in: DATA step and PROC step

Affected by system option: SPDEMAXTHREADS=

Default: The value of the SPDEMAXTHREADS= system option, if set; otherwise, the default is 2 times the number of CPUs on your machine.

Syntax

THREADNUM=*n*

n

specifies the number of threads.

Details

THREADNUM= allows you to specify the maximum number of I/O threads that the SPD Engine will spawn for processing a data set. The THREADNUM= value applies to any SPD Engine I/O processing, including:

- WHERE expression processing
- parallel index creation
- I/O requested by thread-enabled applications.

Adjusting THREADNUM= enables the system administrator to adjust the level of CPU resources the SPD Engine can use for any process. For example in a 64-bit processor system, setting THREADNUM=4 limits the process to at most four CPUs, thereby enabling greater throughput for other users or applications.

When THREADNUM= is greater than 1, parallel processing is likely to occur and therefore, physical order might not be preserved in the output.

You can also use this option to explore scalability for WHERE expression evaluations.

SPDEMAXTHREADS=, a configurable system option, imposes an upper limit on the consumption of system resources, and therefore constrains the THREADNUM= value.

Note: The SAS system option NOTHREADS does not affect the SPD Engine. △

Note: Setting THREADNUM= to 1 means no parallel processing will occur, which is behavior consistent with the default Base SAS engine. △

Example

The SPD Engine system option SPDEMAXTHREADS= is set to 128 for the session. Explore the effects of parallelism on a given query by using a SAS macro such as the following:

```
%macro dotest(maxthr);
%do nthr=1 %to &maxthr

data _null_;
set spde cen.precs(threadnum= &nthr);
   where occup= '022'
   and state in('37','03','06','36');
run
%mend dotest;
```

UNIQUESAVE= Data Set Option

Specifies to save observations with non-unique key values (the rejected observations) to a separate data set when appending or inserting observations to data sets with unique indexes.

Valid in: PROC APPEND and PROC SQL

Affected by the data set option: SYNCADD=NO

Used in conjunction with automatic macro variable: SPSUSDS

Default: NO

Syntax

UNIQUESAVE=YES | NO

YES
if SYNCADD=NO, writes rejected observations to a separate, system-created data set, which can be accessed by a reference to the macro variable SPDSUSDS.

NO
does not write rejected observations to a separate data set.

Details

Use UNIQUESAVE=YES when you are adding observations to a data set with unique indexes and the data set option SYNCADD=NO is set.

SYNCADD=NO specifies for an append or insert operation to process observations in blocks (pipelining) rather than one at a time. Duplicate index values are detected only after all the observations are applied to a data set. With UNIQUESAVE=YES, the rejected observations are saved to a separate data set whose name is stored in the SPD Engine macro variable SPDSUSDS. You can then specify the macro variable in place of the data set name to identify the rejected observations.

Note: When SYNCADD=YES, the UNIQUESAVE= option is ignored. See the SYNCADD= data set option for more information. △

Examples

In this example, two data sets with unique indexes on the variable "name" are created and then appended together using PROC APPEND with UNIQUESAVE=YES. The SAS log is shown below:

```
1    libname employee spde 'c:\temp';
NOTE: Libref EMPLOYEE was successfully assigned as follows:
      Engine:        SPD Engine
      Physical Name: c:\temp\
2    data employee.emp1 (index=(name/unique));
3    input name $ exten;
4    list; datalines;

RULE:----+----1----+----2----+----3----+----4----+----5----+----6----+
5          Jill 4344
6          Jack 5589
7          Jim 8888
8          Sam 3334
NOTE: The data set EMPLOYEE.EMP1 has 4 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           9.98 seconds
      cpu time            1.28 seconds


9    run;

10   data employee.emp2 (index=(name/unique));
11             input name $ exten;
12             list; datalines;

RULE:----+----1----+----2----+----3----+----4----+----5----+----6----+
13                   Jack 4443
```

```
14                    Ann 8438
15                    Sam 3334
16                    Susan 5321
17                    Donna 3332
NOTE: The data set EMPLOYEE.EMP2 has 5 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.04 seconds
      cpu time            0.04 seconds


18              run;

19   proc append data=employee.emp2 base=employee.emp1
20                 (syncadd=no uniquesave=yes);
21              run;

NOTE: Appending EMPLOYEE.EMP2 to EMPLOYEE.EMP1.
NOTE: There were 5 observations read from the data set EMPLOYEE.EMP2.
NOTE: 3 observations added.
NOTE: The data set EMPLOYEE.EMP1 has 7 observations and 2 variables.
WARNING: Duplicate values not allowed on index name for file
         EMPLOYEE.EMP1. (Occurred 2 times.)
NOTE: Duplicate records have been stored in file EMPLOYEE._D3596FF.
NOTE: PROCEDURE APPEND used (Total process time):
      real time           6.25 seconds
      cpu time            1.26 seconds


22   proc print data=employee.emp1;
23              title 'Listing of Final Data Set';
24              run;

NOTE: There were 7 observations read from the data set EMPLOYEE.EMP1.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           2.09 seconds
      cpu time            0.40 seconds


25
26   proc print data=&spdsusds;
27              title 'Listing of Rejected observations';
28              run;

NOTE: There were 2 observations read from the data set EMPLOYEE._D3596FF.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

Output 4.7 UNIQUESAVE=YES

```
                  Listing of Final Data Set

                      Obs    name    exten
                       1     Jill     4344
                       2     Jack     5589
                       3     Jim      8888
                       4     Sam      3334
                       5     Ann      8438
                       6     Susan    5321
                       7     Donna    3332

               Listing of Rejected observations""

               Obs    name    exten    XXX00000
                1     Jack     4443      name
                2     Sam      3334      name
```

WHERENOINDEX= Data Set Option

Specifies, when making WHERE expression evaluations, a list of indexes to exclude.

Valid in: DATA step and PROC step

Default: blank

Syntax

WHERENOINDEX=(*name1 name2...*)

(name1 name2...)
a list of index names that you wish to exclude from the WHERE planner.

Example

The data set PRECS is defined with indexes:

```
proc datasets lib=spde cen
   modify precs;
   index create stser=(state serialno) occind=(occup industry) hour89;
quit;
```

When evaluating the next query, we want the SPD Engine not to use the indexes for either the STATE and HOUR89 variables.

In this case, we know that our AND combination of the conditions for the OCCUP and INDUSTRY variables will produce a very small yield. Few observations satisfy the respective conditions. To avoid the extra index I/O (machine time) that the query requires for a full-indexed evaluation, use the following SAS code:

```
proc sql;
   create data set hr80spde
   as select state, age, sex, hour89, industry, occup from spde cen.precs
```

```
      (wherenoindex=(stser hour89))
    where occup='022'
    and state in('37','03','06','36')
    and industry='012'
    and hour89 > 40;
  quit;
```

Note: Specify the index names in the WHERENOINDEX list, not the variable names. In the example, both the composite index for the STATE variable, STSER, and the simple index, HOUR89, are excluded from consideration. △

SPD Engine Data Set Options List

"ASYNCINDEX= Data Set Option" on page 36
: specifies to create indexes in parallel.

"BYNOEQUALS= Data Set Option" on page 37
: specifies the index output order of data set observations with identical values for the BY variable.

"BYSORT= Data Set Option" on page 39
: specifies for the SPD Engine to perform an automatic implicit sort when it encounters a BY statement.

"COMPRESS= Data Set Option" on page 41
: compresses data sets on disk.

"ENDOBS= Data Set Option" on page 42
: specifies the ending observation number in a user-defined range for WHERE expressions.

"IDXWHERE= Data Set Option" on page 45
: controls using indexes for WHERE processing.

"IOBLOCKSIZE= Data Set Option" on page 46
: specifies the number of observations in a block.

"PADCOMPRESS= Data Set Option" on page 47
: specifies a number of bytes to add to compression blocks in a data set opened for UPDATE.

"PARTSIZE= Data Set Option" on page 48
: specifies the partition size of the data component files. This is also a LIBNAME option.

SEGSIZE=
: specifies the number of observations to use as the segment size for indexes in an SPD Engine data set.

"STARTOBS= Data Set Option" on page 51
: specifies the starting observation number in a user-defined for WHERE expressions.

"SYNCADD= Data Set Option" on page 53
: specifies to append one observation or a block of observations at a time.

"THREADNUM= Data Set Option" on page 55
: specifies the number of threads to use for the SPD Engine processing.

"UNIQUESAVE= Data Set Option" on page 56 specifies to save in a separate file any observations that were rejected due to non-unique key values during an append to a data set with unique indexes when SYNCADD=NO.

"WHERENOINDEX= Data Set Option" on page 59
specifies a list of indexes to exclude for WHERE evaluations.

SAS Data Set Options That Behave Differently with the SPD Engine Than with the Base SAS Engine

CNTLLEV=
Only the value MEM is accepted.

COMPRESS=
Only YES and NO values are accepted.

MSGLEVEL=I
Produces WHERE optimization information in the SAS log.

SAS Data Set Options Not Supported by the SPD Engine

- BUFNO=
- DLDMGACTION=
- ENCODING=
- GENMAX=
- GENNUM=
- IDXNAME=
- OUTREP=
- POINTOBS=
- REUSE=
- TOBSNO=

CHAPTER 5 SPD Engine System Options

Introduction to SPD Engine System Options **63**
Syntax **63**
  COMPRESS= System Option **64**
  MAXSEGRATIO= System Option **64**
  MINPARTSIZE= System Option **66**
  SPDEINDEXSORTSIZE= System Option **66**
  SPDEMAXTHREADS= System Option **67**
  SPDESORTSIZE= System Option **68**
  SPDEUTILLOC= System Option **68**
  SPDEWHEVAL= System Option **69**
SPD Engine System Options List **71**
SAS System Options That Behave Differently with SPD Engine **71**

Introduction to SPD Engine System Options

SAS system options are instructions that affect your SAS session. They control the way that SAS performs operations such as SAS system initialization, hardware and software interfacing, and the input, processing, and output of jobs and SAS files. The SPD Engine system options work the same way as other SAS system options. This section discusses SPD Engine-specific system options and Base SAS system options that behave differently with the SPD Engine.

Syntax

OPTIONS *option(s)*;

where

option
: specifies one or more SPD Engine system options you want to change.

The following example specifies the SPD Engine system option MAXSEGRATIO=:

```
options maxsegratio=50;
```

Note: *Operating Environment Information:* On the command line or in a configuration file, the syntax is specific to your operating environment. For details, see the SAS documentation for your operating environment. △

COMPRESS= System Option

Specifies to compress the SPD Engine data sets on disk as they are being created.

Valid in: configuration file, SAS invocation, OPTIONS statement, System Options window

Default: NO

Syntax

COMPRESS= YES | NO

YES
performs the run-length compression on the data set.

NO
performs no data set compression.

Details

When COMPRESS=YES, the SPD Engine compresses by blocks the data component file as it is created. To specify the number of observations that you want to store in a compressed block, use the data set option IOBLOCKSIZE= when you create the data set. To add padding to the compressed block, specify PADCOMPRESS= when updating the compressed file or when creating the data set.

Unlike the default Base SAS engine, the SPD Engine does not support BINARY or user-specified compression. In addition, if you are migrating a Base SAS engine data set that is both compressed and encrypted, the encryption is preserved but the compression is dropped.

Note: Once a compressed data set is created, you cannot change its block size. To resize the block, you must copy the data set to a new data set, setting IOBLOCKSIZE= to the block size desired for the output data set. Therefore, if the size of the data set is expected to increase, it is recommended that you use the PADCOMPRESS= option, if updates in place will occur. △

Using COMPRESS= when creating an SPD Engine data set from a Base SAS engine data set. If you are creating an SPD Engine data set from a compressed Base SAS engine data set, the COPY procedure preserves the compression if the Base SAS engine data set was compressed with YES or CHAR. If the Base SAS engine data set was compressed with BINARY, you must create a new SPD Engine data set using either PROC APPEND or the DATA step and specify COMPRESS=YES.

MAXSEGRATIO= System Option

When evaluating a WHERE expression that contains indexed variables, controls what percentage of index segments to identify as candidate segments before processing the WHERE expression.

Valid in: configuration file, SAS invocation, OPTIONS statement, System Options window

Affected by data set option: SEGSIZE=

Default: 75

Syntax

MAXSEGRATIO=*n*

n

specifies an upper limit for the percentage of index segments that the SPD Engine identifies as containing the value referenced in the WHERE expression. The default is 75, which specifies for the SPD Engine to use the index to identify segments that contain the particular WHERE expression value and to stop identifying candidate segments when more than 75% of all segments are found to contain the value.

The range of valid values is integers between 0–100. If n=0, the SPD Engine does not try to identify candidate segments but instead applies the WHERE expression to all segments. If n=100, the SPD Engine checks 100% of the segments in order to identify candidate segments and then applies the WHERE expression only to those candidate segments.

Details

For WHERE queries on indexed variables, the SPD Engine can first determine the number of index segments that contain one or more variable values that match one or more of the conditions in the WHERE expression. Often a substantial performance gain can be realized if the WHERE expression is applied only to the segments that contain observations satisfying the WHERE expression.

The SPD Engine uses the value of MAXSEGRATIO= to determine at what point the cost of applying the WHERE expression to every segment would be less than the cost of continuing to identify candidate segments. When the calculated ratio exceeds the ratio specified in MAXSEGRATIO=, the SPD Engine stops the process of identifying candidate segments and instead applies the WHERE expression to all segments.

Note: For a few tables, 75 percent *might not* be the optimal setting. To determine a better setting, run a performance benchmark, adjust the percentage, and rerun the performance benchmark. Comparing results will show you how the specific data population you are querying responds to shifting the index-segment ratio. △

Example

The following specification causes the SPD Engine to begin identifying index segments that might satisfy the WHERE expression until the percentage of identified segments, compared to the total number of segments, exceeds 65. If the percentage exceeds 65, the SPD Engine stops identifying candidate segments and simply applies the WHERE expression to all segments:

```
options maxsegratio=65;
```

The following specification causes the SPD Engine to apply the WHERE expression to all segments without first identifying any candidate segments:

```
options maxsegratio=0;
```

The following specification causes the SPD Engine to begin identifying index segments and to not stop until it has pre-evaluated all segments (100%). Then, the WHERE expression is applied to all candidate segments that were identified.

```
options maxsegratio=100;
```

MINPARTSIZE= System Option

Specifies a minimum partition size to use for creating SPD Engine data sets.

Valid in: configuration file, SAS invocation

Related to: PARTSIZE= data set and LIBNAME option

Default: 0

Syntax

MINPARTSIZE=*n* | *n*K | *n*M | *n*G

n
specifies the minimum partition size in bytes, kilobytes, megabytes, or gigabytes, respectively. The upper limit for the minimum partition size is 2047 megabytes.

Details

Specifying MINPARTSIZE= sets a lower limit to the partition size that can be specified with the PARTSIZE= option. The MINPARTSIZE= specification could affect whether the partitions are created with approximately the same number of observations. A small partition size means more open files during processing. Your operating system might have a limit on the number of open files allowed.

SPDEINDEXSORTSIZE= System Option

Specifies the size of memory space that the sorting utility can use when sorting values for creating an index.

Valid in: configuration file, SAS invocation, OPTIONS statement, Systems Options window

Affected by data set option: MEMSIZE=

Default: 32 megabytes

Syntax

SPDEINDEXSORTSIZE=*n* | *n*K | *n*M | *n*G

n

specifies the amount of memory in bytes, kilobytes, megabytes, or gigabytes, respectively. If *n*=0, the sort utility uses its default. The valid value range is from 1,048,576 to 10,736,369,664 bytes.

Details

The SPDEINDEXSORTSIZE= option specifies the maximum amount of memory that can be used for sorting when creating an index. When indexes are created in parallel (because ASYNCINDEX=YES), the value you specify in SPDEINDEXSORTSIZE= is divided up among all the concurrent index creation threads.

If the index creation fails due to insufficient memory, restart SAS with the system option MEMSIZE=0*, or increase the size of the utility file space using the SPDEUTILLOC= system option. You increase the memory space used for index sorting using the SPDEINDEXSORTSIZE= system option. If you specify to create indexes in parallel, specify large enough space using the SPDEUTILLOC= system option.

SPDEMAXTHREADS= System Option

Specifies the upper limit on the number of threads that the SPD Engine can spawn for I/O processing.

Valid in: configuration file, SAS invocation

Default: 0

Syntax

SPDEMAXTHREADS=*n*

n

the maximum number of threads the SPD Engine can spawn. The range of valid values is 0–65,536. The default is zero, which means that the SPD Engine uses the value of THREADNUM= if set; otherwise, the SPD Engine sets the number of threads to spawn as equivalent to two times the number of CPUs on your machine.

Details

Specifying SPDEMAXTHREADS= sets an upper limit on the number of threads to spawn for the SPD Engine processing, including WHERE expression processing,

* for OpenVMS Alpha, increase the paging file quota (PGFLQUO); for OS/390 or z/OS, increase the REGION size.

parallel index creation, and any I/O processing requested by thread-enabled applications such as SAS thread-enabled procedures. SPDEMAXTHREADS= constrains the THREADNUM= data set option.

SPDESORTSIZE= System Option

Specifies the size of memory space needed for sorting operations used by the SPD Engine.

Valid in: configuration file, SAS invocation, OPTIONS statement, System Options window

Default: 32 megabytes

Syntax

SPDESORTSIZE=*n* | *n*K | *n*M | *n*G

n

specifies the amount of memory in bytes, kilobytes, megabytes, or gigabytes, respectively. If n=0, the sort utility uses its default. The range of valid values is from 1,048,576 to 10,736,369,664 bytes.

Details

Because the SPD Engine can perform the implicit sort in parallel, the sort size you specify for SPDESORTSIZE= should be multiplied by the number of processes that will be in parallel; this total should be less than the physical memory available to your process. Proper specification of SPDESORTSIZE= can improve performance by restricting the swapping of memory that is controlled by the operating environment.

If the sort process needs more memory than you specify, restart SAS with the system option MEMSIZE=0* or increase the size of the utility file space using the SPDEUTILLOC= system option. You increase the memory space used for index sorting using the SPDEINDEXSORTSIZE= system option. If you specify to create indexes in parallel, specify large enough space using the SPDEUTILLOC= system option.

Note: The SORTSIZE= option documented for the Base SAS engine affects PROC SORT operations. The SPDESORTSIZE= specification affects sorting operations specific to the SPD Engine. △

SPDEUTILLOC= System Option

Specifies one or more file system locations in which the SPD Engine can temporarily store utility files.

* for OpenVMS Alpha, increase the paging file quota (PGFLQUO); for OS/390 or z/OS, increase the REGION size.

Valid in: configuration file or SAS invocation

Syntax

SPDEUTILLOC= *location* | *(location-1 ...location-n)*

location
an existing directory where the utility files are created.

(location-1 ...location-n)
a series of existing directories where the utility files are created. Utility files are partitioned at file size (limit) of 2 gigabytes.

Note: *Location* can be enclosed in single or double quotation marks. Quotation marks are required if *location* contains embedded blanks. △

Details

The SPD Engine creates temporary utility files during certain processing, such as implicit sorting and creating indexes. To successfully complete the process, you must be sure that enough space exists to store the utility files. The SPDEUTILLOC= system option allows you to specify an adequate amount of space for processing. However, for the OpenVMS Alpha operating environment, the libraries must be ODS-5 files.

SAS recommends that you always specify SPDEUTILLOC= to ensure you have enough space for processes that create utility files. If SPDEUTILLOC= is not specified, each operating environment has one or more default locations. The following table shows the default utility file locations.

Table 5.1 Default Utility File Locations

Operating Environment	Default Location 1	Else, Default Location 2
UNIX	UTILLOC= SAS system option, if specified	SASWORK
Windows	UTILLOC= SAS system option, if specified	SASWORK
z/OS	UTILLOC= SAS system option, if specified	SASWORK
OpenVMS Alpha	UTILLOC= SAS system option, if specified	WORK= SAS system option, if specifies an ODS-5 directory.[1]

1 If the WORK= SAS system option does not specify an ODS-5 directory, and if the SAS session was started with an ODS-5 file specification of SASROOT, the utility files will be created in the SASROOT directory. Otherwise, there will be no default location, and the LIBNAME assignment will fail.

SPDEWHEVAL= System Option

Specifies the process used to determine which observations meet the condition(s) of a WHERE expression.

Valid in: configuration file, SAS invocation

Default: COST

Syntax

SPDEWHEVAL=COST | EVAL1 | EVAL3EVAL4

COST
specifies that the SPD Engine decides which evaluation strategy to use in order to optimize the WHERE expression. This process also calculates the number of threads to be used, which minimizes the overhead of spawning underutilized threads. This is the default.

EVAL1
is a multi-threaded index evaluation strategy that can quickly determine the rows that satisfy the WHERE expression, using multiple threads. The number of threads that are spawned to retrieve the observations is equal to the THREADNUM= value.

EVAL3EVAL4
is a single-threaded index evaluation strategy that is used for a simple or compound WHERE expression in which all of the key variables have a simple index and no condition tests for non-equality. Multi-threading might be used to retrieve the observations.

Details

COST, the default setting for SPDEWHEVAL=, analyzes the WHERE expression and any available indexes. Based on the analysis, the SPD Engine chooses an evaluation strategy in order to optimize the WHERE expression. The evaluation strategy can be EVAL1, EVAL3, EVAL4, or a strategy that sequentially reads the data if no indexes are available or if the analysis shows that using the index(es) will not improve processing time.

COST also optimizes the number of threads to use for processing the WHERE expression. COST determines and spawns the number of threads that can be efficiently used. Based on the value of THREADNUM=, COST can save significant processing time by not spawning threads that are underutilized.

COST is the recommended value for SPDEWHEVAL= unless the WHERE expression exactly meets one of the other evaluation strategy criterion. It is strongly recommended that benchmark tests be used in order to determine if a value other than COST is more efficient.

For example, EVAL1 might prove more efficient if the WHERE expression is complex and there are multiple indexes for the variables. EVAL1 spawns multiple threads in order to determine which segments meet the conditions of the WHERE expression. Multiple threads can also be used to retrieve the observations.

Note: In a few situations, COST might not perform the best. To determine if changing the value to EVAL1 or EVAL3EVAL4 can produce better performance, run a performance benchmark, change the value, and re-run the performance benchmark. Comparing results will show you how the specific data population you are querying responds to rules-based WHERE planning. △

SPD Engine System Options List

"COMPRESS= System Option" on page 64
: specifies to compress the SPD Engine data set on disk as they are being created.

MAXSEGRATIO=
: When evaluating a WHERE expression that contains indexed variables, controls what percentage of index segments to identify as candidate segments before processing the WHERE expression.

"MINPARTSIZE= System Option" on page 66
: specifies a minimum partition size to use for creating SPD Engine data sets.

"SPDEINDEXSORTSIZE= System Option" on page 66
: specifies the size of memory space that the sorting utility can use when sorting values for creating an index.

"SPDEMAXTHREADS= System Option" on page 67
: specifies the upper limit on the number of threads that the SPD Engine can spawn for I/O processing.

"SPDESORTSIZE= System Option" on page 68
: specifies the size of memory space needed for sorting operations used by the SPD Engine.

"SPDEUTILLOC= System Option" on page 68
: specifies one or more file system locations in which the SPD Engine can temporarily store utility files.

SPDEWHEVAL=
: specifies the process used to determine which observations meet the condition(s) of a WHERE expression.

SAS System Options That Behave Differently with SPD Engine

MSGLEVEL=
: The value I enables WHINIT planner output.

COMPRESS=
: Does not accept BINARY as an argument; cannot perform user-defined compression.

DLDMGACTION=
: Does not affect the SPD Engine. If an SPD Engine data set is damaged, it must be restored from a system backup file.

P A R T 3

Appendix

Appendix ***1*** . . . . **Quick Guide to the SPD Engine Disk-I/O Set-Up** *75*

Appendix ***2*** . . . . **Recommended Reading** *89*

APPENDIX

1

Quick Guide to the SPD Engine Disk-I/O Set-Up

SPD Engine Disk-I/O Set-Up **75**
Disk Striping and RAIDs **76**
Metadata Area Configuration **77**
    Assigning a Metadata Area **77**
    Metadata Space Requirements **77**
Data Area Configuration **77**
    Assigning a Data Area **78**
    Data Partition Size **78**
    Data Area Set-Up **78**
    Data Space Requirements **81**
Index Area Configuration **81**
    Assigning an Index Area **81**
    Index Space Requirements **82**
        Estimate for HBX file size **82**
        Example **82**
        Estimate for IDX file size **82**
        Example **83**
        Space Requirement for Index Creation **83**
Work Area Configuration **84**
Configuration Validation Program **85**
    Preparation **85**
    Running the Program **86**
    Interpreting the Results **87**

SPD Engine Disk-I/O Set-Up

The SPD Engine usually uses four different areas to store the various components that make up an SPD Engine data set:

- metadata area
- data area
- index area
- work area.

These areas have different disk set-up requirements that utilize one or more RAID (redundant array of independent disks) levels.

Disk Striping and RAIDs

The SPD Engine disk configuration is best performed using RAIDs.

A defining feature of almost all RAID levels is disk striping (RAID-1 is the exception). Striping is the process of organizing the linear address space of a volume into pieces that are spread across a collection of disk drive partitions. For example, you might configure a volume across two 1-gigabyte partitions on separate disk drives (for example, A and B) with a stripe size of 64 kilobytes. Stripe 0 lives on drive A, stripe 1 lives on drive B, stripe 2 lives on drive A, and so on.

By distributing the stripes of a volume across multiple disks, it is possible

- to achieve parallelism at the disk I/O level
- to use multiple threads to drive a block of I/O.

This also reduces contention and data transfer latency for a large block I/O requests because the physical transfer can be split across multiple disk controllers and drives.

Note: *Important:* Regardless of RAID level, disk volumes should be hardware striped, not software striped. This is a significant way to improve performance. Without hardware striping, I/O will bottleneck and constrain performance. A stripe size of 64 kilobytes is a good value. △

The following is a brief summary of RAID levels relevant to the SPD Engine.

RAID 0 (also referred to as striped set)
: High performance with low availability. I/O requests are chopped into multiple smaller pieces, each of which is stored on its own disk. Physically losing a disk means that all the data on the array is lost. No redundancy exists to recover volume stripes on a failed disk. A striped set requires a minimum of two disks. The disks should be identical. The net capacity of a RAID 0 array equals the sum of the single disk capacities.

RAID 1 (also referred to as mirror)
: Disk mirroring for high availability. Every block is duplicated on another mirror disk, which is also referred to as shadowing. In the event that one disk is lost, the mirror disk is likely to be intact, preserving the data. RAID 1 can also improve read performance because a device driver has two potential sources for the same data. The system can choose the drive that has the least load/latency at a given point in time. Two identical disks are required. The net capacity of a RAID 1 array equals that of one of its disks.

RAID 5 (also referred to as striped set with rotating ECC)
: Good performance and availability at the expense of resources. An error-correcting code (ECC) is generated for each stripe written to disk. The ECC distributes the data in each logical stripe across physical stripes in such a way that if a given disk in the volume is lost, data in the logical stripe can still be recovered from the remaining physical stripes. The downside of a RAID 5 is resource utilization; RAID 5 requires extra CPU cycles and extra disk space to transform and manage data using the ECC model. If one disk is lost, rebuilding the array takes significant time because all the remaining disks have to be read in order to rebuild the missing ECC and data. The net capacity of a RAID 5 array consisting of N disks is equal to the sum of the capacities of N–1 of these disks. This is because the capacity of one disk is needed to store the ECC information. Usually RAID 5 arrays consist of three or five disks. The minimum is three disks.

RAID 10 (also referred to as striped mirrors, RAID 1+0)
: Many RAID systems offer a combination of RAID 1 (pure disk mirroring) and RAID 0 (striping) to provide both redundancy and I/O parallelism this configuration (also referred to as RAID 1+0). Advantages are the same as for RAID 1 and RAID 0. A RAID 10 array can survive the loss of multiple disks. The only disadvantage is the requirement for twice as many hard disks as the pure RAID 0 solution. Generally, this configuration tends to be a top performer if you have the disk resources to pursue it. If one disk is lost in a RAID 10 array, only the mirror of this disk has to be read in order to recover from that situation. Raid 10 is not to be confused with RAID 0+1 (also referred to as mirrored stripes), which has slightly different characteristics. The net capacity of RAID 10 is the sum of the capacity of half of its disks.

Non-RAID (also referred to as just a bunch of disks or JBOD)
: This is actually not a RAID level and is only mentioned for completeness. This refers to a couple of hard disks, which can be stand-alone or concatenated to achieve higher capacity than a single disks. JBODs do not feature redundancy and are slower than most RAID levels.

Metadata Area Configuration

The metadata area keeps information about the data and its indexes. It is vital not to lose any metadata. Therefore this area needs disk set-up, which features primarily redundancy, such as RAID 1, also known as mirroring.

Assigning a Metadata Area

The physical metadata location is determined by the primary path definition in the LIBNAME statement. In the example code below, the primary path is /SPDEMETA1:

```
libname mydomain SPDE '/spdemeta1'
   metapath=('/spdemeta2')
   datapath=('/spdedata1' '/spdedata2' '/spdedata3' 'spdedata4')
   indexpath=('/spdeindex1' '/spdeindex2');
```

The “METAPATH= LIBNAME Statement Option” on page 28 specifies a list of overflow paths to store metadata file partitions (MDF components) for a data set.

Metadata Space Requirements

The approximate metadata space consumption is

space in bytes = 12KB + (#variables * 12) + (5KB * #indexes)

This estimate increases if you delete observations from the data set or use compression on the data. In general, the size of this component file is small (below 1 megabyte).

Data Area Configuration

The data area is where the data component files are stored. The data area requires specific set-up in order to provide high I/O-throughput as well as scalability and availability.

Assigning a Data Area

The physical data location is determined by the “DATAPATH= LIBNAME Statement Option” on page 26:

```
libname mydomain SPDE '/spdemeta1'
   metapath=('/spdemeta2')
   datapath=('/spdedata1' '/spdedata2' '/spdedata3' '/spdedata4')
   indexpath=('/spdeindex1' '/spdeindex2');
```

In order to achieve parallel access to the data, the data set is partitioned into multiple physical operating system files (referred to as .DPF components), which should be stored on multiple disks. The DATAPATH= option specifies a list of file systems (under UNIX systems) or disk drives (under Windows) where the data partitions are stored. The first data partition will be stored on the first file system in the list, the second partition on the second file system and so on. After the final file system has been reached, the next partition will again be stored on the first file system. Hence the data file systems will roughly be filled up equally.

The set-up of the data file systems is crucial to the performance that will be achieved when retrieving the data.

The DATAPATH= option is optional. If it’s omitted, all .DPF components will be stored in the primary path. This will work at the expense of performance and scalability.

Data Partition Size

The data partition size should be chosen in a way so that three or four partitions of each data set reside in each data path. The number of partitions per data path should not exceed ten. The main disadvantage of having too many partitions is that too many physical files will be opened when the data set is opened. This has a negative impact on operating system resources and on other applications, which are executed at the same time. Having too many partitions does not help with better performance either. As a guideline for determining a reasonable partition size, use the following formula:

partition size=(#observations*obs length) / (#data file systems*max partitions per file system)

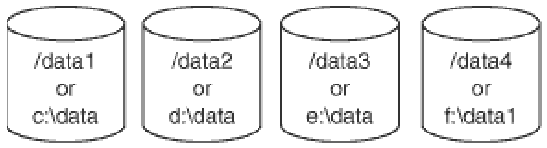
The partition size should then be rounded up to values like 16, 64, 128, 256 megabytes and so on.

Data Area Set-Up

On an *N*-way computer, aim to have *N* identical data paths to spread the data partitions across. A *data path* is a file system on UNIX or a logical disk drive on Windows. It is good practice to have one I/O-controller per data path. Depending on the I/O-bandwidth of the disks, multiple controllers could be required. Keep the set-up as simple as possible; that is, there should be a one-to-one mapping between hard disks (spindles) or groups (RAID) of them on one side and file systems or logical disk drives on the other side.

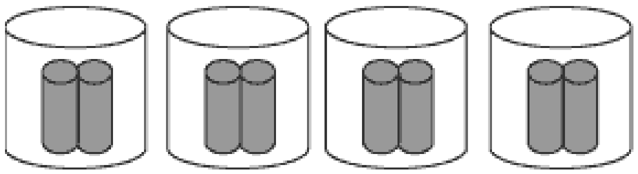
For instance, on a four-way machine, the simplest possible set-up is to use four hard disks and make one file system or logical disk drive per hard disk as shown in the following figure.

Figure A1.1 Four Single Disk Drives



In order to achieve best performance, reserve the disk drives for the SPD Engine storage usage exclusively. In order to get better performance, each of these disk drives could be replaced by a stripe-set of many disks (RAID 0); see the following figure. Usually, better performance can be achieved with wider striping.

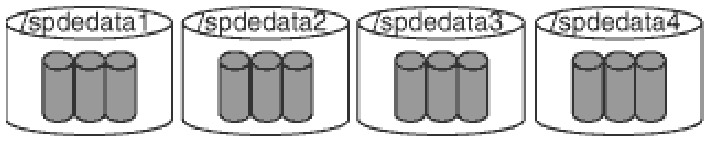
Figure A1.2 Four RAID 0 Arrays, Each Striped across Two Disks



However, if any one of the disks in the above figure fails, then all the data will be lost, because there is no redundancy in striping. In order to achieve redundancy, each of these RAID 0 arrays needs to be replaced with either a mirrored disk array (RAID 1) or a mirrored stripe-set (RAID 10) or a RAID 5 array.

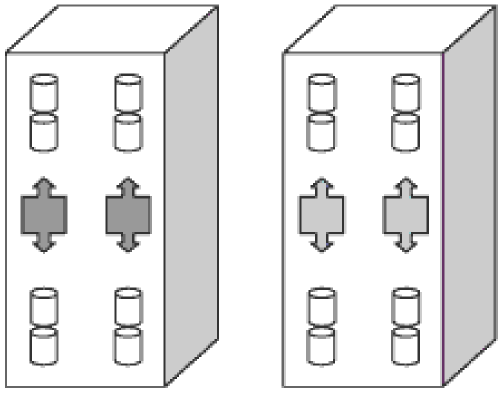
RAID 10 features the best performance while maintaining redundancy at the same time. It requires at least four disks in each array. RAID 5, also referred to as striping with rotating error correction code (ECC), has the best ratio of redundancy and performance versus cost on the other side. A minimum configuration requires only three disks per array as shown in the following figure. There is a small penalty when writing to RAID 5, as the ECC information needs to be refreshed every time the data is changed.

Figure A1.3 Four RAID 5 Arrays, Each Striped across Three Disks



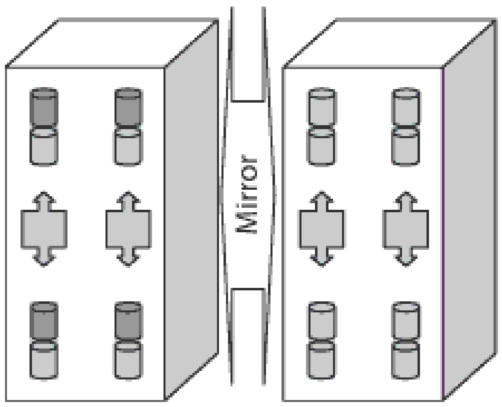
Normally, the hard disks in disk arrays are organized in groups, each of which is connected to its own hard disk controller. The following figure shows two disk towers with eight hard disks and two disk controllers each. Four disks are grouped with each controller.

Figure A1.4 Two Hard Disk Towers



Assuming that each of the disks runs at a throughput of 35 megabytes and each controller features two channels that operate at 80 megabytes each, two disks can effectively saturate one controller channel. The disks need to be carefully striped across the existing controller channels when creating stripe-sets and disk mirrors.

Figure A1.5 Four RAID 10 Data Paths



In order to create four RAID 10 data paths for the SPD Engine to partition the data across, the left disk array is considered to be the actual data array, while the right one is the mirror. See the above figure.

For the first data path, the two uppermost disks in the left array are combined to a stripe-set across two disks. Both disks are connected to different controllers, to avoid any sort of contention. The combined throughput of this stripe-set should be around 60 megabytes in practice. In the right array, the two uppermost disks are defined to be the mirrors of the respective disks in the left array. This gives almost the combined throughput of four disks connected to four controllers when reading from multiple processes, as the I/O subsystem has the choice of serving the request by reading from either the original data disks or their mirrors. Doing the same with the next three rows of disks, the result is four data paths for parallel I/O. Each data path is striped over two disks, which are mirrored in the other array.

The overall throughput when launching four threads should be approximately 4*60MB or 240MB. As the striping and mirroring is symmetric across all components, this also gives reasonable load-balancing in parallel. The theoretical limitation is 640 megabytes, as the four controllers can run at 160 megabytes across two channels.

Different vendor's hardware devices might show different results in this area. However, in principle, these numbers should be a good guideline.

Data Space Requirements

The estimated data space consumption for an uncompressed data set is

space in bytes = #observations * obs length

The space consumption for compressed data sets will obviously vary with the compression factor for the data set as a whole.

Index Area Configuration

The index component files are stored in the index area. With regard to disk set-up, this should be a stripe-set of multiple disks (RAID 0) for good I/O-performance. Reliability, availability, and serviceability (RAS) concerns could eventually dictate to choose any sort of redundancy, too. In this case, a RAID 5 array or a combination of mirroring and striping (RAID 10) would be appropriate.

Assigning an Index Area

The physical index location is determined by the "INDEXPATH= LIBNAME Statement Option" on page 28:

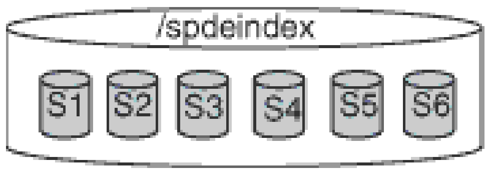
```
libname mydomain SPDE '/spdemeta1'
   metapath=('/spdemeta2')
   datapath=('/spdedata1' '/spdedata2' '/spdedata3' 'spdedata4')
   indexpath=('/spdeindex1' '/spdeindex2');
```

The INDEXPATH= option specifies a list of file systems where the index partitions are stored. The index component file will be stored on the first file system until it fills up. Then the next file system in the list will be used.

The INDEXPATH= option is optional. If it's omitted from the LIBNAME= statement, all index files (IDX and HBX component files) will be stored in the primary path location. Usually this is not a good idea when good performance is expected.

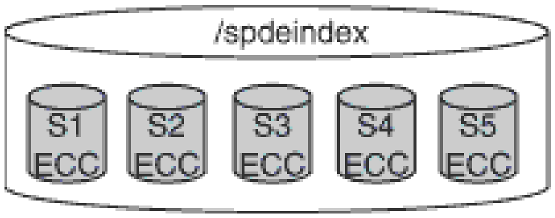
It is strongly recommended to configure INDEXPATH= using a volume manager file system that is striped across multiple disks, as shown in the following figure, to allow adequate index performance, both when evaluating WHERE clauses and creating indices in parallel.

Figure A1.6 Index Area Striped across Six Disks, S1 through S6



In a real-life production environment, the INDEXPATH= option is likely to point to a RAID 5 array as shown in the following figure. This is most cost-effective while maintaining a good read performance and availability at the same time. As indices are not constantly built or refreshed, the lower write performance of RAID 5 should not be an obstacle here.

Figure
A1.7 Index Area on a RAID 5 Array Striped across Five Disks with Rotating ECC



Index Space Requirements

An SPD Engine index uses two component files. The IDX file is the segmented view of the index, and the HBX file is the global portion of the index. You can estimate space consumption roughly for the HBX component of an index as follows.

Estimate for HBX file size

To estimate the size, in bytes, of the HBX file for a given index, use this formula:

HBX size = (number of unique values) * (22.5 + *length*) * *factor*

where *length* is the length (in bytes) of all variables combined in the index, and factor takes the following values:

if *length* < 100, then factor = 1.2 – (0.002 * *length*)

if *length* >= 100, then factor = 1.04 + (0.0002 * *length*)

Note: The estimate for the file size provides a maximum for a newly built index. The estimate might be on the low side for lengths larger than 500 bytes. △

Note: The formula does not apply to files smaller than one megabyte. △

Example

For an index on a character variable of length 10 that has 500,000 unique values, here is the calculation:

HBX	= 500000 * (22.5 + 10) * (1.2 – 0.002*10)
	= 19175000 bytes

The actual size is 19,152,896 bytes.

Estimate for IDX file size

The IDX component file contains the per-value segment lists and bitmaps portion of the index. Estimating disk space consumption for this file is much more difficult than for the HBX component file. This is because the IDX file size depends on the distribution of the key values across the data. If a key variable's value is contained in many segments, then a larger segment list is required, and therefore a greater number of per-segment bitmaps are required.

The size also depends on the number of updates or appends performed on the index. The IDX files of an indexed data set initially created with *N* observations consumes considerably less space than the IDX files of an identical data set on which several append or updates were performed afterward.

With the above in mind, to get a worst-case estimate for space consumption of the IDX component of an index, use the following formula:

IDX size = 8192 + (*D* * (24 + (*P* * (16 + (*S* / 8)))))

where

D is the number of discrete values that occur in more than one observation

P is the average number of segments that contain each value

S is the segment size.

This estimate does not take into consideration the compression factor for the bitmaps, which could be substantial. The fewer occurrences of a value in a given segment, the more the bitmap for that segment can be compressed. The uncompressed bitmap size is the (segment size/8) component of the algorithm.

Example

To estimate the disk usage for a non-unique index on a variable with a length of 8, where the variable contains 1024 discrete values, and each value is in an average of 4 segments with a segment size of 8192 observations, the calculations would be (rounding up the HBX result to a whole number)

HBX size = 1024 * (22.5 + 8) * (1.2 – (0.002 * 8)) = 36979 bytes

IDX size = 8192 + (1024 * (24 + (4 * (16 + (8192/8))))) = 4285440 bytes

To estimate the disk usage of a unique index on a variable with a length of 8 that contains 100,000 values, the calculations would be

HBX size = 100000 * (22.5 + 8) * (1.2 – (0.002 * 8)) = 3611200 bytes

IDX size = 8192 + (0 * (24 + (4 * (16 + (8192/8))))) = 8192 bytes

Note: The size of the IDX file for a unique index will always be 8192 bytes because the unique index contains no values that are in more than one observation. △

Space Requirement for Index Creation

There is a hidden requirement for work area space when creating indexes or when appending indexes in the SPD Engine. This need arises from the fact that the SPD Engine sorts the observations by the key value before adding the key values to the index. This greatly improves the index create/append performance but comes with a price—the temporary disk space that holds the sorted keys while the index create/append is in progress.

You can estimate the work area space for index creation as follows for a given indexed variable:

space in bytes = *#obs * (8 + sortlength)*

where

#obs is the number of observations in the data set if creating; or number of observations in the append if appending.

sortlength is the sum of the length of the variables that make up the index. For example, to create the index for a numeric variable on a data set with 1,000,000 rows, the calculation would be 1,000,000 * (8 + 8) = 16,000,000 bytes. To create a compound index of two variables (lengths 5 and 7) on the same data set, the calculation would be 1,000,000 * (5 + 7 + 8) = 20,000,000 bytes.

If you create the indexes in parallel by using the ASYNCINDEX=YES data set option, you must sum the space requirements for each index that you create in the same create phase.

The same applies to PROC APPEND runs when you append to a data set with indices. In this case, all of the indices are refreshed in parallel, so you must sum the workspace requirement across all indexes.

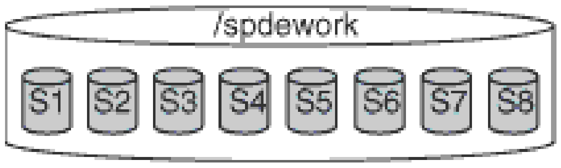
Work Area Configuration

The work area is where temporary files are created. For example, temporary utility files can be generated during the SPD Engine operations that need extra space, like index creation as noted above, or sorting operation of very large files.

Normally a stripe-set of multiple disks (RAID 0) should be sufficient to gain good I/O-throughput. However, again, RAS could also dictate to choose redundancy (RAID 5 or RAID 10) because a loss of the work area could stop the SPD Engine from functioning entirely.

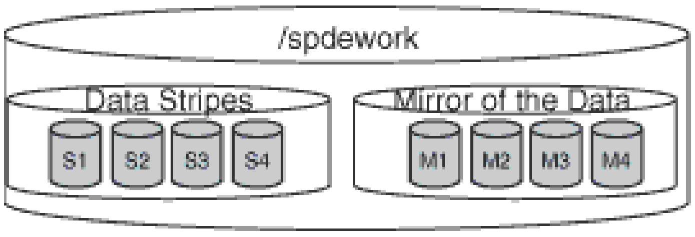
Using “SPDEUTILLOC= System Option” on page 68 to specify multiple storage locations can reduce out-of-space conditions and improve performance. We strongly recommend that you configure SPDEUTILLOC= to use a volume manager file system that is striped across multiple disks in order to provide optimum performance and to allow adequate temporary workspace performance, as shown in the following figure.

Figure A1.8 Work Area Striped across Eight Disks



In a production environment, you will probably point SPDEUTILLOC= to a RAID 5 array or, even better, a RAID 10 array as shown on the following figure. Writing and reading in the work area will probably happen equally often. While RAID 5 is most cost-effective, a RAID 10 would give highest performance and availability, plus there is no write penalty because no ECC information has to be updated. The mirroring will be done during idle times without virtually affecting any requests.

Figure A1.9 Work Area Striped across Four Disks and Mirrored



Configuration Validation Program

The SAS program SPDECONF.SAS, described here, measures I/O scalability and can help you determine whether the system is properly configured.

The program creates a data set with two numeric variables. It then proceeds to repeatedly read the entire data set, each time doubling the number of threads used (by increasing the setting for “THREADNUM= Data Set Option” on page 55) until the maximum number is reached. The resulting SAS log file shows timing statistics for each cycle. By examining this information you can determine whether your system is configured correctly.

Preparation

1 Before you run the program, you must customize it. Gather the following information:

- □ the number of CPUs in your machine.
- □ the number of disks on which you will store data. This number equals the number of paths specified in the “DATAPATH= LIBNAME Statement Option” on page 26.
- □ the amount of RAM in your machine.

2 Use the first two items above to determine the value you must use for the “SPDEMAXTHREADS= System Option” on page 67. That option must be specified either in the SAS configuration file or on the SAS invocation line. (For details on the syntax, refer to Chapter 5, “SPD Engine System Options,” on page 63.) Set SPDEMAXTHREADS= to the larger of the following:

- □ 8 × number of CPUs
- □ 2 × number of paths in the DATAPATH= option.

CAUTION:
Use this value for the validation test only. It is probably too high for most kinds of processing. Following the test, be sure to reset the value, and restart SAS. △

For example, if the machine has six CPUs and the LIBNAME statement is

```
LIBNAME SCALE SPDE '/primary-path' DATAPATH=('/data01' '/data02'
  '/data03' '/data04' '/data05' '/data06' '/data07');
```

then you set SPDEMAXTHREADS=48 (the larger of 8 × 6 and 2 × 7).

3 Now you must edit the SPDECONF.SAS program to set the NROWS macro variable. Set NROWS such that the resulting data set is more than twice the available RAM. For example, if the available RAM is 1 gigabyte, set NROWS=150000000, which is 2G/16 rounded up. The number 16 is used because the data set has two numeric variables, and therefore each observation is 16 bytes long. This calculation for NROWS is used to create a data set that is large enough to overcome the beneficial effects of caching by the operating system.

Here is SPDECONF.SAS. Edit the items to fit your operating environment.

```
options source2 fullstimer;

%let nrows = nrows;
```

```
/* LIBNAME statement */
LIBNAME SCALE SPDE '/primary-path' DATAPATH=('/path01' '/path02'
  '/path03' '/path04' '/path05' '/path06' '/path07');

data scale.test;
   do i = 1 to &nrows;
      x = mod(i,33);
      output;
   end;
run;

%macro ioscale(maxth);
   %put "SPDEMAXTHREADS = &maxth";
   %let tcnt = 1;
   %do %while(&tcnt le &maxth);
      %put "THREADNUM = &tcnt";
      data _null_;
         set scale.test(threadnum=&tcnt);
         where x = 33;
      run;
   %let tcnt = %eval(&tcnt * 2);
   %end;
%mend;

%ioscale(%sysfunc(getoption(spdemaxthreads)));
%ioscale(%sysfunc(getoption(spdemaxthreads)));
%ioscale(%sysfunc(getoption(spdemaxthreads)));

proc datasets lib=scale kill;
run;
quit;
```

Running the Program

Follow these steps to run the SPDECONF.SAS program:

1 You must take the following precautions before you run the %IOSCALE macro, because it measures performance:
   - □ Ensure that no other programs are running on the machine.
   - □ Ensure that no other users can access the machine during the test.
   - □ Ensure that SPDEMAXTHREADS= is set.

2 Create the SCALE.TEST data set.

3 Run the %IOSCALE macro three times, noting the time for each run.

4 To complete your results, use the following code to create the same data set with the Base SAS engine:

```
data testbase.test;
   do i = 1 to &nrows;
      x = mod(i,33);
         output;
      end;
   run;
```

Run the following DATA step against the TESTBASE.TEST data set:

```
data _null_;
   set scale.test;
   where x=33;
run;
```

As in step 3, write down the real time that it took to run the DATA _NULL_.

Interpreting the Results

First, average the results from the three %IOSCALE macros. (You will find the data you need in the log file for the program, SPDECONF.LOG). If the machine is correctly configured, you should see these results:

- The real time for each successive DATA step should ideally follow the curve 1/THREADNUM. That is, it should take half as much time to process with each successive doubling of the number of threads.

  At the very least, for the first several iterations, the time for each successive DATA step should decline and then after some point the curve should begin to flatten or bottom out.

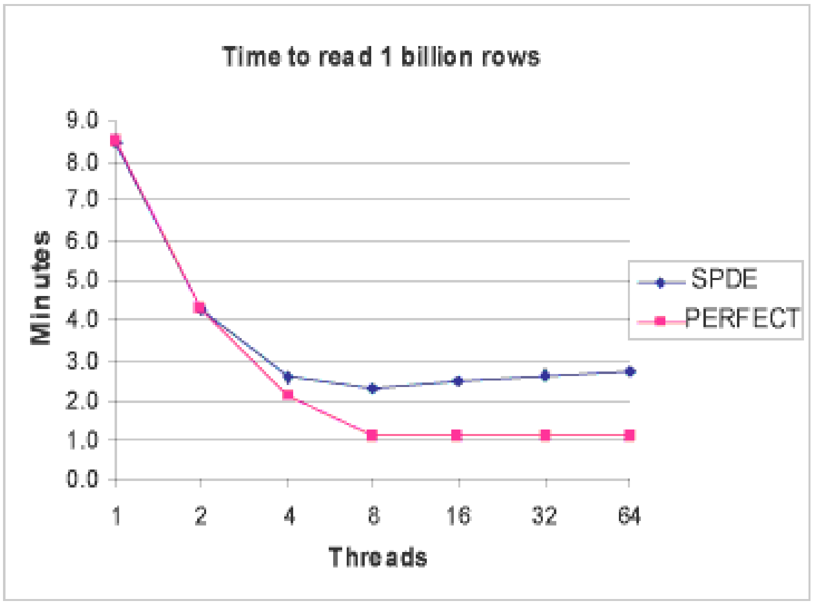
- The time with one thread should be less than or equal to the time to process the DATA step with the Base SAS engine.

If the results do not fit the criteria above, something is wrong with the configuration and must be corrected.

Once you get a curve with the characteristics listed above, set the value of the invocation option SPDEMAXTHREADS= in the SAS configuration file to the value of THREADNUM= where the curve flattens/bottoms (see the graph below). This will generally be the most efficient configuration for WHERE-clause processing but might not be best for other kinds of processing. In any case, if you need to specify fewer threads for any individual SAS job, you can use THREADNUM= to override SPDEMAXTHREADS= temporarily. See "THREADNUM= Data Set Option" on page 55 and "SPDEMAXTHREADS= System Option" on page 67 for details about these options.

The following graph summarizes the results from an actual use of the SPDECONF.SAS program. The data set has 2 numeric variables and 1 billion observations. The WHERE expression asks for a record that is not in the data set. Without any indexes, the SPD Engine is forced to do a full scan of the data file. Note that the number of threads is a surrogate for the number of CPUs. The scalability levels off with eight threads, which is the number of CPUs on the test machine. Specifying a number of threads larger than 2 or 3 times the number of available CPUs does not improve performance.

Figure A1.10 Time to Read 1 Billion Rows



Note:

□ This type of scalability might not be seen when all the data file systems reside on the same RAID 5 array, consisting of only three or five disks, in which case the curve will be more or less flat all the way through. You might want to try altering your hardware set-up. A much better set-up would be to place each data file system on its own RAID 5 array. Then rerun this test to see if there are improvements.

□ Not only the scalability but also the overall throughput in megabytes per second is a figure that should be calculated in order to know whether the set-up is appropriate. To calculate this number, just take the size of the data set, in megabytes, and divide it by the real time the step took in seconds. This number should come as close as 70 to 80 percent to the theoretical maximum of your I/O-bandwidth, if the set-up is correct.

□ Make sure you assign data paths that use separate I/O controllers and that you use hardware striping.

□ On some systems, DATAPATHs are not needed if the LIBNAME domain's primary directory is on a file system with hardware striping across multiple controllers.

□ Check your SPDEMAXTHREADS system option. If the THREADNUM value exceeds the SPDEMAXTHREADS setting, then SPDEMAXTHREADS will take precedence. You need to temporarily change SPDEMAXTHREADS to the artificially high value for this test and then restore it after the test is complete. Remember that the Base SAS software needs to be restarted in order to pick up the change to SPDEMAXTHREADS.

△

APPENDIX
2

Recommended Reading

Recommended Reading **89**

Recommended Reading

Here is the recommended reading list for this title:

- *Base SAS Procedures Guide*
- *SAS Language Reference: Concepts*
- *SAS Language Reference: Dictionary*

For a complete list of SAS publications, see the current *SAS Publishing Catalog*. To order the most current publications or to receive a free copy of the catalog, contact a SAS representative at

SAS Publishing Sales
SAS Campus Drive
Cary, NC 27513
Telephone: (800) 727-3228*
Fax: (919) 677-8166
E-mail: **sasbook@sas.com**
Web address: **support.sas.com/pubs**
* For other SAS Institute business, call (919) 677-8000.

Customers outside the United States should contact their local SAS office.

Glossary

block (of data)
a group of observations in a data set. If an application is thread-enabled, it can read, write, and process the observations faster when they are delivered as a block than when they are delivered as individual observations.

compound WHERE expression
a WHERE expression that contains more than one operator, as in `WHERE X=1 and Y>3`. See also WHERE expression.

controller
a computer component that manages the interaction between the computer and a peripheral device such as a disk or a RAID. For example, a controller manages data I/O between a CPU and a disk drive. A computer can contain many controllers. A single CPU can command more than one controller, and a single controller can command multiple disks.

CPU-bound application
an application whose performance is constrained by the speed at which computations can be performed on the data. Multiple CPUs and threading technology can alleviate this problem.

data partition
a physical file that contains data and which is part of a collection of physical files that comprise the data component of an SPD Engine data set. See also partition, partitioned data set.

I/O-bound application
an application whose performance is constrained by the speed at which data can be delivered for processing. Multiple CPUs, partitioned I/O, threading technology, RAID (redundant array of independent disks) technology, or a combination of these can alleviate this problem.

light-weight process thread
a single-threaded subprocess that is created and controlled independently, usually with operating system calls. Multiple light-weight process threads can be active at one time on symmetric multiprocessing (SMP) hardware or in thread-enabled operating systems.

multi-threading
See threading.

ODS-5 (On-Disk Structure Level 5)
a file system structure that is implemented by OpenVMS Alpha Version 7.2. ODS-5 allows longer filenames, supports more legal characters within filenames, preserves case within filenames, supports deeper directory structures, and provides better compatibility with other file systems such as those used with UNIX or Windows.

parallel I/O
a method of input and output that takes advantage of multiple CPUs and multiple controllers, with multiple disks per controller to read or write data in independent threads.

parallel processing
a method of processing that uses multiple CPUs to process independent threads of an application²s computations. See also threading.

partition
part or all of a logical file that spans devices or directories. In the SPD Engine, a partition is one physical file. Data files, index files, and metadata files can all be partitioned, resulting in data partitions, index partitions, and metadata partitions, respectively. Partitioning a file can improve performance for very large data sets. See also data partition, partitioned data set.

partitioned data set
in the SPD Engine, a data set whose data is stored in multiple physical files (partitions) so that it can span storage devices. One or more partitions can be read in parallel by using threads. This improves the speed of I/O and processing for very large data sets. See also parallel processing, partition, thread.

primary path
the location in which SPD Engine metadata files are stored. The other SPD Engine component files (data files and index files) are stored in separate storage paths in order to take advantage of the performance boost of multiple CPUs.

process
a functional unit of a program or task. In a thread-enabled operating system, a process can consist of a single thread, or it can contain many threads that collectively perform a complex function. See also thread, thread-enabled operating system.

RAID (redundant array of independent disks)
a type of storage system that comprises many disks and which implements interleaved storage techniques that were developed at the University of California at Berkeley. RAIDs can have several levels. For example, a level-0 RAID combines two or more hard drives into one logical disk drive. Various RAID levels provide various levels of redundancy and storage capability. A RAID provides large amounts of data storage inexpensively. Also, because the same data is stored in different places, I/O operations can overlap, which can result in improved performance. See also redundancy.

redundancy
a characteristic of computing systems in which multiple interchangeable components are provided in order to minimize the effects of failures, errors, or both. For example, if data is stored redundantly (in a RAID, for example), then if one disk is lost, the data is still available on another disk. See also RAID (redundant array of independent disks).

SASROOT
a term that represents the name of the directory or folder in which SAS is installed at your site or on your computer.

scalability

the ability of a software application to function well with little degradation in performance despite changes in the volume of computations or operations that it performs and despite changes in the computing environment. Scalable software is able to take full advantage of increases in computing capability such as those that are provided by the use of SMP hardware and threaded processing. See also scalable software, server scalability, SMP (symmetric multiprocessing).

Scalable Performance Data Engine
See SPD (Scalable Performance Data) Engine.

scalable software
software that responds to increased computing capability on SMP hardware in the expected way. For example, if the number of CPUs is increased, the time to solution for a CPU-bound problem decreases by a proportionate amount. And if the throughput of the I/O system is increased, the time to solution for an I/O-bound problem decreases by a proportionate amount. See also server scalability, SMP (symmetric multiprocessing), time to solution.

server scalability
the ability of a server to take advantage of SMP hardware and threaded processing in order to process multiple client requests simultaneously. That is, the increase in computing capacity that SMP hardware provides increases proportionately the number of transactions that can be processed per unit of time. See also SMP (symmetric multiprocessing), threaded processing.

SMP (symmetric multiprocessing)
a hardware and software architecture that can improve the speed of I/O and processing. An SMP machine has multiple CPUs and a thread-enabled operating system. An SMP machine is usually configured with multiple controllers and with multiple disk drives per controller.

spawn
to start a process or a process thread such as a light-weight process thread (LWPT). See also thread.

SPD (Scalable Performance Data) Engine
a SAS engine that is able to deliver data to applications rapidly because it organizes the data into a streamlined file format. The SPD Engine also reads and writes partitioned data sets, which enable it to use multiple CPUs to perform parallel I/O functions. See also parallel I/O.

SPD Engine data file
the data component of an SPD Engine data set. In contrast to SAS data files, SPD Engine data files contain only data; they do not contain metadata. The SPD Engine does not support data views. See also SPD Engine data set.

SPD Engine data set
a data set created by the SPD Engine that has up to four component files: one for data, one for metadata, and two for any indexes. The minimum number of component files is two: data and metadata. Data is separated from the metadata for SPD Engine file organization.

symmetric multiprocessing
See SMP (symmetric multiprocessing).

thread
a single path of execution of a process in a single CPU, or a basic unit of program execution in a thread-enabled operating system. In an SMP environment, which uses multiple CPUs, multiple threads can be spawned and processed simultaneously. Regardless of whether there is one CPU or many, each thread is an independent flow

of control that is scheduled by the operating system. See also SMP (symmetric multiprocessing), thread-enabled operating system, threading.

thread-enabled operating system
an operating system that can coordinate symmetric access by multiple CPUs to a shared main memory space. This coordinated access enables threads from the same process to share data very efficiently.

thread-enabled procedure
a SAS procedure that supports threaded I/O or threaded processing.

threaded I/O
I/O that is performed by multiple threads in order to increase its speed. In order for threaded I/O to improve performance significantly, the application that is performing the I/O must be capable of processing the data rapidly as well. See also I/O-bound application.

threaded processing
processing that is performed in multiple threads on multiple CPUs in order to improve the speed of CPU-bound applications. See also CPU-bound application.

threading
a high-performance method of data I/O or data processing in which the I/O or processing is divided into multiple threads that are executed in parallel. In the boss-worker model of threading, the same code for the I/O or calculation process is executed simultaneously in separate threads on multiple CPUs. In the pipeline model, a process is divided into steps, which are then executed simultaneously in separate threads on multiple CPUs. See also parallel I/O, parallel processing, SMP (symmetric multiprocessing).

time to solution
the elapsed time that is required for completing a task. Time-to- solution measurements are used to compare the performance of software applications in different computing environments. In other words, they can be used to measure scalability. See also scalability.

WHERE expression
a type of SAS expression that specifies a condition for selecting observations for processing by a DATA step or a PROC step. WHERE expressions can contain special operators that are not available in other SAS expressions. WHERE expressions can appear in a WHERE statement, a WHERE= data set option, a WHERE clause, or a WHERE command. See also compound WHERE expression.

Index

A

APPEND procedure
  converting Base SAS engine data sets 15
ASYNCINDEX= data set option 36

B

Base SAS engine
  compared with SPD Engine 5
  converting data sets for SPD Engine 7, 14
BYNOEQUALS= data set option 37
BYSORT= data set option 39
BYSORT= option
  LIBNAME statement, SPD Engine 24

C

component files 4
  anticipating space for 12
  configuring space for 12
  naming conventions 16
  storing 8
COMPRESS= data set option 41
COMPRESS= system option 64, 71
compressing SPD Engine data sets 41, 64
compression blocks
  adding bytes to 47
  number of observations in 46
configuration validation program 85
converting data sets
  Base SAS engine to SPD Engine 7, 14
COPY procedure
  converting Base SAS engine data sets 15

D

data area
  assigning 78
  configuring 77
  data partition size 78
  set-up 78
  space requirements 81
data component files 5
data organization 4
data partitions
  size of 29, 48, 66, 78
  storing 14, 26
data paths 78
data set options
  list of 60
  not supported by SPD Engine 61
  SPD Engine 35
  SPD Engine vs. Base SAS engine 61
  syntax 35
data sets
  compressing for SPD Engine 41, 64
  converting for SPD Engine 7, 8, 14
  creating SPD Engine data sets 16
  interim data sets 6
  loading SPD Engine data sets 16
  partition size 48, 66
  threads for SPD Engine data sets 55
DATAPATH= option
  LIBNAME statement, SPD Engine 26
directory paths
  multiple 8
disk-I/O set-up 75
disk striping 14, 76
DLDMGACTION= system option 71

E

ENDOBS= data set option 43
  WHERE expression with 43
ENDOBS= option
  LIBNAME statement, SPD Engine 27
engines
  SPD Engine 3

F

file sharing 8
file systems
  SPD Engine 6

G

group formatting 25

I

I/O performance 8
I/O scalability 85
I/O threads
  number to spawn 55
IDX files
  size estimate 82
IDXWHERE= data set option 45
implicit sorting 9, 24, 39
index area
  assigning 81
  configuring 81
index component files 5
  storing 14, 28
indexes
  efficiency 17
  excluding from WHERE evaluation planner 59
  parallel creation 9, 18, 36
  parallel updates 18
  queries with 9
  segment size 50
  segments in WHERE expressions 65
  sorting values for creating 67
  space requirements for creating 83
  unique indexes 57
  WHERE expressions with 45
INDEXPATH= option
  LIBNAME statement, SPD Engine 28
interim data sets
  temporary storage of 6
IOBLOCKSIZE= data set option 46

L

LIBNAME statement, SPD Engine 23
  introduction 23
  options list 33
  syntax 23
libraries
  allocating space 11
  SPD Engine 6
  storing in temporary subdirectory 32
loading SPD Engine data sets 16

M

MAXSEGRATIO= system option 65
metadata area
  assigning 77
  configuring 77
  space requirements 77
metadata component files 5
  overflow partitions 28
  storing 13, 28
METAPATH= option
  LIBNAME statement, SPD Engine 28
MINPARTSIZE= system option 66
MSGLEVEL= system option 71

N

naming conventions
  component files 16

O

observations
  appending with unique indexes 57
  end number 27, 43
  inserting with unique indexes 57
  meeting conditions of WHERE expressions 70
  number for index segment size 50
  number in compressed blocks 46
  output order of 37
  saving with non-unique key values 57
  starting number 31, 51
  synchronous vs. asynchronous processing 53
output
  physical order in 37

P

PADCOMPRESS= data set option 47
parallel index creation 9, 18, 36
parallel index updates 18
parallelism 12
PARTSIZE= data set option 48
PARTSIZE= option
  LIBNAME statement, SPD Engine 29
performance
  I/O performance 8
  processing performance 9
physical order of output 37
primary path 12
processing performance 9

Q

queries
  indexes with 9

R

RAIDs 14, 76
redundant arrays of independent disks (RAIDs) 14, 76

S

saving observations
  with non-unique key values 57
scalability
  I/O scalability 85
Scalable Performance Data Engine
  See SPD Engine
segments
  size for indexes 50
SEGSIZE= data set option 50
sharing files 8
SMP machines 4
sort utility
  memory space for 67
sorting
  implicit sorting 9, 24, 39
  memory space for 68
  values for index creation 67
SPD Engine 3
  compared with Base SAS engine 5
  converting Base SAS engine data sets 7, 14
  converting SPD Server data sets 8
  data organization 4
  file systems 6
  libraries 6
SPD Server
  converting data sets for SPD Engine 8
SPDECONF.SAS 85
SPDEINDEXSORTSIZE= system option 67
SPDEMAXTHREADS= system option 67
SPDESORTSIZE= system option 68
SPDEUTILLOC= system option 69
SPDEWHEVAL= system option 70
SQL procedure
  partitioning SPD Engine data sets 49
STARTOBS= data set option 51
  WHERE expression with 51
STARTOBS= option
  LIBNAME statement, SPD Engine 31
symmetric multiprocessing (SMP) 4
SYNCADD= data set option 53
system options
  different behavior for SPD Engine 71
  list of 71
  SPD Engine 63
  syntax 63

T

TEMP= option
  LIBNAME statement, SPD Engine 32
temporary storage
  interim data sets 6
THREADNUM= data set option 55
threads 3
  number to spawn 55
  upper limit for SPD Engine 67

U

unique indexes 57
UNIQUESAVE= data set option 57
utility files 6
  storing 69

W

WHERE evaluation planner 9
  excluding indexes 59
WHERE expressions
  ENDOBS= data set option with 43
  excluding indexes 59
  index segments in 65
  indexes with 45
  observations meeting conditions of 70
  STARTOBS= data set option with 51
WHERE optimization 9
WHERENOINDEX= data set option 59
work area
  configuring 84

Your Turn