Scalable Access to SAS Data

Billy Clifford
SAS Institute
SUGI 27

Abstract

Symmetric multiprocessor (SMP) computers can increase performance by reducing the time required to analyze large volumes of data. Using a new engine in Version 9, large SAS data sets can be partitioned and an I/O stream can be initiated for each processor. Within the engine, WHERE expressions are filtered in parallel by assigning a processor to each partition. Outside the engine, SAS applications like Datamining Regression, which uses the DMREG procedure, can initiate parallel I/O paths to the partitions. This new engine is compatible with data sets created by the Scalable Performance Data Server (SPD Server) product.

Scalability through Parallelism

Scalability is all about adding more hardware in order to reduce the real time it takes to complete your data processing request. Buying a faster CPU for your computer may satisfy this definition on paper, but that's not necessarily an adequate solution in practice. Even the fastest processor made today may not be sufficient to process your data rapidly enough. The true key to making large reductions in time is parallel processing. Replacing your single-CPU computer with a four-CPU system, for example, is the type of hardware upgrade that can take advantage of the new V9 scalable features of SAS and provide significant real-time savings.

In CPU-bound applications, CPU scalability occurs when data to be analyzed is in memory and is divided into separate segments. Each segment is assigned a CPU and is analyzed in parallel. The results of each operation are then combined to form the final result. Summarization can be performed in this manner. Such scalability is transparent to the user.

In I/O-bound applications, I/O scalability occurs when data stored on disk are organized to allow multiple I/Os to proceed in parallel. This allows separate I/O streams to be surfaced to the code that is requesting the data. I/O scalability is not completely transparent to the user. The user must ensure that the disks are configured properly and must tell SAS where to put the data (more about this later). I/O scalability is the main focus of this paper.

The best performance comes when you combine both CPU and I/O scalability.

Organizing Data for Scalable I/O

In order to take advantage of I/O scalability, the data must be stored in separately addressable units. Think of these units as partitions. Each partition can then be independently accessed in parallel (assuming you have appropriate hardware). Data separation can be obtained by using different disk devices and/or disks that permit parallel I/O such as RAID (Redundant Array of Independent Disks) devices.

In order for any large data file to span devices and directories, it must be divided into pieces or partitions. The software and operating system that reads these files must be able to surface these multiple physical partitions to the user as a single logical file.

Another aspect of the scalable I/O strategy is storage of separate classes of data in separate files. Data, metadata, and indexes can be stored as separate files on different devices. This allows parallel access to these files.

New LIBNAME Engine – SPDE

The SAS SPD Server has been a standalone client/server product on Windows and UNIX platforms for several years. It supports and utilizes both CPU and I/O scalability. In V9.0, we are taking the key scalable technology from SPD Server and packaging it as a SAS LIBNAME engine. This engine is called SPDE (Scalable Performance Data Engine).

SPDE exploits a hardware and software architecture known as symmetric multiprocessing (SMP). An SMP machine has multiple CPUs and an operating system that supports threads. A thread is a single, independent flow of control through a program. An SMP machine is usually configured with multiple controllers and multiple disk drives per controller. When SPDE reads a file it may launch one or more threads for each CPU. These threads then read data partitions in parallel from multiple disk drives, driven by one or more controllers per CPU. SPDE running on an SMP machine provides the capability to read and deliver much more data to an application in a given elapsed time than ever before.

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 the processing performance. The scalability, in turn, significantly increases overall processing speed for tasks such as creating indexes, appending data, and sub-setting the data by using WHERE expressions.

Components of an SPDE Data Set

An SPDE data set consists of at least two, and possibly four types of files. Each file type has a different designator.

MDF contains all the data set's metadata. This metadata includes column names, labels, index information, and the location of the other files in the data set. There is one MDF file per data set.

DPF contains all the user's data, that is, the rows and columns. Each partition is a separate DPF file, so there may be many DPF files for a given data set.

HBX contains the distinct (unique) values for a given index. For each index, there is one IDX file and one HBX file.

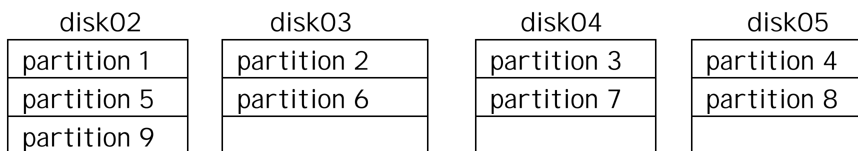
IDX contains the values that occur multiple times.

All four types of SPDE component files can span devices and directories. The metadata and index components span files or directories when their current space is filled.

The concept of partitions is key to understanding the SPDE data file organization. A single SPDE data file is divided into fixed-size physical files called partitions. The user has control over the partition size.

LIBNAME options are used to specify the list of disks (paths) to be used for storage of the files. As data are loaded into the data (DPF) files, the list of data paths is used in a round-robin manner for the partition files. Here's an example with nine data file partitions spread out over four disks.

```
libname cars SPDE
  metapath=("c:\disk01")  /* primary path - metadata location */
  datapath=("d:\disk02"   /* first  data path */
            "e:\disk03"   /* second data path */
            "f:\disk04"   /* third  data path */
            "g:\disk05"); /* fourth data path */
```



Filename Conventions

The SPDE architecture depends upon having a directory-based (hierarchical) file system with support for long filenames. Such file systems are standard on Windows and UNIX platforms. On MVS platforms, SPDE uses the HFS file system. On OpenVMS Alpha platforms, SPDE uses ODS-5 enabled volumes.

SPDE embeds some metadata in the long filenames it uses. Here's the format:

Dsname.FtypeSuffix.Fuid.F#.V#.SPDS9

where

```
Dsname is the data set name
Ftype  is MDF, DPF, IDX, or HBX
Suffix is empty for MDF or DPF files
       is the index name for IDX and HBX files
Fuid   is the name of the primary path
P#     is the partition number
V#     is the version number
SPDS9  is the file extension
```

Here's an example:

- the data set name is "houses"
- the primary path is "u:\v9\test". Note "\" and ":" are changed to "!".
- the index name is "price"

```
houses.mdf.0.0.0.spds9
houses.dpf.U!!v9!test.0.1.spds9
houses.hbxprice.U!!v9!test.0.1.spds9
houses.idxprice.U!!v9!test.0.1.spds9
```

Key SPDE Features

- **Partitioned data files**
  A data set is allowed to span directories and devices, creating a single logical file, possibly larger than can be supported by the underlying operating system. This partitioning is the key to I/O scalability.
- **Hybrid index**
  SPDE uses an index that combines the strengths of a B-tree and a bitmap to provide fast lookup for both unique (low-cardinality) data and multiple occurring (high-cardinality) data. More details about the index later.
- **Parallel block interface to procedures**
  The traditional I/O interface to procedures can only transfer one row at a time via a single thread. This is a single, narrow data path. A new interface allows a procedure to initiate multiple threads and transfer a block of rows at a time for each thread. This interface provides multiple, wide paths to the data.
- **WHERE processor uses multiple indexes**
  The WHERE processor can take advantage of multiple indexes to optimize a WHERE expression. Bitmaps (representing the qualified rows) from each applicable index are combined via AND and OR operations to identify the final subset.
- **Parallel WHERE evaluation**
  Combining both I/O scalability and CPU scalability, SPDE can initiate multiple threads that examine indexes and data file partitions in parallel to evaluate a WHERE expression.

- **Implicit sort for BY**
  SAS BY processing requires the data to be sorted. Usually this requires use of the SORT procedure to sort the data set. SPDE automatically sorts the rows (but not the data set on disk) according to the BY specifications before delivering the rows to the procedure.
- **More than 32K columns**
  SPDE is the only SAS engine that supports more than 32,767 columns.
- **More than 2GB rows**
  SPDE is the only SAS engine that supports more than 2,147,483,647 rows on 32-bit platforms.
- **Data set compatibility between SPDE and SPD Server**
  Data sets created with SPDE can be accessed with SPD Server. Given that data sets created with SPD Server have the correct ACL security, SPDE can access them.
- **Parallel load (with the APPEND procedure)**
  Data from a base engine data set can be loaded into an SPDE data set with the DATA step, the APPEND procedure, and the COPY procedure. When the APPEND procedure is used, the rows are buffered and appended to the data file in blocks. These blocks can be processed by separate threads, both for updating the indexes and appending to the data file.
- **Parallel index creation**
  The user can specify that SPDE create multiple indexes in parallel. All of the indexes are populated by a single scan of the data file. By default, multiple indexes are not created in parallel and a full scan of the data file is made for each index.

Indexes Optimize Access to Your Data

Indexes can improve the performance of WHERE expression processing and BY processing. An index can be created on a single column, or the concatenation of multiple columns (called a composite index). Multiple indexes can be utilized to optimize a given query.

The SPDE index structure is designed to efficiently handle both unique (discrete) data and data with many multiple occurrences. The HBX (or B-tree) component contains a single entry for each unique value, regardless of whether that value has multiple occurrences.

The IDX component uses bitmaps to represent the multiple occurrences for a given value. A bitmap is divided into segments representing groups of contiguous rows in the data set (DPF) component file. A segment may be smaller than a data set partition or it may span partitions. When SPDE performs data queries, SPDE can examine index segments in parallel to identify the qualified segments (that is, the segments that have one or more rows matching the indexed value). The qualified segments are then used to select the specific rows in the data file that satisfy the query.

- If a unique index is specified by the user, the leaf nodes (or terminal nodes) in the B-tree hold the record ID for the row containing the unique value. The IDX component is not used.

- If the index is not unique, the leaf nodes in the B-tree contain a pointer to a list of bitmaps (segments) in the IDX file that contain the value. Each bitmap identifies which rows in the segment contain the value.

Performance-Tuning Knobs

This section identifies some of the options available in SPDE that can impact performance. However, it is always difficult to predict performance changes when a given option is used, because of variables, like your data, your computer, and your specific application. So try these options yourself – twist the knobs and determine how they change the performance of your application.

- **METAPATH=**
  specifies the directories for the metadata file (MDF).
- **DATAPATH=**
  specifies the directories for the partition data files (DPF).
- **INDEXPATH=**
  specifies the directories for the index files (HBX and IDX files).
- **PARTSIZE=**
  specifies the size of each data file partition (DPF).
- **SEGSIZE=**
  specifies the number of rows represented by an index segment (the number of rows represented in the bitmap).
- **COMPRESS=**
  controls whether SPDE compresses the data file. Compressing a data set usually reduces its size (and therefore I/O) at the expense of some added CPU cost. Default is NO.
- **IOBLOCKSIZE=**
  specifies the number of rows to be compressed at a time. This value also determines the size of the I/O buffer.
- **ASYNCINDEX=**
  instructs SPDE to create multiple indexes in parallel. Default is NO.
- **BYSORT=**
  controls whether SPDE sorts the data (not the data set) for a BY statement. Default is YES.
- **SPDESORTSIZE=**
  specifies the maximum amount of memory used by the sort. Note that there may be multiple sorts executing in parallel. So the real amount of memory used by sorts is SPDESORTRSIZE multiplied by the number of concurrent sorts.
- **SPDEINDEXSORTSIZE=**
  specifies the maximum amount of memory used by the sort when creating an index. When indexes are created in parallel (because ASYNCINDEX=YES), the SPDEINDEXSORTRSIZE value is divided up among all the concurrent index creation threads.

- **SPDEMAXTHREADS=**
  sets the upper limit on the number of threads SPDE is allowed to use. In a computer shared by multiple users, it may be important to limit the number of threads used by SPDE to avoid using too many CPUs.
- **NOINDEX=**
  instructs SPDE to ignore all indexes when processing a WHERE expression.
- **WHERENOINDEX=**
  specifies a list of indexes to be excluded from WHERE processing.

Primary Users of SPDE

SPDE is optimized for the storage and sequential read access to large and very large data sets (millions of rows, many GB of data). From what has been presented so far, it should be clear that the best results with SPDE will require a user with:

- a strong SAS background
- solid familiarity with the SPDE tuning knobs
- knowledge of disk and CPU configurations

Minimum and Preferred Hardware Configuration

General guidelines for a computer system to support SPDE include:

- an SMP computer with at least two (minimum) or four (preferred) CPUs
- at least one I/O channel per two CPUs
- enough disk drives to have at least one mount point per CPU isolated on its own disk; two mount points per CPU are better.
- at least 20GB of disk space

Configuring a system to provide at least one separate I/O path to the disks for each CPU is crucial to getting the best performance from SPDE. A sample program that measures I/O scalability is supplied with SPDE to help you maximize your system's configuration.

SPDE or SPD Server?

SPDE and SPD Server share a common heritage and, therefore, share a great number of features and performance benefits. However, there are some important differences, primarily in the execution environment.

SPDE should be considered an entry-level scalable product. It runs as a LIBNAME engine in the SAS environment. SPD Server is a standalone client/server product. Data sets initially developed on SPDE can be migrated to SPD Server with ease, as the need to move to a full client/server environment arises.

- SPD Server requires its computer be mostly a dedicated server. The more you run other kinds of workloads on this server, the more variable your SPD Server performance will be.
- SPD Server requires more skills to set up and administer than SPDE.
- SPD Server supports multiuser client/server access.
- SPD Server supports an Access Control List-based security model.

Sample Performance Results

All the tests described here were run on the following Compaq computer:

- 8 CPUs
- 12 disks, 8 for SPDE data
- 5 disk controllers, 4 of the controllers control the 8 SPDE disks
- 3.9 GB RAM
- Windows 2000.

Parallel Index Creation Example

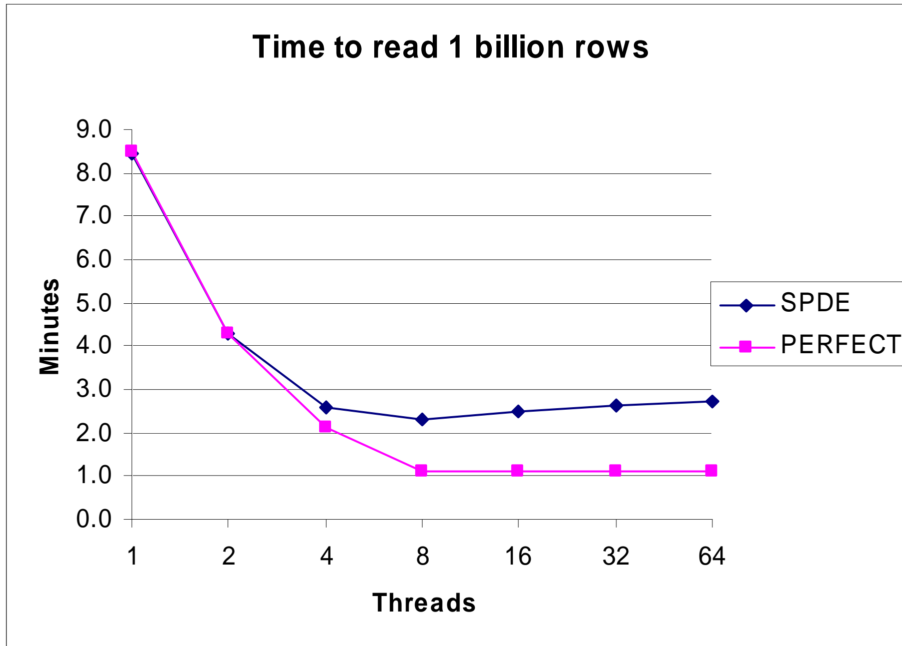
This example compares index creation times for SPDE and the base engine. Three single-column indexes were created. The SPDE indexes were created in parallel. The base engine indexes were created serially.

Number of Rows	SPDE	Base
2,000,000	31 seconds	53 seconds
20,000,000	5.3 minutes	10.1 minutes
200,000,000	1.3 hours	3.4 hours

I/O Scalability Example

Perfect scalability occurs when the real time required to run a job on multiple CPUs is a fraction of the time for one CPU, where the denominator of the fraction is the number of CPUs. For example, if a job takes 24 minutes to run with one CPU, perfect scalability would predict that the job would take 24 / 4 = 6 minutes on a four-processor computer. In reality, perfect scalability is seldom realized because there is an overhead cost involved in managing the threads and merging the multiple results back into a single result.

The test program in this example can be used to determine if a computer is properly configured for I/O scalability. The data set has 2 numeric columns and 1 billion rows. The WHERE expression asks for a record that is not in the data set. Without any indexes, SPDE 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 reaches a plateau at eight threads, which is the number of CPUs on the test machine. Specifying a number of threads larger than 2-3 times the number of available CPUs does not improve performance.



Use of Multiple Indexes and OR Example

This last example shows the power of using multiple indexes and the ability to handle an OR operation in the WHERE expression. The base engine cannot optimize a WHERE expression with an OR operation and it cannot take advantage of multiple indexes.

The data set characteristics are:

- 100 million rows
- 5 columns (one 7-byte character, four 8-byte numeric)
- index on column I (100 million distinct values)
- index on column Y (500 distinct values)

WHERE Expression	SPDE	Base
I = 100	0.17 seconds	0.05 seconds
I = 100 or Y = 3	0.90 seconds	6:29 minutes

As expected, the "**I = 100**" condition is resolved very quickly by both engines, thanks to the index on I. Since the base engine cannot optimize a WHERE expression with an OR, it must make a full scan of the data set for "**I = 100 or Y = 3**." SPDE merges bitmap segments from the index on column I and the index on column Y to rapidly locate the requested rows.