THE POWER TO KNOW.

SAS® Scalable Performance Data (SPD) Server® 4.52 User's Guide

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Insitute Inc.. 2010. *SAS® Scalable Performance Data Server 4.52: User's Guide*. Cary, NC : SAS Institute Inc.

SAS® Scalable Performance Data Server 4.52: User's Guide

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

1st electronic book, May 2010

Contents

Part 1

Product Notes

Chapter 1
SAS Scalable Performance Data (SPD) Server Product Notes

Overview

This document summarizes enhancements and changes in SAS Scalable Performance Data (SPD) Server 4.52, as well as changes and enhancements for SPD Server 4.5. The most recent changes in SPD Server are presented first.

- The SPD Server 4.52 installation includes client modules that are compatible with SAS 9.2.
- SPD Server 4.52 is not compatible with SAS versions earlier than SAS 9.2. Refer to the appropriate SPD Server UNIX or Windows installation guide for more information about SAS software requirements for use with SPD Server 4.52.

What's New in SPD Server 4.52?

Overview of SPD Server 4.52

SPD Server 4.52 has a new server parameter named MAXSORTTHREADS=. Use MAXSORTTHREADS= to provide more control over the number of concurrent threads that are used during an SPD Server parallel sort. For more information about using the MAXSORTTHREADS= server parameter, see, “Setting Up SPD Server Parameter Files,” in the *SAS Scalable Performance Data Server 4.52: Administrator's Guide*.

The NOMISS= index support parameter is no longer recognized in SPD Server 4.52. If you submit the NOMISS= parameter to SPD Server 4.52, the parameter declaration is ignored.

The SPD Server parameter file is enhanced to include performance-level parameters that you can specify for different classes of users. Each class of users can be associated with a specific set of SPD Server parameter settings in order to regulate SPD Server resources for each class. A user's resource class is defined by the performance class attribute that is specified in the user's SPD Server Password Manager database record. The configurable user resource class attribute levels are LOW, MEDIUM, HIGH, and LOCKING record. For more information about assigning SPD Server users to configurable user resource classes, see the section, “Server Performance Levels,” in the *SAS Scalable Performance Data Server 4.52: Administrator's Guide*.

The SPDSCLEAN utility now removes temporary directories and files that were created using the TEMP=YES LIBNAME parameter. In the past, orphaned directories and files could be left behind if the SPDSBASE process terminated abnormally or was killed. SPDSCLEAN now checks for temporary directories in LIBNAME domains during the standard *–libnamefile* cleanup. The orphan directory check and cleanup function is integrated into the SPDSCLEAN utility. You do not need to specify an argument for SPDSCLEAN to remove orphaned TEMP=YES LIBNAME directories and files. For more information, see “SPD Server Directory Cleanup Utility” in the *SAS Scalable Performance Data Server 4.52: Administrator's Guide*.

The 2 GB size limit on SPD Server files on Windows Win32 platforms has been removed. Win32 platforms can now create and access files (.dpf partitions, index files, and sortbins) that are greater than 2 GB in size. SPD Server on Win32 now supports files up to ($2^{63} – 1$) bytes in size, the same file size supported by SPD Server on 64-bit UNIX platforms.

SPD Server 4.52 for Windows supports the Windows 2008 R1 operating system running in 32-bit compatibility mode.

Automatic maintenance and cleanup of SPD Server directory **.spdslibll** files have been implemented. The **.spdslibll** directory files are cleaned up when SPD Server starts up and reads the **libnames.parm** file. In fact, the **.spdslibll** directory files are cleaned up whenever SPD Server refreshes the **libnames.parm** file. This ensures that the path information that is contained in **.spdslibll** files remains current. SPD Server creates a **.spdslibll** file in a domain if a user connects to a domain where there is no **.spdslibll** file. The enhanced **.spdslibll** file management strategy supports changing data partition paths and changing index file paths for existing LIBNAMES in SPD Server.

A new SPD Server reset option called SQLHIMEM has been added to SPD Server SQL. SQLHIMEM enables you to allocate and deallocate large blocks of memory for certain SQL queries. For more information about using SQLHIMEM in SPD Server SQL, see “SQLHIMEM” on page 107.

The following SPD Server messages have been changed from WARNINGS to NOTES:

- **`Directory created from LIBNAME assignment.`** This message is displayed when a TEMP=YES LIBNAME assignment is created.
- **`Read-only access to LIBNAME domain restricted by *LIBNAM* ACL.`** This message appears if you try to make a LIBNAME assignment within a read-only domain.
- **`Parallel WHERE evaluation suppressed due to sort order on table.`** This message appears when a sorted table is filtered by a WHERE clause. “Variables for WHERE Clause Evaluations” on page 214For more information, see
- **`WHERE clause requires SAS filtering support which could affect performance because SPD Server could not fully evaluate the expression.`**
- **`Because ASYNC operations create indexes in parallel, the status for all defined indexes will be determined at closing time.`** This message appears during asynchronous index creation.

Base SAS software can now read views that are created by SPD Server, and SPD Server can read views created in Base SAS software. That is, Base SAS software can read SQL views that are created in SPD Server by using explicit pass-through SQL, and SPD Server can read SQL views that are created in Base SAS software by using SAS PROC SQL. There are restrictions on how Base SAS can use some SPD Server SQL views. For more information, see “SPD Server Views” on page 135.

SPD Server 4.52 includes a new security feature called TABLE WHERE constraints. TABLE WHERE constraints enable the owner of an SPD Server table to use a WHERE clause to filter or control how the table can be accessed by other SPD Server users. TABLE WHERE constraints can be used with SPD Server symbolic substitution to implement row-level security by filtering table rows based on User ID, Group ID, or ACLSPECIAL attribute settings. For more information, see the section, “TABLE WHERE Constraints,” in the *SAS Scalable Performance Data Server 4.52: Administrator's Guide*.

What's New in SPD Server 4.5?

Overview of SPD Server 4.5

The operating system requirements for SPD Server 4.5 have changed from the operating system requirements for SPD Server 4.4. For more detailed information about operating system requirements for SPD Server 4.5, see the Chapter 2, "SPD Server Pre-Installation and System Requirements Guide," in the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.

- SPD Audit logging has been enhanced to include the user LIBNAME in the proxy and SQL audit logs. For additional information, see the section on SPD Server Auditing in Chapter 14, ACL Security Overview, of the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.
- You can now specify recycle times for the SPD Server Name Server log and the SPD Server SNET log. For additional information about configuring SPD Server log cycle times for Windows installations, see the section, "Configuring SPD Server Software on your Windows Host" in Chapter 4, "SPD Server Windows Installation Guide," of the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*. For additional information about configuring SPD Server log cycle times for UNIX installations, see "Configuring SPD Server Host Software for Your Site" in Chapter 3, "SPD Server UNIX Installation Guide," of the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.
- SPD Server now supports user formats with the put() function that are greater than 8 characters in length. An SPD Server host can read user format catalog files that were created by SAS running on Windows, or on the same machine as the SPD Server host. The spdsls list utility has been enhanced to add a **`-verbose`** option. The **`-verbose`** option provides information such as the number of observations, observation length, index segment size, partition size, and whether the table is compressed, encrypted, or is a member of a cluster. For more information about SPD Server list utilities, see "SPD Server Table List Utility Spdsls," in Chapter 18 of the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.
- SAS implicit Pass-Through SQL now permits SQL queries to SPD Server that include supported SPD Server functions. contains a section, Chapter 8 of the SAS Scalable Performance Data (SPD) Server 4.5: User's Guide, "SPD Server SQL Features" contains a section “Differences between SAS SQL and SPD Server SQL ” on page 146that lists the functions that SPD Server supports via implicit Pass-Through SQL.

- The installation and delivery of the SPD Server 4.5 client components for SAS is now part of your SAS installation. For more detailed information about installing SPD Server 4.5 on a Windows platform, see Chapter 4, "SPD Server Windows Installation Guide," of the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*. For more detailed information about installing SPD Server 4.5 on a UNIX platform, see Chapter 3, "SPD Server UNIX Installation Guide," of the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.
- The installation and delivery of SAS Management Console components for SPD Server 4.5 is now part of your SAS Management Console installation. For more detailed information about installing SAS Management Console components for SPD Server 4.5 on a Windows platform, see "Before You Install: Precautions and Required Permissions" in Chapter 4, "SPD Server Windows Installation Guide" of the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*. For more detailed information about installing SAS Management Console components for SPD Server 4.5 on a UNIX platform, see, " Before You Install: Precautions and Required Permissions," in Chapter 3, "SPD Server UNIX Installation Guide" of the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.

SPD Server 4.5 Platform Support Changes

SPD Server 4.5 now supports the Linux x64 platform.

Part 2

SPD Server Usage

Chapter 2 SAS Scalable Performance Data (SPD) Server Overview

Introduction to SAS Scalable Performance Data (SPD) Server

SAS Scalable Performance Data (SPD) Server software is designed for high-performance data delivery. Its primary function is to provide user access to SAS data for intensive processing (queries and sorts) on the host server machine. When client workstations from varying operating platforms send processing requests to an SPD Server host, the host returns results in the format required by each client workstation. SPD Server uses the power of parallel processing to exploit the threading capabilities of servers with multiple processors.

SPD Server executes threads, units of processing, in parallel on an SPD Server host. The software tasks are performed in conjunction with an operating system that enables threads to execute on any of the machine's available processors. A specialized machine and operating system are important processing partners, but SPD Server's power is derived from the software architecture that enables it to rapidly and efficiently process SAS data in concurrent parallel threads on multiple processors .

SPD Server is the high-speed processing tool among SAS products. SPD 4.5 provides on-disk structures that are compatible with SAS 9 and the large table capacities that it supports. Enterprise-wide data mining often creates immense tables. In order to generate business intelligence quickly, the ability to update tables that contain billions of rows is more important than ever. The cluster table structure provides a new foundation for the next generation of SAS data storage. Previous versions of SPD Server were based on 32-bit architecture that supported just over 2 billion rows and 32,768 columns. SPD Server is based on a 64-Bit architecture that supports tables with over nine quintillion rows and over 2 billion columns.

SPD Server 4.5 operates on computers running SAS 9.2 or later. PC users that do not use SAS can still use SPD Server. Information about connecting to SPD Server with Other Clients is found in Chapter 6, “Using SAS Scalable Performance Data (SPD) Server with Other Clients,” on page 55. SAS users can access SPD Server either by using SQL Pass-Through or by using SAS language.

Syntax Conventions: SPD Server software supports SAS users and other users. The SPD Server document uses common terminology that both audiences should understand. In the SPD Server documentation, SAS data sets are referred to as tables, SAS variables are referred to as columns, and SAS observations are referred to as rows. The SPD Server product is referred to as SPD Server or the software, depending on the context of the documentation.

The SPD Server Client/Server Model

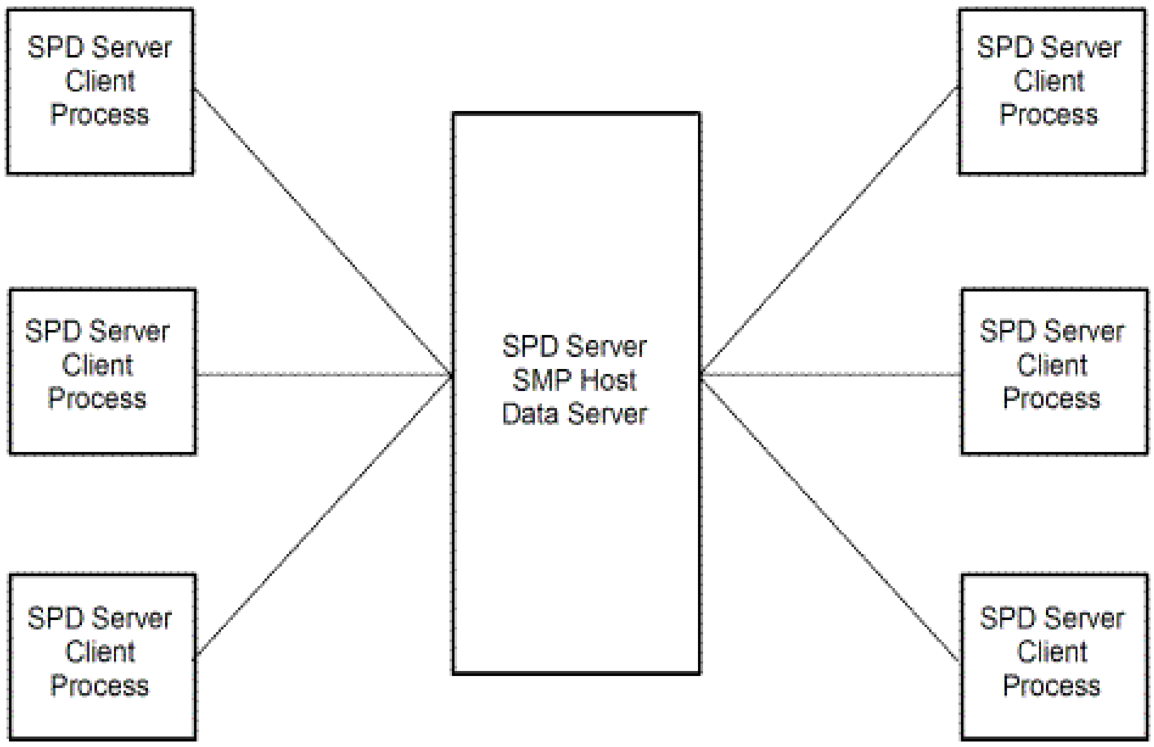
Overview of the Client/Server Model

SPD Server software divides SAS processing loads between the client and server. The Figure 2.1 on page 11 diagram shows a simple client/server topology. The server hosts multiple concurrent clients while performing the heaviest processing tasks. Typical clients are desktop PCs or low-end UNIX workstations running front-end software. The front-end application sends the client's data requests over the network to the server and processes the information that the server returns.

You can create one or more SPD Servers on the host server machine. When an SPD Server host receives a client's data request, it performs some action on behalf of the client. The action varies with the request received.

Where does the user fit within in the SPD Server Client/Server model? Users initiate SPD Server client sessions. In this documentation, the term 'user' refers to the operator of an SPD Server client.

Figure 2.1 *The SPD Server Client/Server Model*


SPD Server Client Process
SPD Server Client Process
SPD Server Client Process
SPD Server SMP Host Data Server
SPD Server Client Process
SPD Server Client Process
SPD Server Client Process

Symmetric Multi-Processor Hosts

SPD Server host machines use operating systems that can process concurrent threads in parallel on multiple processors. SPD Server exploits symmetric multiprocessing (SMP) hardware and software architecture.

The number of processors on an SMP server varies by manufacturer and model. The operating system of the machine must also support the parallel processing. Operating systems that contain a threaded kernel enjoy enhanced performance because the threaded kernel prevents contention issues among competing threads in real time. Synergy between processors and threads allows SPD Server to scale processing performance. The scalability, in turn, significantly improves the speed of SPD Server table creates, appends, scans, queries, and sorts.

SPD Server Host Services for Clients

SPD Server hosts provide multiple services to SPD Server clients:

- **Access to data stores** SPD Server offers concurrent read access and retrieval of SAS data.
- **High-speed data server** SPD Server manages and processes massive SAS tables.
- **Offloads heavy processing work** SPD Server divides the labor. The Server process retrieves, sorts, and subsets SAS data. A client process reviews and analyzes the data that the Server returns.

- **Embellishes client hardware** SPD Server host machines are able to use the computing hardware resources that are required to process large tables efficiently and rapidly.
- **Reduces network traffic** SPD Servers read, sort, and subset entire SAS tables and then return answer sets. A query subset replaces large file downloads to the client machine. SPD Server also offers a common storage facility. Multiple client users can use the same SAS data on the server without having to each transfer the SAS data to their workstations.
- **Provides multi-platform support** SPD Server allows clients to share SAS data across computing platforms with other SAS users.

Table 2.1 *SPD Server Features*

SPD Server Feature	SPD Server Client Action	SPD Server Host Response
Support for Gigabytes of data	The SPD Server client inputs existing SAS tables with a PROC COPY statement or creates an SPD Server table using a SAS DATA step or procedure. SPD Server clients can also use SQL Pass-Through CREATE, COPY, or LOAD statements to input SAS tables.	The SPD Server host creates component files that consist of one or more physical partition files. The server stores the physical partition files in one or more device / directory paths.
Scalable Symmetric Multiple Processor (SMP) Support	The SPD Server client runs SAS procedures and SQL Pass-Through syntax to read, sort, index, or query an SPD Server table.	The SPD Server host uses its threaded operating system to perform concurrent processing tasks distributed across multiple processors.
Selective Parallel Queries	The SPD Server client uses WHERE clause or SQL SELECT syntax. Pass-through SQL, PROC SQL, and WHERE alternatives not in SAS are supported.	The SPD Server host supports and subsets SPD Server tables, and then delivers query answer sets to clients.
Parallel Loads	The SPD Server client runs SAS procedures with LOAD or COPY to store SAS data and indexes.	The SPD Server host uses multiple threads to load and store tables and indexes.
Parallel Indexes	The SPD Server client creates table indexes using a DATA step or the DATASETS procedure with an INDEX option, or Pass-Through SQL with the LOAD or COPY command.	The SPD Server host creates SPD Server table indexes in parallel.
SAS Data Security	The SPD Server client accesses the SPD Server host using SQL Pass-Through, a LIBNAME statement, or an alternative connection that is not in SAS.	The SPD Server host secures SPD Server files at the LIBNAME domain and / or table, column, and row level.

Accessing SPD Server Using SAS

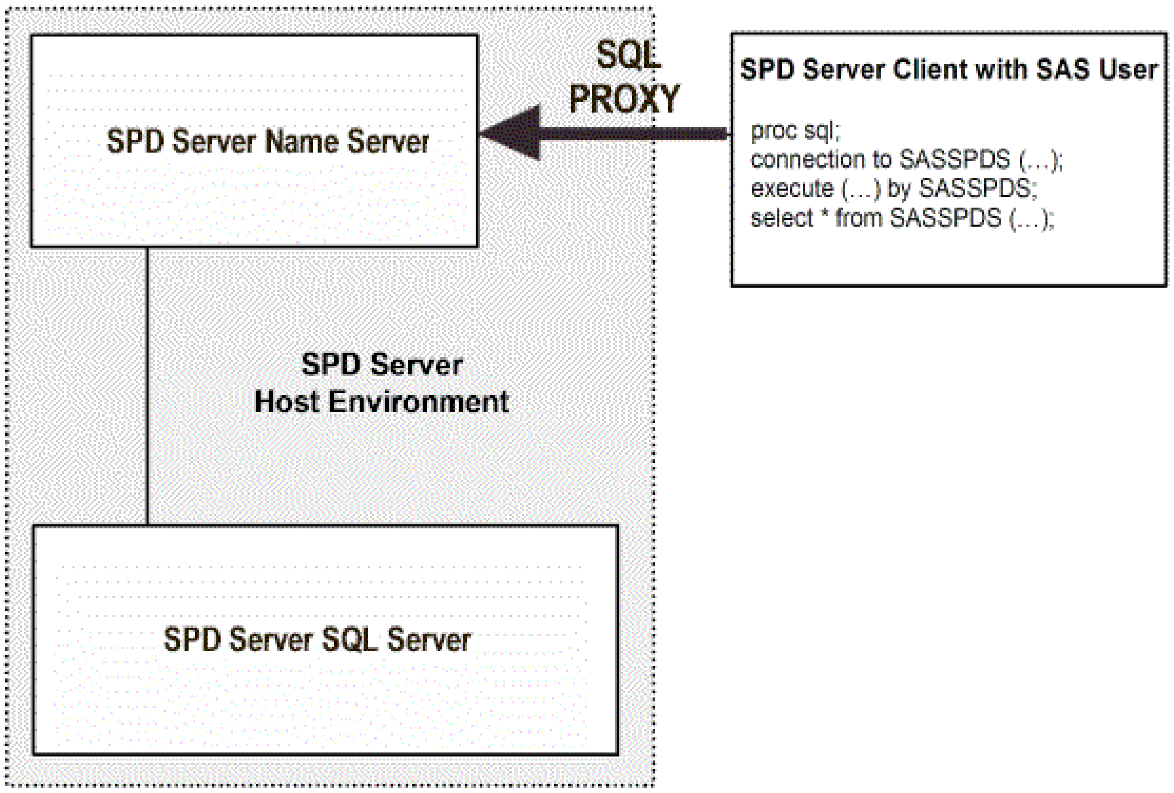
You begin an SPD Server session by starting your SPD Server client. There are two ways to start your SPD Server client session. You can use SQL Pass-Through commands to start your SPD Server client session, or you can use a LIBNAME statement to start your SPD Server client session. Both methods use the SASSPDS engine and initiate communication between the SPD Server client machine and SPD Server host.

SQL Pass-Through Facility

SPD Server can use SQL Pass-Through commands. The SPD Server host can perform complete SQL-expression evaluation. SPD Server also supports nested SQL Pass-Through commands. Nested SQL Pass-Through commands permit you to connect to other SPD Server hosts while you are still connected to your SPD Server host. You can use nested Pass-Through commands to distribute simultaneous SQL queries across multiple SPD Server hosts on your network.

The SQL Pass-Through Facility can be accessed with or without SAS syntax and applications. You can use SAS to connect to an SPD Server host by using Pass-Through syntax from PROC SQL or from other SQL-aware SAS applications. The chapter on Chapter 4, "Accessing and Creating SAS Scalable Performance Data (SPD) Server Tables," on page 33 contains more detailed information about the SPD Server Pass-Through Facility and provides examples of the syntax.

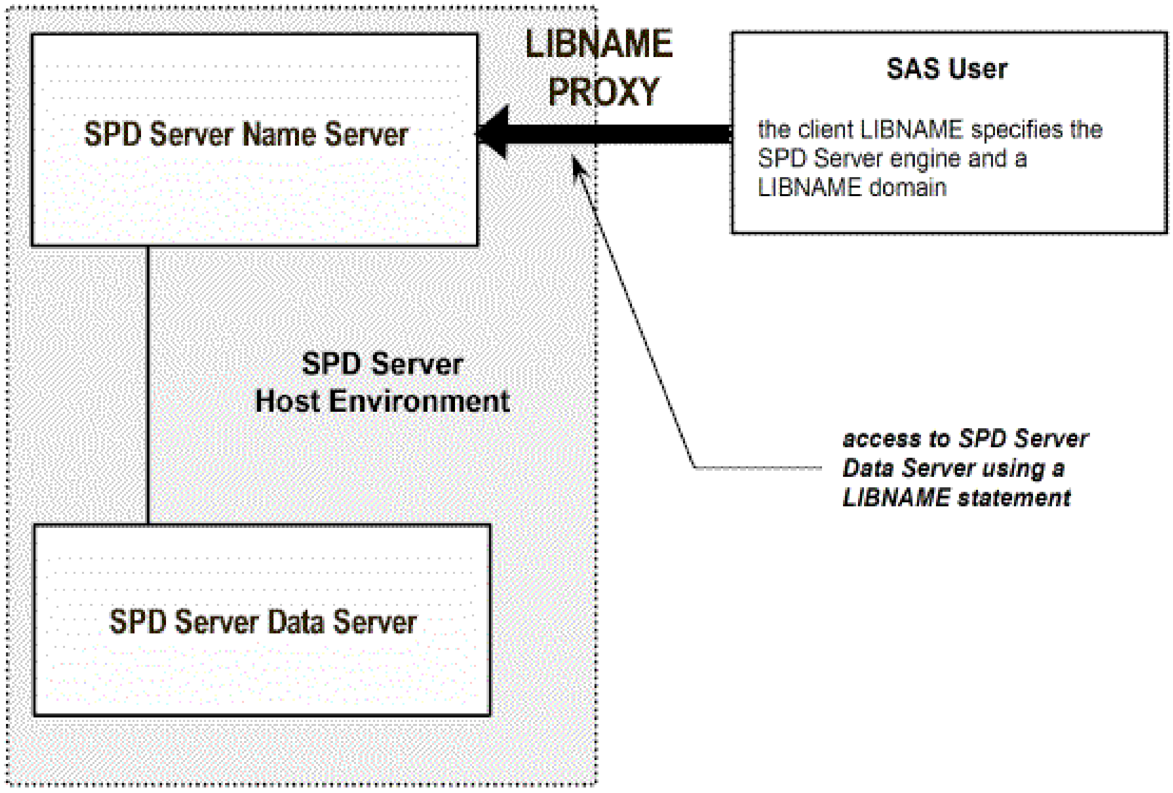
Figure 2.2 *SPD Server Client Access to SPD Server Host Using SQL Pass-Through and SAS CONNECT*


SPD Server Name Server
SQL PROXY
SPD Server Client with SAS User
proc sql;
connection to SASSPDS (...);
execute (...) by SASSPDS;
select * from SASSPDS (...);
SPD Server Host Environment
SPD Server SQL Server

LIBNAME Access

SAS users can initiate a client session by issuing a LIBNAME statement using the engine SASSPDS. LIBNAME access is illustrated in Figure 1.3. The documentation chapter on Chapter 3, “Connecting to SAS Scalable Performance Data (SPD) Server,” on page 21 explains the mechanics of LIBNAME access to the engine and SPD Server LIBNAME options.

Figure 2.3 *SPD Server Client (SAS User) Access to SPD Server Host Using a LIBNAME Statement*


LIBNAME PROXY
SPD Server Name Server
SAS User
the client LIBNAME specifies the SPD Server engine and a LIBNAME domain
SPD Server Host Environment
access to SPD Server Data Server using a LIBNAME statement
SPD Server Data Server

SPD Server Host Name Server

Distributed computing can enrich user resources, but it has an inherent problem. To connect to an SPD Server, you must know its location within your network. Instead of requiring users to memorize long paths or IP addresses, SPD Server software uses a specialized server called a Name Server. The SPD Server Name Server locates active SPD Server hosts on your network. A Name Server recognizes active SPD Server machines because all the SPD Servers 'register' with the Name Server as they come up and contact the host machine.

The Name Server keeps network addresses and a list of the LIBNAME domains for each SPD Server host. What is an SPD Server LIBNAME domain? An SPD Server LIBNAME domain is a logical entity that SPD Server creates. A LIBNAME domain maintains domain attributes such as the library name, owner, and contents. Whenever you use a LIBNAME statement to specify a LIBNAME domain, a Name Server can determine the correct directory path to the SPD Server data library and connect your SPD Server client to the SPD Server host for that domain.

Specifying the Port Address for the Name Server

SPD Server clients use port addressing to locate SPD Server Name Servers. SPD Server administrators must assign a port address to a Name Server. Most UNIX system clients use their local `/etc/services` file to register port assignments. The service name for an SPD Server Name Server in an `/etc/services` file must be **SPDSNAME**. PC clients use services files to register port assignments. The services files on PC clients vary according to the software that the PC network uses.

When a client SPD Server application issues a LIBNAME statement that does not contain the port address of the Name Server, SPD Server checks the services file for the SPDSNAME entry and the port address. Registering the Name Server port assignment in your client's network services file relieves you from the responsibility of coding Name Server port numbers when you write SAS jobs. For examples of using a LIBNAME statement to connect, see "LIBNAME Example Statements " on page 28.

Securing SAS Data

LIBNAME Domain Registration

The Name Server helps SPD Server clients locate and connect to SPD Server hosts. The Name Server also controls access to the SPD Server LIBNAME domains. How does the Name Server get domain information? The SPD Server administrator defines LIBNAME domains in an SPD Server LIBNAME parameter file.

When an SPD Server administrator brings up a server on the host machine, SPD Server reads the **spdssrv.parm** parameter file and registers the domains that are listed in the parameter file with the Name Server. The Name Server remembers which SPD Server host or hosts have access to a given LIBNAME domain. If you want to specify a LIBNAME domain, you can do so using a LIBNAME statement or a Pass-Through SQL CONNECT statement. Your SPD Server administrator can provide you with a list of the LIBNAME domains that are mapped to your SPD Server host machine.

ACL File Security

SPD Server uses Access Control Lists (ACLs) and SPD Server user IDs to secure domain resources. You obtain your user ID and password from your SPD Server administrator.

SPD Server also supports ACL groups, which are similar to UNIX groups. SPD Server administrators can associate an SPD Server user as many as five ACL groups.

ACL file security is turned on by default when an administrator brings up SPD Server. ACL permissions affect all SPD Server resources, including domains, tables, table columns, catalogs, catalog entries, and utility files. When ACL file security is enabled, SPD Server only grants access rights to the owner (creator) of an SPD Server resource. Resource owners can use PROC SPDO to grant ACL permissions to a specific group (called an ACL group) or to all SPD Server users.

The resource owner can use the following properties to grant ACL permissions to all SPD Server users:

READ
: universal READ access to the resource (read or query).

WRITE
: universal WRITE access to the resource (append to or update).

ALTER
: universal ALTER access to the resource (rename, delete, or replace a resource and add, delete indexes associated with a table).

The resource owner can use the following properties to grant ACL permissions to a named ACL group:

GROUPREAD
: group READ access to the resource (read or query).

GROUPWRITE
: group WRITE access to the resource (append to or update).

GROUPALTER
: group ALTER access to the resource (rename, delete, or replace a resource and add, delete indexes associated with a table).

Organizing SAS Data

SPD Server Tables

SPD Server software alters SAS tables to enable high-performance processing. SPD Server tables are physically different from a Base SAS table. You can use tables in either SAS or native SPD Server format. The SPD Server User's Guide chapter on Chapter 4, “Accessing and Creating SAS Scalable Performance Data (SPD) Server Tables,” on page 33 discusses how a simple SAS PROC COPY statement handles “Migrating Tables between SAS and SPD Server” on page 41.

How are SAS tables organized? SAS tables stores a single file that contains the data descriptors and the table data. The data are column values, the descriptors are metadata that describe the column and data formatting that the table uses.

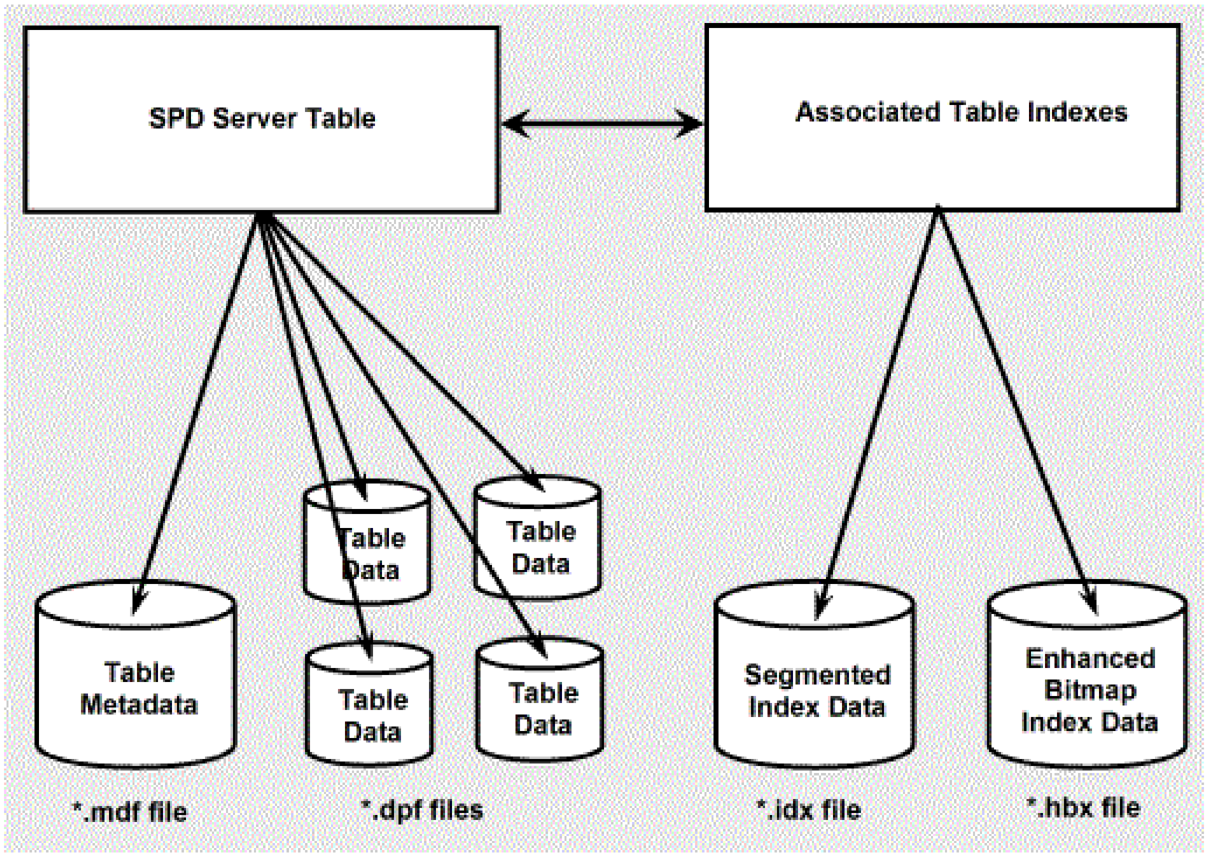
SPD Server tables do not reuse space. When an SQL command to delete one or more rows from a table is issued, the row is marked deleted and the space will not be reused. To recapture the space, the table must be copied.

The diagram of the Figure 2.4 on page 17 shows differences in the architecture between SPD Server tables and SAS tables. SPD Server uses component files to store tables. One component file stores the stream of data values. Another component file stores the column and data descriptors, the metadata. If you create an index for a column or a composite of columns, SPD Server creates component files for each index.

SPD Server Component Files

SPD Server uses four types of component files to store SPD Server tables. The diagram of the Figure 2.4 on page 17 shows the components of SPD Server tables. Two component files store table information: the *.dpf component file stores a stream of the table's data values, and the *.mdf component file stores the table's metadata (column and data descriptors) information. SPD Server also creates two more component files to manage index data: *.hbx components are unique global B-tree indexes and *.idx components are segmented views of the indexed column data. The *.idx components are especially useful in evaluating parallel WHERE clauses.

Figure 2.4 *SPD Server Component Files*



SPD Server partitions component files when they are created to keep them from growing too large. Each partitioned component file is stored as one or more disk files. There are several advantages to partitioning the component files:

- **Very Large Tables**: SPD Server bypasses file size limits imposed by many applications and operating systems. By using partitioned component files, SPD Server can support any file system transparently.
- **Multiple Directory Paths:** SPD Server can access data libraries that span numerous directory paths and storage devices. SPD Server software partitions massive data libraries into component files. The component architecture enables rapid threaded data access while circumventing device capacity and file size limitation issues. Storage lists transparently track component file locations so users can access multiple storage devices as a single volume, even if file partitions exist in different locations.
- **Flexibility in Storage**: There is no need to store data tables and associated indexes in the same location when using SPD Server component files. Data files and associated indexes can be stored on different directory structures or devices if you want. When deciding where to store component SPD Server tables, you only need to consider the cost, performance, and availability of the disk space.
- **Improved Table Scan Performance**: Data component partitions that are created using fixed-size intervals will perform aggressively during parallelized full-table scans. The documentation chapter on "SPD Server Table Options " on page 28 contains information about how to use the PARTSIZE= option to control partition size.

SPD Server Table Indexes

SPD Server allows you to create indexes on table columns. SPD Server can thread WHERE clause evaluations for tables that are not indexed. Indexes enable more rapid WHERE clause evaluations. Large tables in particular should be indexed to exploit SPD Server performance. More information about the SPD Server index is provided in “ Indexing a Table ” on page 49.

SPD Server Performance Enhancements

SPD Server Pass-Through SQL Enhancements

You can use Pass-Through SQL to submit SQL statements that use SPD Server tables directly to SPD Server. The SPD Server SQL planner contains several optimizations that you can use to create SQL queries that can take advantage of symmetric multiprocessing and SPD table indexes, resulting in improved SQL query performance. Refer to the SPD Server User's Guide section on the Chapter 8, “SPD Server SQL Features ,” on page 95 for more information about SPD Server Pass-Through SQL enhancements.

Implicit and Explicit Server Sorts

You can use implicit or explicit sorts with SPD Server. For example, the PROC SORT in Base SAS software is an explicit sort. You can use PROC SORT with SPD Server as well.

An implicit sort is unique to SPD Server. Each time you submit a SAS statement with a BY clause, SPD Server sorts your data -- unless the table is already sorted or indexed on the BY column. The automatic sort is very convenient. The documentation chapter on Chapter 4, “Accessing and Creating SAS Scalable Performance Data (SPD) Server Tables,” on page 33 contains tips on how and when to use each sort type.

Modified SAS Heapsort

SPD Server uses Heapsort as its default sort with some slight changes. Under SPD Server, Heapsort compares available memory on the server to the memory required to load and process the index key data in memory. If the memory is not constrained, SPD Server performs the Heapsort in RAM memory.

Indexed Parallel Table Scan

SPD Server indexes are designed to support parallelism. Experienced RDBMS users are accustomed to a perceptible processing lag that occurs when databases must read or process enormous tables. When SPD Server performs table queries, the SPD Server index architecture enables the software to analyze different table sections or segments in parallel. By processing large table segments in parallel, SPD Server delivers much faster data throughput. The faster throughput might be difficult to perceive on small tables, but when SPD Server performs scans on very large tables, the processing performance is significantly faster than database systems that support only serial indexed table scans.

Improved Table Appends

SPD Server decomposes table append operation 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 table. The more indexes you have, the greater the potential exploitation of parallelism during the append processing.

Tip: You can save time by creating an empty table in SPD Server, define your indexes on it, and then append the data, as opposed to loading the table, and then creating the indexes afterward. It is faster to create indexes on an empty table.

SPD Server Extensions to Base SAS

You can access SPD Server by using an SQL Pass-Through CONNECT statement or you can issue a SAS LIBNAME statement. After connecting to SPD Server, you can run SAS DATA steps, SAS procedures, or PROC SQL statements.

The documentation in the SPD Server Administrator's Guide and the SPD Server User's Guide furnish syntax and examples that use SPD Server extensions to Base SAS language. Most of your existing SAS programs will work in SPD Server with only minor modifications.

SPD Server extensions to the Base SAS language include:

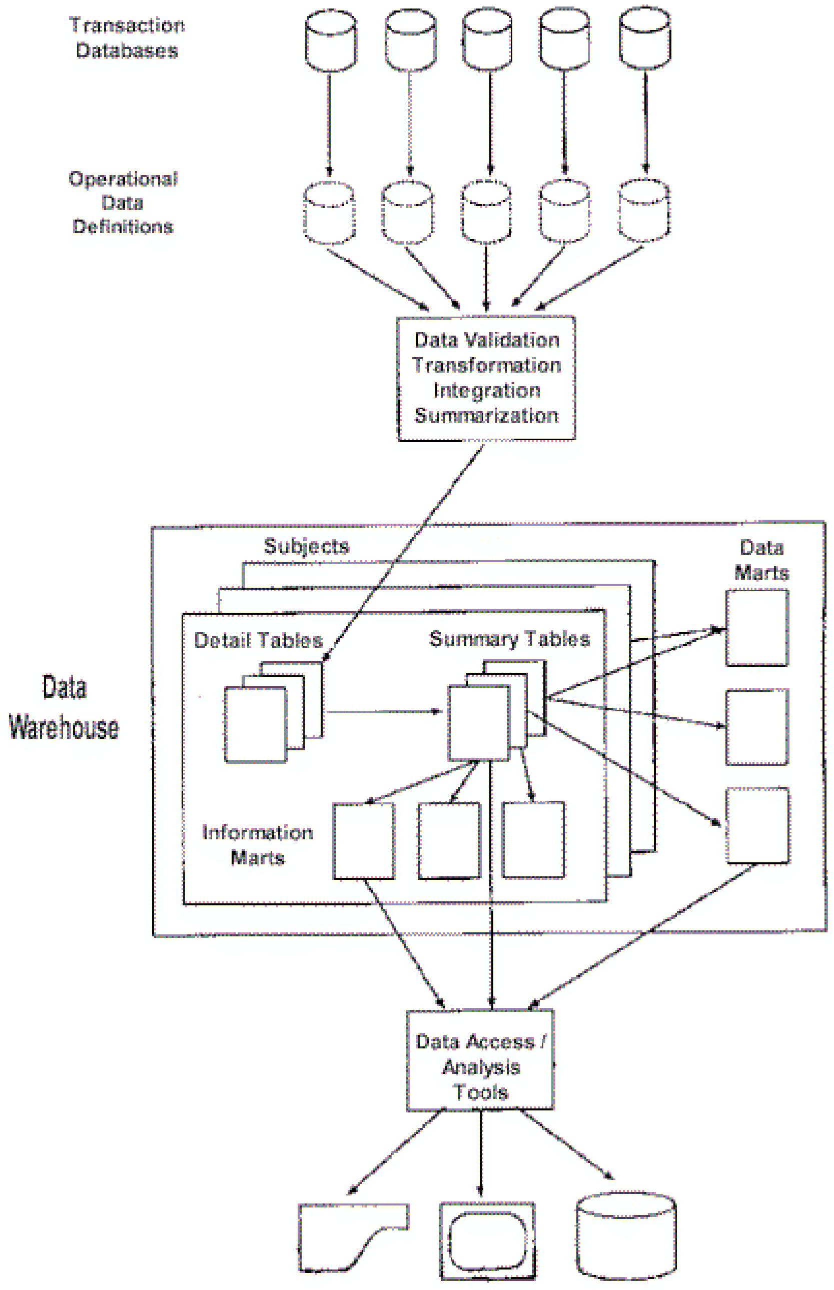
- new LIBNAME statement options
- SPD Server SQL Pass-Through syntax
- new table options
- new macro variables
- parallel WHERE clause processing
- parallel GROUP BY processing
- BY data grouping
- parallel index creation
- PROC SPDO, an operator interface procedure

Using SPD Server with Data Warehousing

SPD Server offers SAS Data Warehousing customers an excellent facility to store data. Using component files and partitioning, SPD Server alleviates large table constraints such as device or directory size limits. SPD Server can perform storage services on a reliable and relatively inexpensive machine.

Besides providing efficient, economical storage, SPD Server can deliver the enhanced processing capabilities users need to manage and query data in a warehouse. SMP processing furnishes the machine's horsepower to parallel-process huge tables. SPD Server also offers multiple access, domain protection, and table locking: these features enable data warehouse users to secure and access their shared SPD Server.

Figure 2.5 Data Warehouse With Large Data Stores


Transaction Databases
Operational Data Definitions
Data Validation Transformation Integration Summarization
Subjects
Data Marts
Detail Tables
Summary Tables
Data Warehouse
Information Marts
Data Access / Analysis Tools

Within a data warehouse, there are several data stores (repositories for data). Three stores are of interest above: detail tables, summary tables, and data marts. Organizations often store transactions that are up to 90 days old in a detail store, transactions that are up to a year old in a summary store, and additional data snapshots in data marts. The three data stores share a common requirement -- they must maintain hundreds of gigabytes of data.

To perform queries, data warehouse users can use the SAS System with SAS syntax or PROC SQL syntax. Alternatively, the software supports use of other vendors' applications that allow Pass-Through SQL and comply with other connection standards that are not SAS. In brief, SPD Server can contribute significantly to objectives for a data warehouse: to deliver low-cost, relevant, machine-independent, and timely information to users throughout the organization.

Chapter 3

Connecting to SAS Scalable Performance Data (SPD) Server

Introduction

All SAS users should read the Help section on Chapter 4, “Accessing and Creating SAS Scalable Performance Data (SPD) Server Tables,” on page 33 to review the methods that they can use to access SPD Server. These methods include LIBNAME statements and SQL

Pass-Through statements. Syntax statements and options are provided for each method, as well as useful table options and macro variables.

SAS and SPD Server Tables

Overview of SPD Server Tables

SPD Server tables have different physical structures than SAS tables. In a general discussion, a SAS table can also refer to an SPD Server table. If the context is specific (for example, an SPD Server command), then the reference is specific. A SAS table refers to the Base SAS format; an SPD Server table refers to the SPD Server format.

Using SPD Server and SAS together, you can

- convert tables from the Base SAS format to the SPD Server format
- convert tables from the SPD Server format to the Base SAS format
- create a new SPD Server table
- read, query, append to, update, sort, and index SPD Server tables.

SAS Libraries

The term 'SAS library' refers either to a collection of SAS files or SPD Server files. For SPD Server, a SAS library, or data library, is a collection of one or more directories that specify the location of stored SPD Server files. A data library has a primary file system. This is the directory an SPD Server administrator defines for the LIBNAME domain when it is set up. In addition, a SAS library can have other directories for separation of SPD Server component files.

An SPD Server data library can contain the following LIBNAME domain files:

- SPD Server tables
- SPD Server indexes
- SPD Server catalogs
- SPD Server ACL files
- SPD Server utility files, such as a VIEW, an MDDB, and so on

Temporary LIBNAME Domains

SPD Server allows you to create temporary LIBNAME domains that exist only for the duration of the LIBNAME assignment. Using this capability, SPD Server users can create space analogous to the SAS WORK library. To create a temporary LIBNAME domain, use the SPD Server LIBNAME statement option, TEMP=YES.

When you end your SPD Server session, all the data objects, including tables, catalogs, and utility files in the TEMP=YES temporary domain are automatically deleted. This is similar to how the SAS WORK library functions.

SPD Server Resource Security

SPD Server provides two levels of data security: UNIX file security and ACL file security. ACL file security enforces SPD Server permissions with SPD Server user IDs and Access Control Lists (ACLs).

UNIX File Security

The software enables ACL file security by default. While ACL file security is strongly recommended, the default can be changed. Only an SPD Server administrator can change the default file security setting. When an SPD Server administrator specifies the NOACL option, all clients for SPD Server obtain the SPD Server user ID 'anonymous'. There is no SPD Server security in effect. SPD Server tables are then secured only by the UNIX file protections that are currently in force.

When UNIX file security controls SPD Server file access, it validates on the user ID associated with SPD Server. Which UNIX user ID is associated with SPD Server? The UNIX ID associated with SPD Server is the UNIX ID of the user that brings up the server. Suppose an SPD Server administrator brings up the SPD Server host machine, using his SPD Server administrator's account named SPDSADMN. When any SAS client connects to this SPD Server host, they will be able to read only files that have UNIX Read permissions set for the SPDSADMN user. As a result, SAS clients that are connected to this SPD Server host must write all files in a directory created by SPDSADMIN that also has Write permission set for SPDSADMN. SPDSADMN will own all files written in this directory.

How is security maintained? The SPD Server administrator can set up the SPD Server LIBNAME domain directories such that only the administrator has appropriate read and write access to those directories.

It is possible for a site to give different UNIX permissions to a group of users. To do this, an SPD Server administrator must bring up another SPD Server using a different UNIX user account. (Bringing up a different SPD Server affects only the new SPD Server files created, not existing SPD Server files.)

ACL File Security

UNIX file security alone is not adequate for many installations. For more complex workplace environments, SPD Server provides a finer level of controls, called ACL file security. ACL file security is used by default for SPD Server LIBNAME domains. SPD Server always enforces ACL file security unless an SPD Server administrator specifies the NOACL option when bringing up a Server.

To understand ACL file security, you must know how SPD Server user IDs work. The SPD Server administrator assigns each approved SPD Server user an ID, a password, a level of data authorization, and, if desired, membership in up to five ACLGROUPS. (The SPD Server user ID 'anonymous' does not require a password.)

Once your SPD Server user ID has been created, you and the SPD Server administrator can use PROC SPDO to create ACLs that grant or deny other users access to an SPD Server table. The documentation chapter on Chapter 4, “Accessing and Creating SAS Scalable Performance Data (SPD) Server Tables,” on page 33 explains how to use the PROC SPDO operator interface to secure SPD Server resources.

Accessing SPD Server from a SAS Client

SQL Pass-Through Facility

SPD Server SQL Pass-Through processing supports an associated proxy process for each new client (via the Name Server). The proxy issues SQL Pass-Through requests. To connect to an SPD Server SQL server from a SAS session, you must submit a CONNECT statement that specifies the SASSPDS engine and SPD Server options, and then issues the SQL commands.

For example:

```
PROC SQL;
 connect to sasspds
   (dbq='mydomain'
    host='namesvrID'
    serv='5555'
    user='neraksr'
    passwd='siuya');
 select *
   from connection
   to sasspds
    (select * from employee_info);
 disconnect from sasspds;
 quit;
```

LIBNAME Access

Overview of LIBNAME Access

A logical name, or libref, is a name for the data library that you associate with an SPD Server domain during a SAS job or session. Once a libref is assigned, SPD Server allows you to read, create, or update files in the data library if you have the appropriate access to the data library.

A libref is valid only for the current SAS job or session. Librefs can be referenced repeatedly during a valid job or session. SAS does not limit the number of librefs that you can assign during a session. Once you define a libref, it is most commonly used as the first element in two-level SAS filenames: LibraryName.Tablename. The library name, or libref, identifies where the SPD Server can find or store the file.

The documentation chapter on Chapter 4, “Accessing and Creating SAS Scalable Performance Data (SPD) Server Tables,” on page 33 contains several examples that use librefs. The following example is a libref used with LIBNAME access to an SPD Server.

Example: A Libref Used with LIBNAME Access

The statement below creates the table TRAVEL and stores it in a permanent SAS library with the libref ANNUAL.

```
data annual.travel;
```

Below is a LIBNAME statement that associates a libref, the SASSPDS engine, and an SPD Server domain.

```
LIBNAME mydatalib sasspds 'mydomain'
   host='namesvrID'
   serv='5555'
   user='neraksr'
   passwd='siuya';
```

```
LIBNAME libref SASSPDS <'SAS-data-library'> <SPD Server-options>;
```

Use the following arguments:

libref
: a name that is up to eight characters long and that conforms to the rules for SAS names.

SASSPDS
: the name of the SPD Server engine.

'*SAS-data-library*'
: the logical LIBNAME domain name for an SPD Server data library on the host machine. The Name Server resolves the domain name into the physical path for the library.

SPD Server-options
: one or more SPD Server options.

LIBNAME Options

You must supply the SASSPDS engine name to access SPD Server LIBNAME domains with a LIBNAME statement. You must also specify one or more SPD Server options. The syntax for an SPD Server option is

```
<SPD Server-option>=<value>;
```

SPD Server-option
: a keyword to name the option.

value
: a value expected by the keyword.

Option values in a LIBNAME statement enable the engine to initiate, manage, and tailor a client session. This section summarizes LIBNAME options and groups them by function.

Connect to a Specified SPD Server Host

Overview of Connecting to a Specified SPD Server Host

To connect to a host, SPD Server needs the network node name for the SPD Server host machine or the IP address of the server machine, and the port number of a Name Server. SPD Server provides the following options to locate a Name Server using a named service.

SERVER=
: specifies a node name for an SPD Server host machine and a port number for the Name Server running on the machine.

HOST=
: specifies a node for an SPD Server host machine and a port number for the Name Server running on the machine.

Both options have the same function. SERVER= arguments are compatible with SAS/SHARE software. HOST= arguments support FTP conventions. The HOST option allows a node to be an IP address (for example, 123.456.76.1); the SERVER option requires a network node name.

SPDSHOST= Macro Variable

If you create a SAS macro variable named SPDSHOST= or an environment variable named SPDSHOST=, whenever a LIBNAME statement does not specify an SPD Server host machine, SPD Server will look for the value of SPDSHOST= to identify the server.

```
%let spdshost=samson;
  LIBNAME myref sasspds 'mylib'
  user='yourid'
  password='swami';
```

The first statement assigns the SPD Server host SAMSON to the macro variable SPDSHOST. Therefore, a subsequent LIBNAME statement does not need to name the host server again.

Validate the Client User ID

SPD Server uses the Name Server to secure its domains. SPD Server uses ACL file security to secures domain resources. If ACL file security is enabled, the software grants access in the following hierarchy:

- using the permissions that belong to the UNIX ID that is associated with the SPD Server
- using the permissions that belong to the SPD Server user ID

You can use SQL Pass-Through and LIBNAME options to specify the identify of an SPD Server user. SPD Server uses a special ID table to validate user IDs and passwords. The following LIBNAME options identify a client:

ACLGRP=
: Specifies one of up to five ACL groups that the user can belong to.

ACLSPECIAL=
: Grants special privileges to an SPD Server user who is previously set up as special (ACLSPECIAL=YES is defined for the user in the password file.) Special privileges override other ACL restrictions that apply to resources in the domain.

CHNGPASS=
: Prompts a client user to change his or her SPD Server password.

NEWPASSWORD= or NEWPASSWD=
: Specifies a new password for an SPD Server client user.

PASSWORD= or PASSWD=
: Specifies a password to validate an SPD Server client user.

PROMPT=
: Prompts for a password to validate an SPD Server client user.

PASSTHRU=
: Specifies implicit SQL Pass-Through options for an SPD Server client user.

USER=
: Specifies the SPD Server user ID.

Table 3.1 *User ID Options When ACL File Security Is Enabled*

User=	Password= or Prompt=	Grants Access To . . .
Required unless the SAS client process has a user ID, that is, not a Windows client. Submitted values for User= are validated against the SPD Server user ID Table.	Required and validated against the SPD Server user ID Table.	Resources that you create within the SPD Server LIBNAME domain and in other resources that are not excluded by ACLs or by UNIX file permissions.

Table 3.2 *User ID Options When UNIX File Security Only Is Enabled*

User=	Password= or Prompt=	Grants Access To . . .
Not required. The SPD Server user ID under UNIX file security only is anonymous.	Not required with anonymous user ID.	All resources within the LIBNAME domain granted by UNIX permissions for the SPD Server's UNIX ID.

Manage Server Network Traffic

If your SPD Server installation uses the same physical machine to run your SPD Server client process and your SPD Server host services, you can use the two following SPD Server options to improve client/server network traffic:

NETCOMP=
compresses the data stream in an SPD Server network packet.

UNIXDOMAIN=
uses UNIX domain sockets for data transfer between the client and the SPD Server.

Additional LIBNAME Options

BYSORT=
performs an implicit sort when a BY clause is encountered.

DISCONNECT=
specifies when to close network connections between the SAS client and the SPD Server. This can be after all librefs are cleared or at the end of a SAS session.

ENDOBS=
specifies the end row (observation) in a user-defined range.

NOSASSORT=
ignores an explicit PROC SORT statement.

STARTOBS=
specifies the start row (observation) in a user-defined range.

TRUNCWARN=
Suppresses hard failure on NLS transcoding overflow and character mapping errors. When using the TRUNCWARN=YES LIBNAME option, data integrity can be compromised because significant characters can be lost in this configuration. The default setting is NO, which causes hard Read and Write stops when transcode overflow or mapping errors are encountered. When TRUNCWARN=YES, and an overflow or

character mapping error occurs, a warning is posted to the SAS log at data set close time if overflow occurs, but the data overflow is lost.

LIBNAME Example Statements

Example 1

Example 1 creates the libref MINE, associates it with the SASSPDS engine, and specifies the SPD Server LIBNAME domain GOLDMINE. Values for the SPD Server options specify to

- locate the server machine FASTCPUS and use the default service SPDSNAME to get the port number of the Name Server
- validate the SPD Server user EXPLORER
- prompt for EXPLORER's old SPD Server password
- change the password.

```
LIBNAME mine sasspds 'goldmine'
  user='explorer'
  host='fastcpus'
  prompt=yes
  chngpass=yes;
```

Example 2

Example 2 represents the first LIBNAME statement that was made for the SPDSDATA domain. It creates the libref MYLIB, associates MYLIB with the SASSPDS engine, and specifies the SPD Server LIBNAME domain SPDSDATA. Values for the SPD Server options specify to

- locate the server machine HEFTY and use the named service SPDSNAME to get the port number of the Name Server.
- validate the SPD Server user ID **camills** and account password of **escort**.
- store data file partitions in the directories MAINDATA on device DISK1, MOREDATA on device DISK2, and MOREDATA on device DISK3. This example implies that the metadata and index partitions for tables are stored in the primary file system, that is, the path set up by the SPD Server administrator for SPDSDATA.

```
LIBNAME mylib sasspds 'spdsdata'
  server=hefty.spdsname
  user='camills' password='escort'
  datapath=('/disk1/maindata'
            '/disk2/moredata'
            '/disk3/moredata');
```

SPD Server Table Options

SPD Server table options specify processing actions that apply only to a specific table. When you use a LIBNAME statement, you should specify the options in parentheses next to the table name. If you use an SQL Pass-Through statement, use brackets to specify the options next to the table name.

Options to Enhance Performance

BYNOEQUALS=
specifies the index output order of table rows with identical values for the BY column.

NETPACKSIZE=
controls the size of an SPD Server network data packet.

SEGSIZE=
sizes the segment for index files associated with an SPD Server table.

Options for Other Functions

BYSORT=
performs an implicit sort of a given table when a BY clause is encountered and there is no index available.

ENDOBS=
specifies the end row (observation) number in a user-defined range.

STARTOBS=
specifies the start row (observation) number in a user-defined range.

SORTSIZE=
specifies the amount of memory (in number of bytes, not Kbytes or Mbytes) that SPD Server is able to allocate to complete a sorting request. The SORTSIZE= table option declared must be less than the global sortsize parameter specified in the spdsserv.parm server parameter file.

VERBOSE=
details all indexes associated with an SPD Server table. This option also provides other information, such as who is the table owner and the ACL group.

SPD Server Macro Variables

Overview of Macro Variables

You can use global macro variables in SPD Server to simplify your work. Global macro variables use default values set by the SPD Server software and operate in the background. You can make global changes to the values of macro variables in your code by specifying a new the default setting for the specified variable. The new default setting is applied to all macro variables in the code that you submit to SPD Server. You can also override the setting for a single macro variable by using a table option to change the setting for only the specified table.

The default macro variable values automate sophisticated processing decisions. The default settings furnish good performance. However, top performance often requires intelligent changes to some macro variable default settings. When you make changes to the macro variable default settings, you should attempt to find the best processing opportunity for the type of data that you have.

Learning the best way to set SPD Server macro variables and options takes time. Sometimes, performance testing is the only way to determine whether changing a setting improves processing performance. Performance testing is time well spent. After you quantify performance parameters under various macro variable settings, you can customize SPD Server so that it solves your real business or data problems with maximum efficiency.

Each SPD Server installation is different. You might want to change many values, or just a few default values. When you make changes, you will find macro variables are friendly, flexible, and easy to manipulate.

Use a %LET statement to change macro variable values. You can place the macro variable assignment anywhere in the open code of a SAS program except data lines. The most convenient place to put your %LET statements to initialize macro variables is in your **autoexec.sas** file or at the beginning of a program. The macro variable assignment is valid for the duration of your session or the executing program. Macro variable values remain in effect until they are changed by a subsequent assignment.

Assignments for macro variables with YES|NO arguments must be entered in uppercase (capitalized).

Because the SPD Server macro variables operate behind the scenes, you cannot query SPD Server to find out the status of a macro variable. SAS does not 'know' about the status of macro variables. If you want to see which SPD Server macro variables are in effect, or their default values, you can use PROC SPDO.

Macro Variables and Corresponding Table Options

When you need to apply the action to a single table that a macro variable applies globally to all tables, you should use a table option instead of the macro variable setting. A table option is more selective because you can turn the macro variable function on or off for a single table.

Summary of SPD Server Macro Variables

This section summarizes the SPD Server macro variables and groups them by the function of their default value.

Variable for a Client and Server Running on the Same UNIX Machine

SPDSCOMP=
specifies to compress the data when sending a data packet through the network.

Variable for Compatibility with the Base SAS Engine

SPDSBNEQ=
specifies the output order of table rows with identical values in the BY column.

Variables for Miscellaneous Functions

SPDSEOBS=
specifies, when processing a table, the end row (observation) number in a user-defined range.

SPDSSOBS=
specifies, when processing a table, the start row (observation) number in a user-defined range.

SPDSUSAV=
specifies, when appending to tables with unique indexes, to save rows with nonunique (rejected) keys to a separate SAS table.

SPDSUSDS=
returns the name of a hidden SAS table generated by the SPD Server which stores rows with identical (nonunique) table values.

SPDSVERB=
specifies when executing a PROC CONTENTS statement to provide more details that are specific to SPD Server indexes that are associated with the table. Examples of information include ACL information, index information, PARTSIZE= value, and others.

SPDSFSAV=
specifies to retain the table if an abnormal condition is encountered during a table-creation operation. (Normally SAS closes and deletes these tables.)

SPDSEINT=
specifies disconnect behavior for the SQL Pass-Through EXECUTE() statement.

Variables for Sorts

SPDSBSRT=
specifies for the SPD Server to perform a sort whenever it encounters a BY clause, and there is no index available.

SPDSNBIX=
specifies whether to turn BY-sorts with an index on or off.

SPDSSTAG=
specifies whether to use non-tagged or tagged sorting for PROC SORT or BY processing.

Variables for WHERE Clause Evaluations

SPDSTCNT=
specifies the number of threads to be used for WHERE clause evaluations.

SPDSEV1T=
specifies whether the data returned from WHERE clause evaluations that use an index should be in strict row (observation) order.

SPDSEV2T=
specifies whether the data returned from WHERE clause evaluations that do **not** use an index should be in strict row (observation) order.

SPDSWDEB=
specifies when evaluating a WHERE expression, whether WHINIT, the WHERE clause planner, should display a summary of the execution plan.

SPDSIRAT=
controls, when WHERE clause processing with enhanced bitmap indexes, whether to perform segment candidate pre-evaluation.

Variables That Affect Disk Space

SPDSCMPF=
: specifies to add a number of bytes to a compressed block as growth space.

SPDSDCMP=
: specifies to compress SPD Server tables on the disk.

SPDSIASY=
: specifies, when creating multiple indexes on an SPD Server table, whether to create the indexes in parallel.

SPDSSIZE=
: specifies the size of an SPD Server table partition.

Variables to Enhance Performance

SPDSNETP=
: sizes a buffer in server memory for the network data packet.

SPDSSADD=
: specifies whether to apply a single row, or multiple rows at a time, when appending to a table.

SPDSSYRD=
: specifies whether to perform data streaming when reading a table.

SPDSAUNQ=
: specifies whether to cancel an append operation if uniqueness is not maintained.

Chapter 4
Accessing and Creating SAS Scalable Performance Data (SPD) Server Tables

Introduction

This documentation chapter describes how to access SPD Server using SAS and an SPD Server SQL Pass-Through Facility or SAS LIBNAME statement. The chapter also demonstrates typical data tasks on an SPD Server host. Finally, it discusses how to secure SPD Server resources using PROC SPDO. Power users who have special privileges should see Chapter 16, "SPD Server Operator Interface Procedure (PROC SPDO)" in the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.

Note: For readability, the SPD Server SQL Pass-Through Facility is shortened here to SQL Pass-Through Facility, unless the context requires a more explicit reference. Similarly, when the chapter references a Name Server, it is the Scalable Performance Data Server Name Server.

Using a LIBNAME Statement to Access SPD Server

Overview of Using a LIBNAME Statement

It is not necessary to understand all possible LIBNAME and table options to initiate an SPD Server client session. There are only a few required elements which are shown in the example below. The LIBNAME statement should specify

- the local library reference (libref)
- the required engine name SASSPDS
- a valid domain name that is registered to the Name Server and defined to the SPD Server host
- the Name Server host's name
- the user ID
- password access, either through the PROMPT=YES switch or using the PASSWD keyword. (The PROMPT=YES approach is recommended for security reasons.)

Example: Issuing an Initial LIBNAME Statement

```
LIBNAME market sasspds 'mktdata'
 host='sunone'
 user='user id'
 prompt=yes;
```

This example specifies the libref market, the engine name SASSPDS, the LIBNAME domain mktdata, and the Name Server host called **sunone**. It identifies an SPD Server user ID and is configured to prompt the user for a password. Alternately, but less recommended, is

```
LIBNAME market sasspds 'mktdata'
 host='sunone'
 user='user id'
 passwd='beemer';
```

The only difference between this and the previous example is the password specification. Here the password **beemer** is recursed into the LIBNAME statement. This method can be used for batched SPD Server jobs that run unattended.

The Client Session

Successfully issuing the LIBNAME statement or SQL Pass-Through statement(s) initiates an SPD Server client session. The client session operates using a combination of up to four distinct components:

SPD Server Name Server
: The Name Server acts as a traffic cop and provides a central point of control between clients and SPD Server hosts. The Name Server maintains a list of LIBNAME domains associated with each SPD Server host. Client sessions will always connect to an SPD Server host through a Name Server. The Name Server resolves the submitted

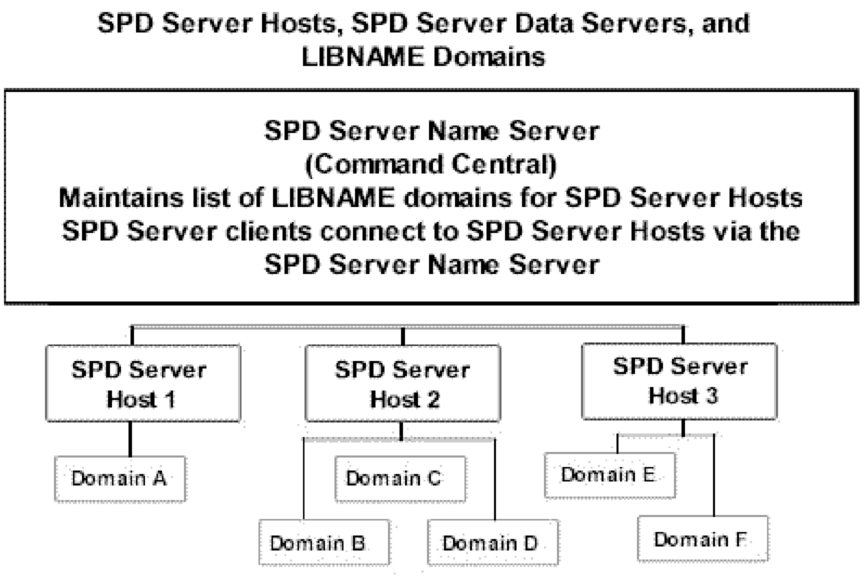
LIBNAME domain name (a logical entity) to a physical path (usually a UNIX or Windows directory). The Name Server then connects you to the SPD Server serving the domain without requiring you to know physical addresses. An SPD Server administrator sets up the LIBNAME domains in a parameter file for SPD Server which then registers its domains with the Name Server.

SPD Server Host
: Each SPD Server host controls security access to the domain resources it manages. When an SPD Server host starts up, it registers its LIBNAME domains with the Name Server. Clients can connect to an SPD Server host only through a Name Server -- direct connections between clients and SPD Server hosts are not permitted. The SPD Server host validates the client user ID and password (passed in the LIBNAME statement), launches the system process (client proxy) for each client, and grants access to the appropriate SPD Server domain.

SQL Server
: The SQL server parses and processes the Pass-Through SQL syntax submitted by the SAS client.

SPDSSNET Server
: The SPDSSNET server enables access between clients without SAS software and SAS Scalable Performance Data Server. The SPDSSNET server runs as a stand-alone process on either the client or SPD Server host machine. It acts as a bridge between the SAS ODBC driver and the SPD Server host. SPDSSNET can also be used with JDBC drivers and HTMSQL used with Web Servers. SPDSSNET can run multiple processes concurrently and perform parallel processing.

Figure 4.1 *SPD Server Hosts, SPD Server Name Servers, and LIBNAME Domains*


SPD Server Hosts, SPD Server Data Servers, and LIBNAME Domains
SPD Server Name Server (Command Central) Maintains list of LIBNAME domains for SPD Server Hosts SPD Server clients connect to SPD Server Hosts via the SPD Server Name Server
SPD Server Host 1
SPD Server Host 2
SPD Server Host 3
Domain A
Domain C
Domain E
Domain B
Domain D
Domain F

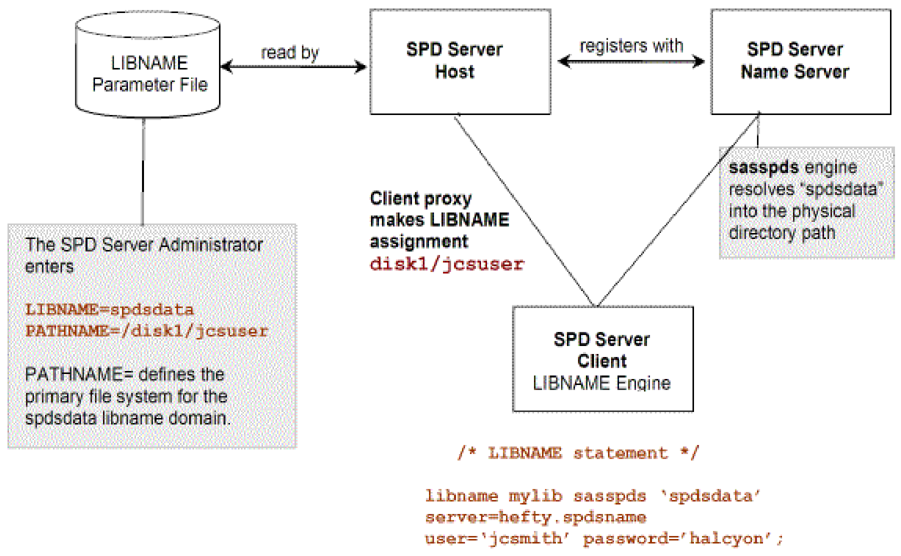
Managing Large SPD Server Files

Leaving aside performance issues, managing large files is a matter of file storage and disk space. Optimally, an SPD Server administrator will manage storage space for SPD Server LIBNAME domains. In this case, you do not need to consider storage issues -- SPD Server does the work for you. The Help section on Chapter 11, "Optimizing SPD Server Performance ," on page 173 contains more detail on managing large SPD Server files.

Initial Setup of SPD Server LIBNAME Domain Storage

Figure 4.2 reviews how an SPD Server domain is set up. An SPD Server administrator must define the name and primary path for the domain in the LIBNAME parameter file for SPD Server. The path that the administrator defines for each domain is referred to as the primary file system for that domain. The LIBNAME parameter file is read by the SPD Server at startup. The SPD Server registers the domains with the SPD Name Server. When the user issues a LIBNAME statement, the client sends a message to the SPD Name Server that will resolve the domain name to its physical directory path and also determine the SPD Server that registered the domain.

Figure 4.2 *SPD Server LIBNAME Domains*


LIBNAME Parameter File
read by
SPD Server Host
registers with
SPD Server Name Server
sasspds engine resolves "spdsdata" into the physical directory path
Client proxy makes LIBNAME assignment disk1/jcsuser
The SPD Server Administrator enters
LIBNAME=spdsdata PATHNAME=/disk1/jcsuser
PATHNAME= defines the primary file system for the spdsdata libname domain.
SPD Server Client LIBNAME Engine
/* LIBNAME statement */
libname mylib sasspds 'spdsdata' server=hefty.spdsname user='jcsmith' password='halcyon';

Chapter 2, "SAS Scalable Performance Data (SPD) Server Overview," on page 9 discusses LIBNAME path options that allow a user to specify additional storage devices and paths for a domain. To manage their own disk space, a user must be aware of the DATAPATH=, METAPATH=, and INDEXPATH= options, as well as the ROPTIONS= option that the SPD Server administrator uses.

Effect of the Administrator Option, ROPTIONS=

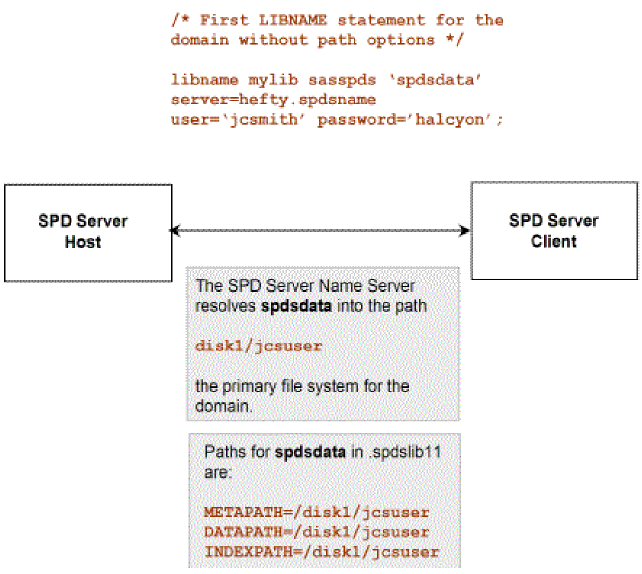
After defining a primary file system for a domain, an SPD Server administrator can use LIBNAME parameter file options, identical to the DATAPATH=, METAPATH=, and INDEXPATH= options in the LIBNAME statement, to set up additional paths for the domain. However, the administrator can also exercise an option to restrict a user from defining additional paths via the LIBNAME statement with the ROPTIONS= LIBNAME parameter file option. When an SPD Server administrator uses the ROPTIONS= option, the administrator's specification takes precedence over the users. More information is available in the Help section on Configuring LIBNAME Domain Disk Space in the SPD Server Administrator's Guide.

For example, assume that a user uses the DATAPATH= option to specify a path or paths to store table data for a domain, and that the SPD Server administrator also uses the

DATAPATH= option, along with ROPTIONS= for that domain entry in the LIBNAMES parameter file. The user's DATAPATH= specifications are then ignored.

The administrator's use of ROPTIONS= with path options is recommended. It relieves users of the complicated task of managing disk space and avoids the need to embed physical path information in SAS programs. Instead, SAS jobs need to refer to only the logical LIBNAME, relying on ROPTIONS= embedded by the administrator to specify all of the physical information. This approach uses the power of the Name Server, allowing it to resolve path information for an SPD Server domain.

Figure 4.3 *Primary File System Default Paths*



```
/* First LIBNAME statement for the
domain without path options */

libname mylib sasspds 'spdsdata'
server=hefty.spdsname
user='jcsmith' password='halcyon';
```

The SPD Server Name Server resolves **spdsdata** into the path

`disk1/jcsuser`

the primary file system for the domain.

Paths for **spdsdata** in .spdslib11 are:

```
METAPATH=/disk1/jcsuser
DATAPATH=/disk1/jcsuser
INDEXPATH=/disk1/jcsuser
```

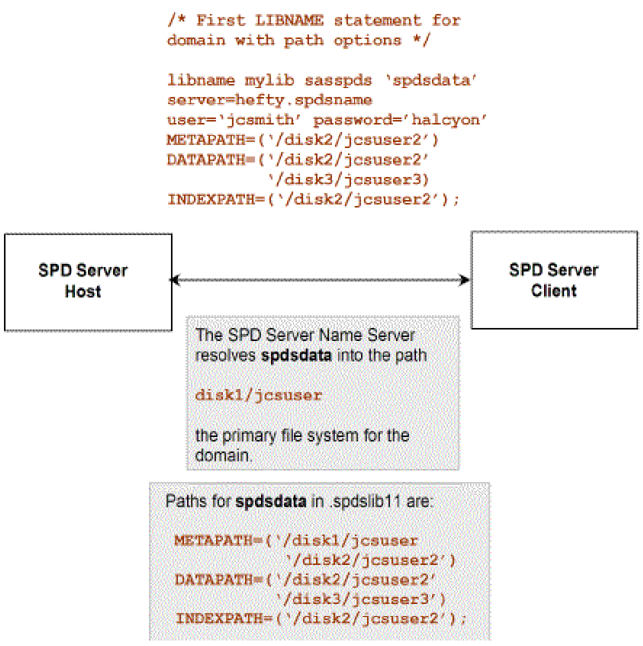
Explicit or Default Storage Paths

You might wonder why the software offers you path options and then discourages their use? The answer is flexibility. A site can elect to allow users to manage their own disk space. While this practice is not recommended, the software allows for the possibility.

To use path options effectively, you must know that the first LIBNAME assignment or SQL Pass-Through CONNECT statement naming a domain establishes an initial set of paths for the domain. You can specify the paths, or the software can establish a default set. Figure 4.3 shows a default set of paths. Figure 4.4 shows an explicit initial set of paths.

The path options METAPATH=, DATAPATH=, and INDEXPATH= store partitions for the component files: metadata, data, and indexes. Subsequent LIBNAME assignments augment the path list created by the initial LIBNAME assignment. That is, SPD Server appends each new path assignment to any prior list for the component.

Figure 4.4 *Explicit Initial Set of Paths*



```
/* First LIBNAME statement for
domain with path options */

libname mylib sasspds ‘spdsdata’
server=hefty.spdsname
user=‘jcsmith’ password=‘halcyon’
METAPATH=(‘/disk2/jcsuser2’)
DATAPATH=(‘/disk2/jcsuser2’
          ‘/disk3/jcsuser3)
INDEXPATH=(‘/disk2/jcsuser2’);
```

The SPD Server Name Server resolves **spdsdata** into the path

```
disk1/jcsuser
```

the primary file system for the domain.

Paths for **spdsdata** in .spdslib11 are:

```
METAPATH=(‘/disk1/jcsuser
           ‘/disk2/jcsuser2’)
DATAPATH=(‘/disk2/jcsuser2’
          ‘/disk3/jcsuser3’)
INDEXPATH=(‘/disk2/jcsuser2’);
```

In summary, unless you or an SPD Server administrator specify an initial set of paths, the software uses the domain's primary file system in the LIBNAMES parameter file for the default path set. As you will learn in the next section, the default path set might not be ample for large tables or provide optimal performance.

Understanding SPD Server Component Storage

Earlier, you learned that the software creates a list of paths for storage of table files in an SPD Server domain, but file partition storage was not discussed. This section focuses on using path options when an SPD Server administrator has not used the ROPTIONS= option.

Minimally, each table consists of a metadata component and a data component. Each component file consists of one or more partition files on disk. The software requires that the first metadata partition reside in the primary file system, that is, the path defined for the domain by an SPD Server administrator. Other metadata partitions can overflow to additional paths specified using the METAPATH= option.

If no paths are specified for index and data components by the INDEXPATH= or DATAPATH= options, the software stores these partitions in the primary file system too. If other paths are specified, the software stores the initial partition for these classes in the first path with available space. (Unlike metadata partitions, data, and index partitions do not have to start in the primary file system.) A partition can expand until the path fills up; remaining partitions then overflow to the next path with available space, and so on. (See Figure 4.5 on page 39.)

Forced Partitioning of the Data Component

To improve parallel processing of various operations involving full-table scans (for example, WHERE clause evaluations without indexes or SQL GROUP BY evaluations) the SPD Server allows you to force creation of data component partitions at fixed-size intervals. To specify the size interval, use the PARTSIZE= table option. By default, the SPD Server sets PARTSIZE= to 16 megabytes. See“SPD Server Table Options ” on page 28 for details.

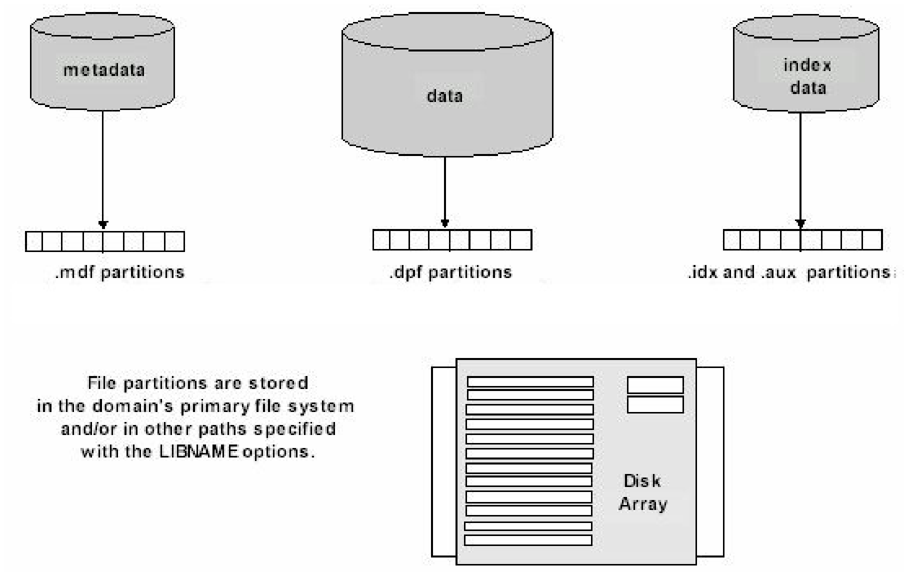
The SPD Server uses the collection of file systems that you specify with the DATAPATH= option to distribute partitions in a cyclic (round-robin) fashion. But, instead of creating partitions until the first file system is full, the SPD Server randomly chooses a file system from the DATAPATH= list for the first partition, and then sequentially assigns partitions to successive file systems in the DATAPATH= list. The software continues to cycle through the file system set, as many times as needed, until all data partitions for the table are stored. Assume that you specify

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

Subsequently, you store your BIGONE table into the domain. SPD uses random placement of data partitions in the DATAPATH= list, so the first BIGONE partition can be stored in either the **`/data1`** or the **`/data2`** directory. Subsequent partitions will alternate between the **`/data1`** and **`/data2`** directories, and so on.

If you set PARTSIZE=0, SPD Server uses the DATAPATH= file systems strictly as overflow space. That is, it creates partitions in the first file system, up to the file size limit of your operating system. Then, when the first file system is full, it proceeds to the second file system, and so on.

Figure 4.5 *SPD Server Component Storage*


metadata
data
index data
.mdf partitions
.dpf partitions
.idx and .aux partitions
File partitions are stored in the domain's primary file system and/or in other paths specified with the LIBNAME options.
Disk Array

What happens when you issue the first LIBNAME statement for a domain but do not specify path options? If your tables are small, most likely the primary file system is probably adequate. However, if you store large tables, the primary file system can fill up quickly. How do you know when the primary file system is full? SPD Server will return an error message when you perform an append operation on an existing table or create a new table in the domain.

Importance of the First Metadata Partition

If the primary file system is full, you can issue a subsequent LIBNAME statement specifying additional paths. This allows a data append to an existing table but might not allow creation of a new table in the domain. The reason why the new paths did not solve the create failure might not be obvious. The answer is the software cannot store the first metadata file partition because the primary file system is still full. What is the create failure solution? Either free space in the primary file system or have the SPD Server administrator create a new LIBNAME domain.

Using Path Options for Large Table Storage

Overview of Using Path Options

If you must manage your table storage, anticipate disk space for large tables. Use the LIBNAME path options with the first LIBNAME statement for the domain. Store data and index partitions using the DATAPATH= and INDEXPATH= options on a different storage device than the primary file system. This reserves the primary file system for metadata files.

Example 1: Specify Explicit Initial Set of Paths

SITEUSR1 issues the first LIBNAME statement for the MYLIB domain. By default, the domain's primary file system is used to store metadata partitions but another device MYDISK30 and directory SITEUSER is specified to store the data and index partitions. (The SPD Server administrator created the primary file system for MYLIB.)

```
/* I anticipate the primary file system for the MYLIB domain    */
/* is ample for metadata files, but I will use MYDISK30         */
/* to store my data and index partitions.                       */
   LIBNAME myref sasspds 'mylib'
      datapath=('/mydisk30/siteuser')
      indexpath=('/mydisk30/siteuser')
      server=husky.spdsname
      user='siteusr1' prompt=yes;
```

Example 2: Specify Subsequent LIBNAME Statement to Add Paths

SITEUSR1 issues a subsequent LIBNAME statement for the MYLIB domain specifying additional paths for the data and index partitions. The user is storing very large tables so two storage devices (and directories) for data are listed, and a third device for indexes associated with the tables is listed.

```
/* I noticed today MYDISK30 is getting full.                    */
/* I am adding MYDISK31 for possible overflows.                 */
   LIBNAME expand sasspds 'mylib'
      datapath=('/mydisk31/siteuser' '/mydisk32/siteuser')
      indexpath=('/mydisk33/siteuser')
      server=husky.spdsname
      user='siteusr1' prompt=yes;
```

The software appends the new paths listed to the prior list for each component type. The entire path list that **.spdslib11** maintains now is

```
datapath=('mydisk30/siteuser'
          '/mydisk31/siteuser'
          '/mydisk32/siteuser')
indexpath=('mydisk30/siteuser'
           '/mydisk33/siteuser')
```

How does SPD Server use the path list? It stores partitions of the data components for MYLIB tables in the specified data paths. (How the software uses the paths depends upon the value of the PARTSIZE= option.) For index components, it stores the files in the first path listed until the space is filled, then it proceeds to fill the next path listed.

Migrating Tables between SAS and SPD Server

Many organizations use SPD Server because they need more horsepower to handle very large SAS tables. As a result, there are many instances where it is handy to be able to migrate data in both directions between SAS and SPD Server. SPD Server provides simple methods to easily migrate data between SAS and SPD Server.

SAS and SPD Server Table Migration Examples

Example 1: Create a SAS Table from an SPD Server Table

To create a SAS table from an SPD Server table, issue a LIBNAME statement but do not specify the engine SASSPDS. Your program will then create a Base SAS table. (Later, if you decide to use SPD Server capabilities, you can convert the SAS table to the SPD Server format. Conversion is easy: interchange table formats using the SAS System's COPY procedure. See Example 2.)

```
/* Create local racquets data set. */
   LIBNAME local '/u/sasdemo/local';

   data local.racquets;
      input racquet_name $20. @22 weight_oz @28 balance $2.
         @32 flex @36 gripsize
         @42 string_type $3. @47 retail_price @55 inventory_onhand;
      datalines;
   Filbert VolleyMaster 10.5  HL   5  4.5   syn  129.95   5
   Solo Queensize       10.9  HH   6  5.0   syn  130.00   3
   Perkinson AllCourt   11.0   N   5  4.25  syn  159.99  12
   Wilco Specialist      8.9  HL   3  5.0   nat  287.50   1
   ;
```

Example 2: Convert from SAS to SPD Server Format

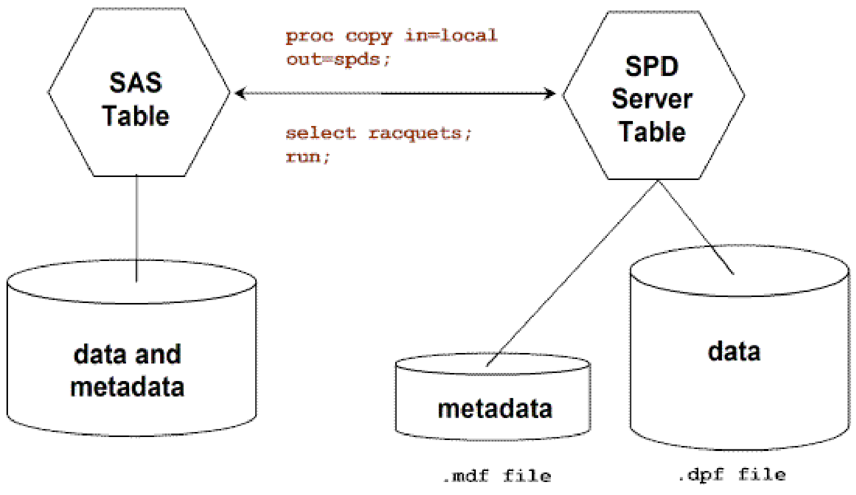
SITEUSR1 makes a libref SPORT, associates SPORT with the SPD Server engine SASSPDS, and points to the CONVERSION_AREA domain on an SPD Server host server named HUSKY. User SITEUSR1 uses a default named service SPDSNAME to locate the port number of the Name Server and requests a prompt for the password.

The PROC COPY statement inputs the SAS table LOCAL.RACQUETS and outputs the SPD Server table SPORT.RACQUETS to the CONVERSION_AREA domain. After the PROC COPY statement executes, the SAS table becomes two SPD Server table component files. (See Figure 4.6.)

```
/* Copy existing SAS table to the SPD Server format. */
   LIBNAME sport sasspds 'conversion_area'
   server=husky.spdsname
   user='siteusr1'
   prompt=yes;
```

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

Figure 4.6 PROC COPY Converts a SAS Table to an SPD Server Table



The SQL Pass-Through Facility

Overview of the SQL Pass-Through Facility

SPD Server uses Pass-Through SQL commands to access and manipulate data. What does this mean? Enabling Pass-Through SQL functionality provides SPD Server clients with a new way to establish a connection with an SPD Server host or direct load from an external database such as Oracle. Users now have broader data access in the SPD Server environment and growing connectivity to external databases using the SPD Server engine.

The Chapter 9, “SPD Server SQL Syntax Reference Guide,” on page 151 documentation chapter provides additional detailed reference information about using SPD Server SQL syntax.

Accessing Data Using the SQL Pass-Through Facility

The SQL Pass-Through Facility is another access method allowing SPD Server to connect to a SQL server and manipulate data. An overview of the steps is presented here, and followed with examples. These are the major steps for using SQL Pass-Through:

1. Establish a connection from an SPD Server client using a CONNECT statement.
2. Send SPD Server SQL statements using the EXECUTE statement.
3. Retrieve data SQL query with the CONNECTION TO component in a SELECT statement's FROM clause.
4. Terminate the connection using the DISCONNECT statement.

SQL Pass-Through Statements

CONNECT Statement

Specifies the SAS I/O engine that will provide the SQL Pass-Through access.

Syntax

CONNECT TO *dbms-name* < AS *alias* >(*dbms-args*);

Arguments:

dbms-name (required)

Specifies the name of the engine.

When running SAS and PROC SQL, you must specify **sasspds** to obtain SQL Pass-Through **to** an SPD Server SQL server. You must specify **spdseng** to obtain SQL Pass-Through **from** an SPD Server SQL server. The later examples show CONNECT statements specifying these engines.

AS alias (optional)

Specifies an alias or logical name for a connection. When specifying an alias to identify the connection, use a string without quotes. Then refer to this logical name in subsequent SQL Pass-Through statements.

Note: The alias must specify the connection that will execute the statement.

Example - Using an Alias

```
execute(...) by alias
```

or

```
select * from connection to alias(...)
```

dbms-args (required and optional arguments)

Identifies the SQL server and data source. The following dbms-args arguments are for the SPD Server engines, **sasspds**, and **spdseng**. SPD Server SQL uses the following simple syntax: Keyword=*Value*

DBQ=*libname-domain* (required)

Specifies the primary SPD Server LIBNAME domain for the SQL Pass-Through connection. The name that you specify is identical to the LIBNAME domain name that you used when making a SAS LIBNAME assignment to **sasspds**. Use single or double quotes around the specified value.

HOST=*name-server-host* (optional)

Specifies a node name or IP address for a Name Server that is currently running. Use single or double quotes around the specified string. If you do not specify a name, the software uses the current value of the SAS macro variable **spdshost** to determine the node name.

SERVICE=*name-server-port* (optional)
SERV=*name-server-port* (optional)

Specifies the network address (port number) for a Name Server that is currently running. Use single or double quotes around the specified value. If you do not specify a port number for the Name Server, the software determines the port address from the named service **spdsname** in the `/etc/services` file.

USER=*SPD Server user ID* (required on Windows, but not UNIX)

Specifies an SPD Server user ID to access an SPD Server SQL Server. Use single or double quotes around the specified value.

PASSWORD=*password* (required)
PASSWD=*password* (required, or use PROMPT=YES, unless USER='anonymou')
: Specifies an SPD Server user ID password to access an SPD Server. (This value is case sensitive.) Normally you would not specify a password in text files that others can view. More likely you would use this argument in batch jobs that are protected by file system permissions, prohibiting others from reading the job files.

PROMPT=YES (required, or use PASSWD or PASSWORD=, unless USER='anonymou')
: Specifies a password prompt to access an SPD Server SQL server. This value is case sensitive.

DISCONNECT Statement

Disconnects you from your DBMS source.

Syntax
: DISCONNECT FROM [*dbms-name* | *alias*];

Description

When you are finished with a PROC SQL connection, you must disconnect from the DBMS source. This automatically occurs when you exit the PROC SQL procedure. You can, however, explicitly disconnect from the DBMS by using the DISCONNECT statement.

Arguments

dbms-name
: the name specified in the CONNECT statement that established the connection.

alias
: the alias value specified in the CONNECT statement that established the connection.

EXECUTE Statement

The EXECUTE statement is part of the Pass-Through SQL facility. It allows the user to use specific SQL statements during a Pass-Through connection. Before using the EXECUTE statement, the user must first establish a connection using the CONNECT statement. After a user has created a Pass-Through connection, use EXECUTE to submit valid SQL statements (except the SELECT statement).

Syntax
: EXECUTE (*SQL-statement*)BY [*dbms-name* | *alias*];

Arguments

(*SQL-statement*)
: A valid SQL statement passed for execution (except SELECT statements). This argument is required and must be enclosed within parentheses.

dbms-name (required, or use *alias*)
: Identifies the DBMS to which you want to direct the SQL statement. Note that *dbms-name* must be preceded by the keyword BY.

alias (optional, or use *dbms-name*)
: Specifies an optional alias used in the CONNECT statement.

CONNECTION TO Statement

CONNECTION TO is an SQL Pass-Through component that can be used in a SELECT statement's FROM clause as part of the *from-* list. The CONNECTION TO component enables you to make Pass-Through queries for data and to use that data in a PROC SQL query or table. PROC SQL treats the results of the query like a virtual table.

Syntax

CONNECTION TO *dbms-name*(*SQL-query*)

Arguments

dbms-name (required)

If you have a single connection, *dbms-name* is the *dbms-name* specified in your CONNECT statement. If you have multiple connections, use the alias specified in the AS clause of the CONNECT statement.

(*SQL-query*)

The (*SQL-query*) specifies the SQL query you want to send. Your SQL query cannot contain a semicolon because that represents the end of a statement to SPD Server. Character literals are limited to 32,000 characters. Be sure your SQL query is enclosed in parentheses.

alias (optional)

Specifies an optional alias used in the CONNECT statement.

Example 1: Using SAS PROC SQL to Connect to a SQL Server

To connect from a SAS session to a SQL server, in this example the SPD Server's SQL Server, execute a CONNECT statement. After making the connection, the first execute statement creates a table EMPLOYEE_INFO with three columns, EMPLOYEE_NO, EMPLOYEE_NAME, and ANNUAL_SALARY. The second execute statement inserts an observation into the table where EMPLOYEE_NO equals **1** and EMPLOYEE_NAME equals **The Prez**.

The subsequent FROM CONNECTION TO statement retrieves all the records from the new EMPLOYEE_INFO table. (In this example, that would be the single observation inserted by the second execute statement.) The DISCONNECT statement terminates the data source connection.

```
PROC SQL;
connect to sasspds
 (dbq='mydomain'
  host='workstation1'
  serv='spdsname'
  user='me'
  passwd='noway');

execute (create table employee_info
  (employee_no num, employee_name char(30),
  annual_salary num) by sasspds;

execute (insert into employee_info
  values (1, 'The Prez')) by sasspds;

select * from connection to sasspds
  (select * from employee_info);
disconnect from sasspds;
quit;
```

Example 2: Nested SQL Pass-Through

SPD Server Pass-Through access can be nested. Nesting allows access to data stored on two different networks or network nodes.

In the example that follows, we nest SQL Pass-Through from the current local network host DATAGATE to access the EMPLOYEE_INFO table, which is available at the PROD host on a remote network. (Our example presumes that we have user access to PROD.)

```
proc sql;
connect to sasspds (dbq='domain1'
                    host='datagate' serv='spdsname'
                    user='usr1' passwd='usr1_pw');
execute (connect to spdseng (dbq='domain2'
                    host='prod' serv='spdsname'
                    user='usr2' passwd='usr2_pw') by sasspds;
select * from connection to sasspds(
            select * from connection to spdseng(
                select employee_no, annual_salary
                from employee_info));
execute (disconnect from spdseng) by sasspds;
disconnect from sasspds;
quit;
```

Creating a New Table

One of the SPD Server's strengths lies in the ability to create, manipulate, and query very large tables. As a rule of thumb, client users generally choose not to store massive tables locally because of their sheer size. The following code examples assume that users will create and store large tables on the SPD Server host.

Example - Creating a New Table Using Pass-Through Statements

First, connect from a SAS session to a SQL server, in this example the SPD Server's SQL Server. Then, execute a CONNECT statement. After you establish the connection, the first EXECUTE statement creates a table LOTTERYWIN with two columns, TICKETNO and WINNAME. The second EXECUTE statement inserts an observation into the table where TICKETNO equals 1 and NAME equals Wishu Weremee.

The subsequent FROM CONNECTION TO statement retrieves all the records from the new LOTTERYWIN table. (In this example, that would be the single observation inserted by the second EXECUTE statement. The DISCONNECT statement terminates the data source connection.

```
proc sql;
connect to sasspds (dbq='mydomain'
                    host='workstation1' serv='spdsname'
                    user='me' passwd='luckyones');
execute (create table lotterywin
    (ticketno num, winname char(30))) by sasspds;
execute (insert into lotterywin
    values (1, 'Wishu Weremee')) by sasspds;
select * from connection to sasspds
    (select * from employee);
disconnect from sasspds;
```

```
quit;
```

Example - Creating a New Table with a LIBNAME Statement

SITEUSR1 creates a new SPD Server table CARDATA.OLD_AUTOS on the server.

```
LIBNAME cardata sasspds 'conversion_area' server=husky.5105
     user='siteusr1' prompt=yes;

/* Create the table CARDATA.OLD_AUTOS on the SPD Server host. */

data cardata.old_autos;
     input year $4. @6 manufacturer $12. model $12. body_style $5.
     engine_liters @39 transmission_type $1. @41 exterior_color
     $10. options $10. mileage conditon;

datalines;

1966 Ford        Mustang      conv  3.5  M  white     00000001 143000 2
1967 Chevrolet   Corvair      sedan 2.2  M  burgundy  00000001  70000 3
1975 Volkswagen  Beetle       2door 1.8  M  yellow    00000010  80000 4
1987 BMW         325is        2door 2.5  A  black     11000010 110000 3
1962 Nash        Metropolitan conv  1.3  M  red       00000111 125000 3
;
```

Chapter 5

Indexing, Sorting, and Manipulating SAS Scalable Performance Data (SPD) Server Tables

Introduction

This chapter describes and provides examples on indexing, sorting, and manipulating SPD Server tables on an SPD Server host.

Indexing a Table

SPD Server provides a single SPD Server index type that efficiently indexes tables of varying size and data distributions. The SPD Server SPD index optimally supports queries that require global table views (such as queries that contain BY clause processing and SQL joins), or queries that require segmented views (such as parallel processing of WHERE clause statements).

The SPD Server Index

The SPD Server index maintains two views of the index values, a global view and a segmented view. The global view is maintained using a unique global B-tree that has a single entry for each discrete value. The segmented view is maintained by the data for each value in the global B-tree, which includes a list of segments that contain the value, and for each segment a bitmap that identifies which rows in the segment contain the value. The global view is maintained in the SPD Server index**.hbx** file, and the segmented data is maintained in the SPD Server index**.idx** file.

For queries that require a global view, SPD Server searches the hybrid global B-tree for a particular value. The segment lists are scanned for the value, then the bitmaps from each segment containing the value are read. SPD Server uses the bitmap to locate and retrieve the observations for that segment. This type of query returns results sorted first by value and then by observation number. This sorting is optimal for BY Clause processing and SQL joins.

A parallel WHERE clause on a table that is indexed is done in two phases. The first phase, pre-evaluation, uses the SPD Server indexes to build a list of segments that satisfy the query. The list drops segments from the WHERE clause scan queue when those segments contain no data in the clause range. As more and more segments are excluded from the scan queue, the benefit of the pre-evaluation phase increases proportionally. The second phase in the evaluation launches threads that read an index in parallel. Each thread queries a particular segment of the index, using information from the pre-evaluation phase. Using the SPD Server index, the thread reads the segment bitmap. The per-segment bitmaps identify the segment rows which satisfy the query for that particular column. If you include more than one indexed column in the WHERE clause, SPD Server retrieves the per-segment bitmaps for each column in parallel (as are the segments for each column). After retrieving all the bitmaps for each column of the segment, SPD Server determines which rows satisfy the query, and returns those segment rows to the client. The multi-threaded per-segment queries begin execution at the same time, and their finishing order varies and cannot be reasonably predicted. As a result, the overall order of the results cannot be guaranteed when you are using this type of query. See the documentation chapter on “WHERE Clause Planner” on page 187 for a more detailed description on using indexed columns with WHERE clause evaluations.

When a table is modified due an append or update, all SPD Server indexes on the table are updated. Updating the index can potentially fragment the per-value segment lists or cause some disk space to be wasted. A highly fragmented SPD Server index can negatively impact the performance of queries that use the index. In this case, you should reorganize the index to eliminate the fragmentation and reclaim wasted disk space, using the **ixutil** utility program. For further information about SPD Server index utilities, see Chapter 15, "Managing SAS Scalable Performance Data (SPD) Server Passwords, Users, and Table ACLs," of the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.

Creating SPD Server Indexes Examples

This section shows how to create SPD Server indexes for new and existing tables.

Creating SPD Server Indexes from a DATA Step

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

The code above creates SPD Server table X. Next, the code creates a simple SPD Server index X on column X, and a composite SPD Server index Y on columns (A B).

Creating SPD Server Indexes from PROC DATASETS

```
PROC DATASETS lib=foo;
  modify x;
  index create x;
  index create y=(a b);
quit;
```

This creates the same simple and composite SPD Server indexes that were created in Example 1, assuming that the same DATA step was executed without index creation included.

Creating SPD Server Indexes Using SQL

```
PROC SQL;
create index x
    on foo.x (x);
create index y
    on foo.x (a,b);
quit;
```

This creates the same simple and composite SPD Server indexes as in Example 1, assuming that the same DATA step was executed without index creation included.

Creating SPD Server Indexes Using Pass-Through SQL

```
PROC SQL;
connect to sasspds (
   dbq="path1"
   server=host.port
   user='anonymous');

  execute(create index x on x (x))
  by sasspds;

  execute(create index y on x (a,b))
  by sasspds;
quit;
```

This creates the same simple and composite SPD Server indexes as in Example 1, assuming that the same DATA step was executed without index creation included.

Using VERBOSE= to See Index Information

There will be times when you want to see information about indexes that are associated with a particular table. The table option VERBOSE= provides details of all indexes associated with an SPD Server table. For example, if the code from Example 2 above is followed with the expression below:

```
PROC CONTENTS
  data=sports.expraqs
```

```
      (verbose=yes);
    run;
```

The following will be output:

```
       Alphabetic List of Index Info:
Bitmap Index (No Global Index):         GRIPSIZE
KeyValue (Min):                 4.250000
KeyValue (Max):                 5.000000
# of Discrete values:           3
```

Using PROC SORT with SPD Server

If you use PROC SORT with SPD Server, your table will be sorted. However, you might want to understand a few sort details to avoid surprises. For example, assume that you submit a PROC SORT statement in order to sort a table that was not previously indexed, or sorted on the table's BY column.

PROC SORT takes advantage of SPD Server sorting implicitly and asserts BY Clause ordering to the SPD Server. This performs the sort on the SPD Server machine, but there will still be significant I/O between the client node and the SPD Server machine. The sorted data still makes a round trip from the server machine to the client machine and back again. Fortunately, the SQL Pass-Through Facility in SPD Server offers an extension to the SQL language to permit a table copy and sort operation, all on the server machine.

Knowing the implications of using PROC SORT with SPD Server, how can you avoid inefficiency? The answer is to eliminate PROC SORT statements from your SAS jobs where possible. Instead, make SAS procedures and DATA steps that require BY Clause processing use SPD Server's implicit sorts.

Example Using Implicit SPD Server BY Clause Sort

```
/* The following DATA step performs a server sort on the  */
/* table column PRICE. There is no prior index for PRICE  */

   data _null_;
   set sport.expraqs;
     by price;
        if (string='nat') then do;
        put '*' @@;
        price = price - 30.00;
     end;
   put raqname @30 price;
```

Example Using PROC SORT

```
/* The following PROC SORT performs a server sort on the */
/* table column MODEL. There is no prior index for MODEL */

  PROC SORT
    data=inventory.old_autos
    out=inventory.old_autos_by_model;
    by model;
  run;
```

Chapter 6

Using SAS Scalable Performance Data (SPD) Server with Other Clients

Overview

This chapter describes using SAS Scalable Performance Data (SPD) Server to connect with ODBC, JDBC, htmSQL, and SQL C API clients.

Scalable Performance Data Server provides ODBC, JDBC, htmSQL, and SQL C API access to SPD Server data stores from all supported platforms.

SPD Server can read tables exported from Base SAS software using PROC COPY, and, with the proper drivers installed on the network, allows queries on the tables from client machines that do not have SAS software.

There are four possible options:

- **ODBC:** Open Database Connectivity - This is an interface standard that provides a common interface for accessing databases. Many software applications running in a Windows environment are compliant with this standard and can access data created by other software. This is a good choice if you have client machines running Windows applications, such as Microsoft Excel or Microsoft Access.
- **JDBC:** Java Database Connectivity - This option allows users with browsers to log on to a Web page and make a query. The results of the request are formatted and returned to a Web page. This makes information available across a wide range of client platforms because all you need, after installing the JDBC Driver on SPD Server, is a Web page with some Java code, and a client machine with a browser enabled by Java.
- **htmSQL:** HyperText Markup Structured Query Language - This option allows users with browsers to log on to a Web page and make a query. The results of the request are formatted and returned to a Web page. This makes information available across a wide range of client platforms. Why? After installing the htmSQL driver in SPD Server, all you need is an htmSQL Web page and a client machine with a browser.
- **SQL C API:** This option allows access to SPD Server tables from SQL statements generated by C/C++ language applications. This access is provided in the form of a C-language run-time access library. This library provides a set of functions that you can use to write custom applications to process SPD Server tables and to generate new ones. This library is designed to support multi-threaded applications and is available on all supported SPD Server platforms.

Note: GUI interfaces might not display all return codes or error messages that the server generates.

Using Open Database Connectivity (ODBC) to Access SPD Server Tables

Read this section if you do not have Base SAS software on the network client, but you want to access SPD Server tables on the network using an ODBC-compliant program, such as Microsoft Word, Query, Excel, or Access, and you have SPD Server tables available for use somewhere on the network, or SPD Server data servers and SPD Server SNET Servers running or client machines in a Windows environment.

Why Use ODBC?

You have SPD Server tables available on your network, and one or more of the following might be true:

- You do not have Base SAS software running on the Windows client, but you need to view or change SPD Server tables.
- You need to view or change the SPD Server tables using a Microsoft spreadsheet, database, or word processor.
- You need to view or change SPD Server tables in ways that cannot be predetermined or programmed into a Web page.

- You need to view or change SPD Server tables using Windows tools you are familiar with.

Installing OBDC Drivers on the Server

Instructions for installing the OBDC driver are included in the SPD Server installation package.

Configuring ODBC on the Client

1. Configure an ODBC data source.
2. Make your query using a Windows program.

Figure 6.1 *Configure ODBC to Connect SPD Server Client to SPD Server Host*

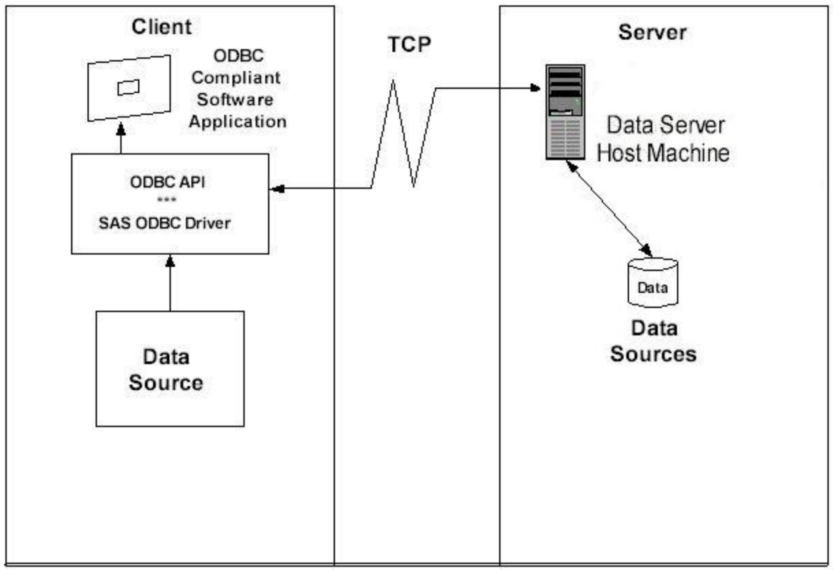
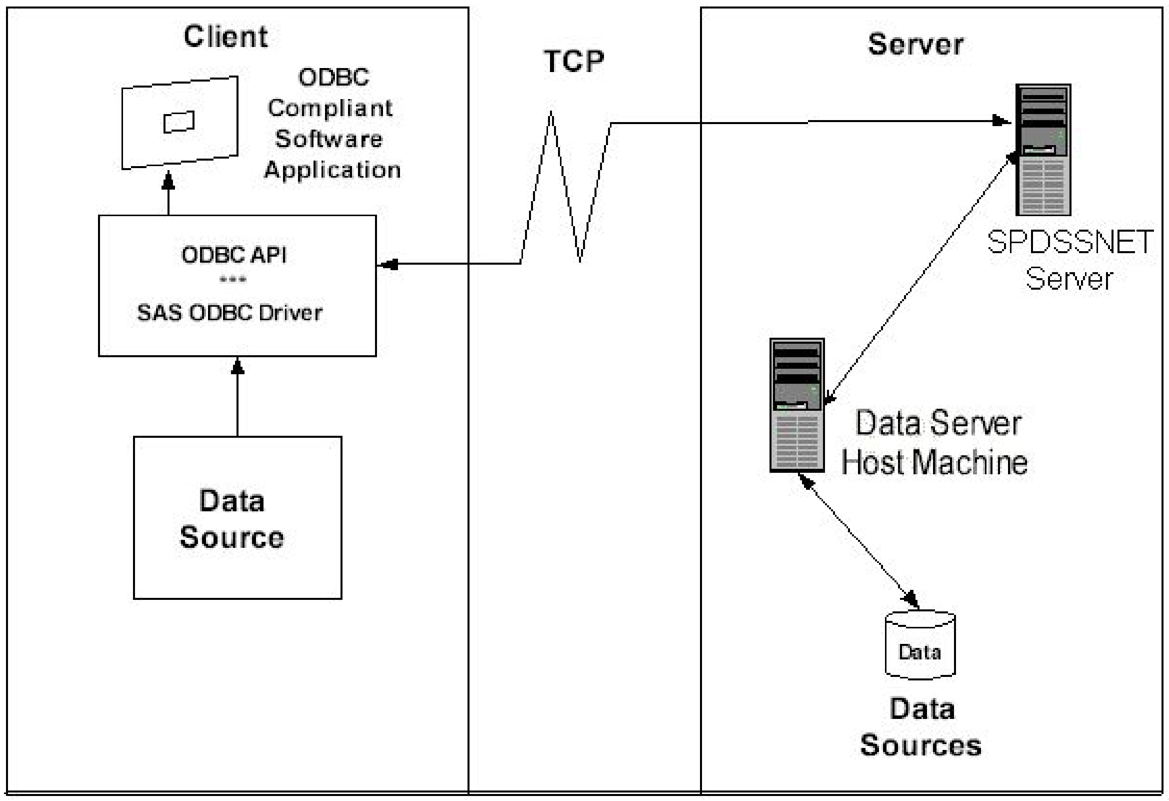


Figure 6.2 *Configure ODBC to Connect SPD Server Client to SPD SNET Server*



Preparing Your Client Machine for ODBC Installation

Before you create the OBDC data sources driver, you will need the following information from your network administrator:

- a user name and password that is defined by an SPD Server administrator
- the primary LIBNAME domain of the SPD Server (also called the DBQ)
- the port number of the SPD Name Server (also called the SERV)
- the machine name or IP address of the SPD Server Name Server (also called the HOST)
- any secondary LIBNAME domains you want to assign to the ODBC connection

Two Types of ODBC Connections

SPD Server software allows you to connect directly to an SPD Server host without going through the SPD SNET Server. Although connecting directly is the preferred method, connections via the SPD SNET Server are still supported.

Primary and Secondary LIBNAME Domains

When a connection to the SPD Server is established a primary LIBNAME domain is assigned. The primary LIBNAME domain is specified by the DBQ connection options parameter. Immediately after the connection is made, the SAS ODBC Driver assigns the secondary LIBNAME domains which are configured through the Libraries tab of the SAS ODBC Driver Configuration window.

ODBC Connections via the SPD SNET Server must have an **odbc.parm** file configured on the SPD SNET Server machine.

Configuring an ODBC Data Source to Connect Directly to an SPD Server

Once the SAS ODBC driver is installed, you will need to configure your ODBC data source. When you open the ODBC manager, you will get a display screen that allows you to enter information that points the OBDC driver to the data on the SPD Server.

1. From the Windows Start button, select **Start** ⇨ **Settings** ⇨ **Control Panel**.
2. Locate the ODBC Data Sources icon and open the Microsoft ODBC Data Source Administrator. The exact location of this program depends on your version of Windows.
3. Select the Add button, then select the SAS ODBC driver.
4. Enter a data source name (and description if desired).
5. Select the Servers panel and type in your two-part server name.
6. Click on the Configure box. The TCP Options window appears:
   - **Server Address:** Enter the network address of the machine on which the SPD Server is running.
   - **Server User Name:** Enter the user name as configured for a DBQ (SPD Server primary LIBNAME domain) on the SPD Server to which you will connect.
   - **Server User Password:** Enter the user password as configured for a DBQ (SPD Server primary LIBNAME domain) on the SPD Server host to which you will connect.
   - **Connection Options:** Enter the Connection Options as follows:
     - **DBQ='SPD Server primary LIBNAME domain',** this is the SPD Server LIBNAME domain
     - **HOST='Name Server node name',**this is the location of the host computer
     - **SERV='Name Server port number',**this is the port number of the SPD Server Name Server running on the HOST.
     - Any other SPD Server LIBNAME options. For more information, see the SPD Server 4.45: User's Guide section on "LIBNAME Options" on page 25.
7. Click **OK**, and then click **Add**, and select the **Libraries** panel.
8. Enter the DBQ name of a secondary LIBNAME domain in both the **Name** and **Host File** text fields.
9. Enter **spdseng** in the **Engine** text field.
10. Follow the syntax rules for the SQL Pass-Through libref statement for entering a value in the Options text field.

Configuring an ODBC Data Source for SPD SNET

Once the SAS ODBC driver is installed, you will need to configure your ODBC data source. When you open the ODBC manager, you'll get a display screen that allows you to enter information that points the OBDC driver to the data on the SPD Server.

1. From the Windows Start button, select **Start** ⇨ **Settings** ⇨ **Control Panel**.

2. Click on the ODBC icon and select the **Add** button.
3. Select the SAS ODBC driver.
4. Enter a data source name (and description if desired).
5. Select the Servers panel and type in the two-part server name. The second part of the server name should match the entry in the services file. In the example that follows that shows you how to edit the services file, the server name is **spdssnet**.
6. Click on the Configure box. The TCP Options window appears with four input fields that you fill:
   - **Server Address:** Enter the network address of the machine on which the SPD SNET Server is running.
   - **Server User Name:** Enter the user name as configured for a DBQ (SPD Server primary LIBNAME domain) on the SPD Server to which you will connect.
   - **Server User Password:** Enter the user password as configured for a DBQ (SPD Server primary LIBNAME domain) on the SPD Server host to which you will connect.
   - **Connection Options:** Enter the connection options as follows:
   - **DBQ='SPD Server primary LIBNAME domain':** this is the SPD Server LIBNAME domain.
   - **HOST='Name Server node name':** this is the location of the host computer.
   - **SERV='Name Server port number':** this is the port number of the SPD Server Name Server running on the HOST.
7. Click **OK**, and then click **Add**.

Editing the Services File on Your Machine - ODBC Details

Editing the Services file is required only for ODBC connections via the SPD SNET Server.

1. Find the Services file on your Windows machine. In Windows, the Services file is usually located in `c:\windows\services`
2. Open the Services file using a text editor.
3. The services file contains four columns. The rows of information can be sorted in port number order. Find the closest port number to the SPD Server port number, which you obtained from the network administrator. For more information, see"Preparing Your Client Machine for ODBC Installation " on page 58. This is where you insert the new information.
4. Add an entry to the Services file, on its own line, in proper numeric order, using the following syntax:

Table 6.1 How to Add Service Name and Port Number to the Services File

column1 <service name>	column2 <port number & protocol>	column3 <aliases>	column4 <comment>
spdssnet spdssnet=name assigned to server	nnnn/tcp nnnn=port number protocol is always /tcp	not required	not required

Remember: The service name, **spdssnet** must match the server name that you used in step 6 of "Configuring an ODBC Data Source for SPD SNET " on page 59. The port number must match the port number on which the SPD SNET Server is running.

Creating a Query Using an ODBC-Compliant Program

The following instructions create a query using Microsoft Access.

1. Start the SPD Server SNET Server.
2. Start Microsoft Access.
3. From the Microsoft Access main menu, select **File** ⇨ **Get External Table**.
4. Select **Link Table**.
5. Select **Files of Type**.
6. Select **ODBC Databases**.
7. Select the data source.

Using JDBC (Java) to Access SPD Server Tables

Read this information if you do not have Base SAS software on the network client, but you want to use the power of the Java programming language to query SPD Server tables from any client on the network that has a browser. You must have SPD Server tables on the network and SPD Server and SPD SNET Servers running on the same server as the Web server to use JDBC to access SPD Server tables.

Why Would I Want to Use JDBC?

You might want to use JDBC if you have SPD Server tables available on your network and one or more of the following is true:

- You do not have Base SAS software on the network client to process the data sets.
- You want to distribute the information across your corporate intranet through a Web page.
- The clients on your network are varied: UNIX boxes, Windows PCs, and workstations. One thing they might have in common is browser access to your intranet.

- The audience for the information understands Web browsing and wants point-and-click access to the information.
- You want to distribute the information over the World Wide Web.
- Your planned application requires the power of the Java programming language.

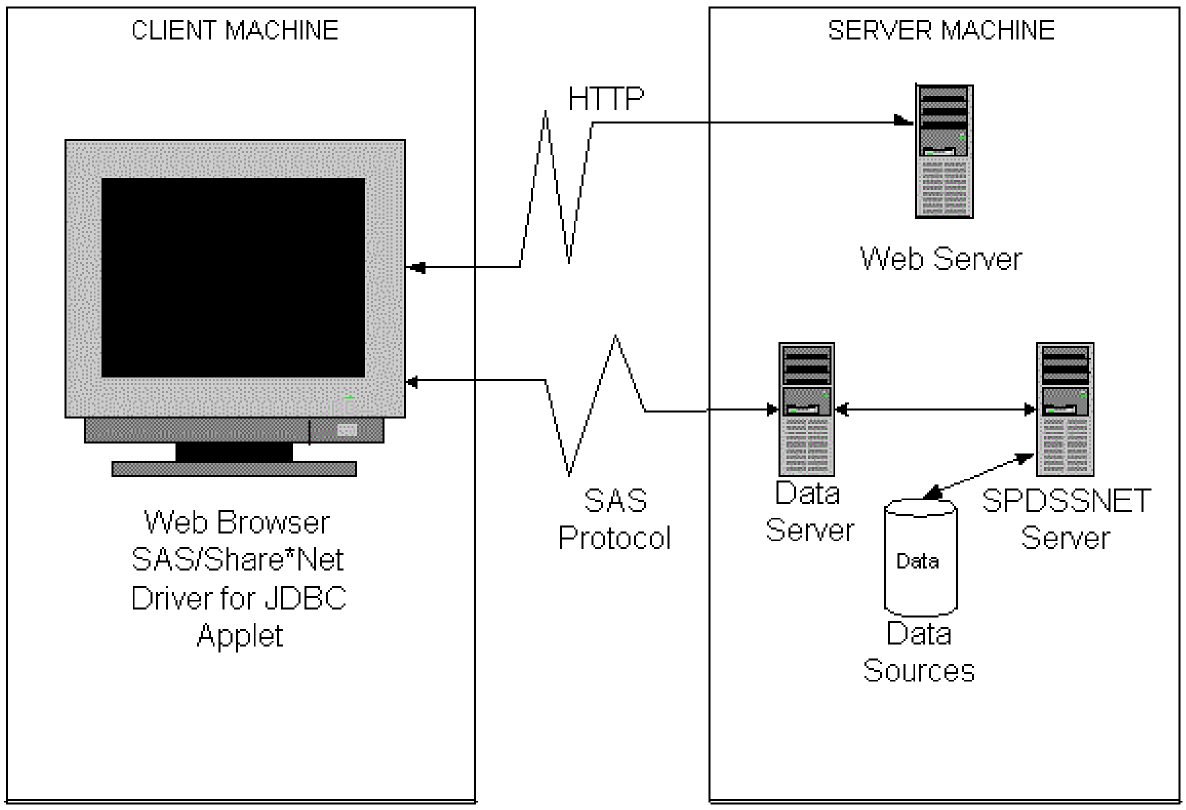
How Is JDBC Set Up on the Server?

JDBC is usually set up on the server at the time the SPD Server is installed. The process is covered in the SPD Server installation manual.

How Is JDBC Set Up on the Client?

The client needs a browser set up to accept Java applets, such as

- Netscape Navigator, Release 3.0 or later
- Microsoft Internet Explorer, Release 3.02 or later



How Do I Use JDBC to Make a Query?

1. Log on to the World Wide Web and enter the URL for the Web page that contains the JDBC code.
2. Click on the desired information.
3. JDBC handles the request, formats the information, and returns the result to the Web page.

JDBC Code Examples and Tips

The following lines must be a part of the HTML file for JDBC:

```
<applet code=CLASSPATH.*.class codebase=../ width=600 height=425>
<param name=url value=jdbc:sharenet://spdssnet_node:PORT>
<param name=dbms_options value=DBQ='LIBNAME' HOST='host_node' SERV='NNNN'>
<param name=spdsuser value=userid>
<param name=sharePassword value=thepassword>
<param name=shareRelease value=V9>
<param name=dbms value=spds>
</applet>
```

Line 1:

- **CLASSPATH** points to the class path set up where the JDBC driver is installed.
- ***.class** is the name of the Java class that consumes all of the <PARAM name=...> lines.

Line 2:

- **spdssnet_node** is the node name of the machine on which the SPD SNET Server is running.
- PORT=port number of the machine on which the SPD SNET Server is running.

Line 3:

- **value=DBQ='libref'** is the LIBNAME domain of the SPD Server.
- **HOST='host_node'**is the location of the SPD SNET Server.
- **SERV='NNNN'**is the port number of the Name Server.

Line 4:

- **spdsuser** value=**user ID** is the user ID that queries the SPD Server table.

Line 5:

- **sharePassword** value=**thepassword** is the password of the user ID that will make the query.

Line 6:

- **shareRelease** value=V9 is the version of the driver you are using. This must not be altered.

Line 7:

- Sets the foreign database property on the JDBC driver. This means that the server is not SAS and JDBC should not create a DataBaseMetaData object. See the examples below for how to get around this.

Limitations of Using JDBC with SPD Server

JDBC Used with SAS Versus JDBC Used with SPD Server

SPD Server is treated as a foreign database. SPD Server clients cannot query the JDBC metadata class for available tables and other metadata. Users must write their own queries to do this.

Example JDBC Query for Getting a List of Tables

(JDBC Used with SPD Server)

```
SELECT '' AS qual,
LIBNAME AS owner,
MEMNAME AS name,
MEMTYPE AS type,
MEMNAME AS remarks FROM dictionary.tables AS tbl
WHERE ( memtype = 'DATA' OR memtype = 'VIEW' OR memtype = 'SYSTEM TABLE' OR
        memtype = 'ALIAS' OR memtype = 'SYNONYM')
AND (tbl.LIBNAME NE 'MAPS' AND tbl.LIBNAME NE 'SASUSER' AND tbl.LIBNAME NE 'SASHELP')
ORDER BY type, qual, owner, name
```

Example JDBC Query for Getting Metadata about a Specific Table

(Your data file)

```
SELECT '' AS qual,
LIBNAME AS owner,
MEMNAME AS tname, name,
length AS datatype,
type || ' ',
length AS prec,length,
length AS scale, length AS radix, length AS nullable,label,
FORMAT FROM dictionary.columns AS tbl
WHERE memname = 'your data file'
AND (tbl.LIBNAME NE 'MAPS'
     AND tbl.LIBNAME NE 'SASUSER'
     AND tbl.LIBNAME NE 'SASHELP')
```

Using htmSQL to Access SPD Server Tables

Read this section if you do not have Base SAS software on the network client, but you want to use the point-and-click convenience of a Web page to query SPD Server tables from any browser-enabled client on the network. You must have SPD Server tables available for use, htmSQL loaded and configured on a UNIX or Windows operating system, and SPD Server and SPD SNET Server running.

Why Would I Want to Use htmSQL?

You might want to use htmSQL if you have SPD Server tables available on your network and one or more of the following is true:

- You do not have Base SAS software on the network client to process the data sets.
- You want to distribute the information across your corporate intranet through a Web page.
- The clients on your network are varied: UNIX boxes, Windows PCs, and workstations. One thing they might have in common is browser access to your intranet.
- The audience for the information understands Web browsing and wants point-and-click access to the data.

- You would like to use the JDBC option to extract the information but cannot permit Java applets to run on your network browsers.
- You want to distribute the information over the World Wide Web.
- Your developers are familiar with SQL and HTML.

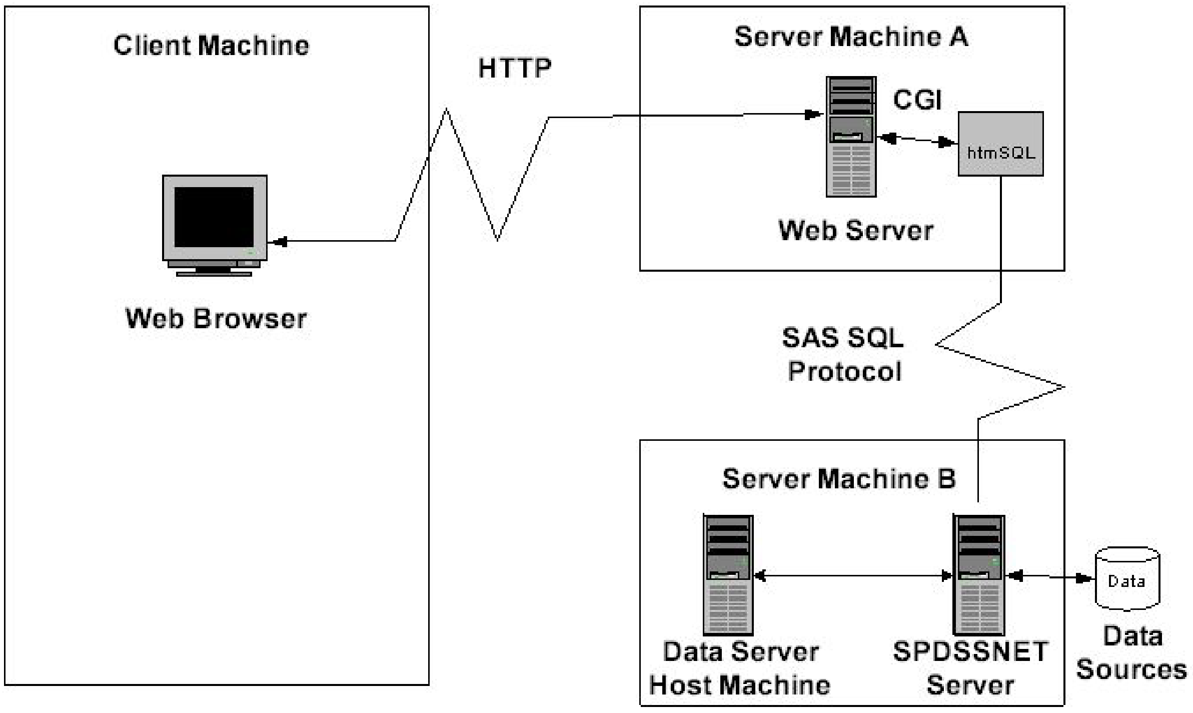
How Is htmSQL Set Up on the Server?

- htmSQL is usually set up on the server at the time the SPD Server is installed. The process is covered in the SPD Server installation manual.
- htmSQL must be installed on the Web server, and youneed the name of a data source that points to the SPD SNET Server and to the specific LIBNAME domain that contains the SPD Server data you are interested in.

How Is htmSQL Set Up on the Client?

HtmSQL requires nothing more than a browser on the network or Web client.

Figure 6.3 *htmSQL Configured on an SPD Server Client*



How Do I Use htmSQL to Make a Query?

1. Log on to the World Wide Web and enter the URL for the Web page that contains the htmSQL code.
2. Click on the desired information.
3. htmSQL handles the request, formats the information, and returns the result to the Web page.

Examples of Setting Up an htmSQL Web Page

SAS Institute maintains a Web site that explains the technical details of setting up htmSQL Web pages. In some cases, there are references to the SAS/SHARE product. The rules for setting up htmSQL for either the SPD Server or SAS/SHARE are virtually the same.

The SAS Institute Web page for htmSQL is `http://support.sas.com/rnd/web/intrnet/htmSQL/index.html`.

Using SQL C API to Access SPD Server Tables

Read this section if you do not have Base SAS software on the network client, but you want to provide your network client machines with the capability of accessing SPD Server tables, using SQL query methods. You must have SPD Server tables available for use, SPD Servers and SPD SNET Servers running, and Network client machines capable of running C/C++ programs.

Why Would I Want to Use SQL C API?

You have SPD Server tables available on your network and one or more of the following might be true:

- You do not have Base SAS software on the network client to process the data sets.
- You want to distribute the information across your corporate intranet.
- The clients on your network are varied: UNIX boxes, Windows PCs, workstations. One thing they might have in common is the ability to run C/C++ programs.
- Your developers are familiar with SQL and C/C++.

Chapter 10, “SAS Scalable Performance Data (SPD) Server SQL Access Library API Reference,” on page 165 contains additional information about SQL C API.

Chapter 7

SAS Scalable Performance Data (SPD) Server Dynamic Cluster Tables

Introduction to Dynamic Cluster Tables

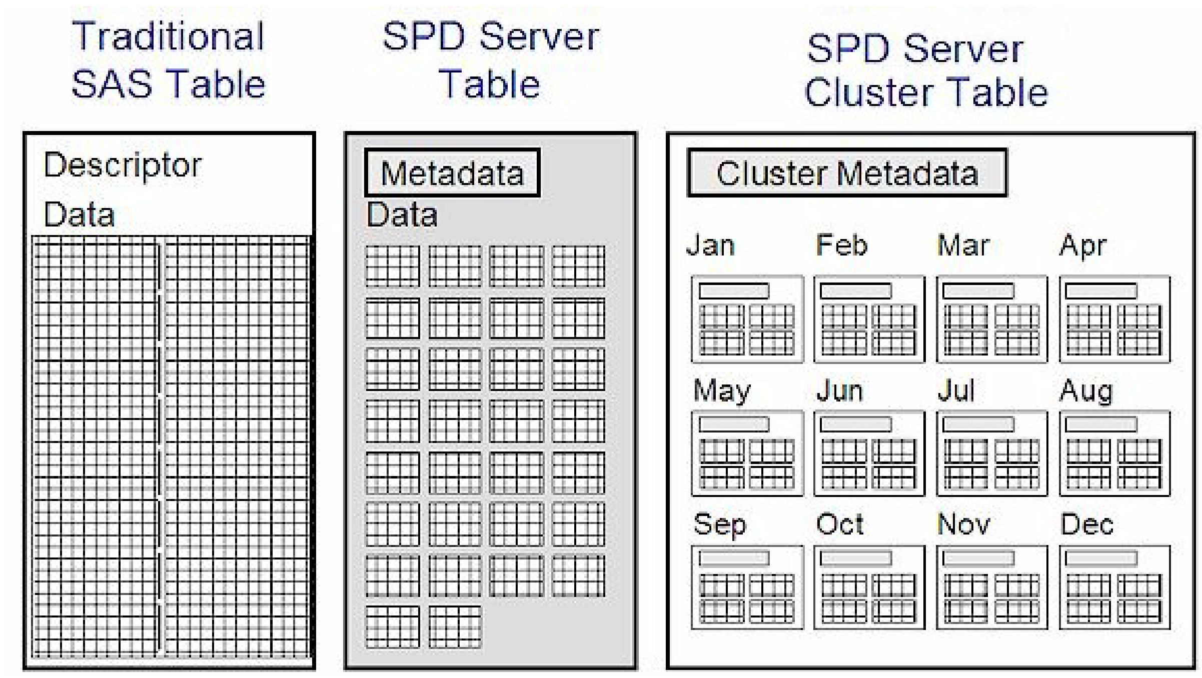
SPD Server is designed to meet the storage and performance demands that are associated with processing large amounts of data using SAS. As the size of the data grows, the demand to process that data increases, and storage architecture must change to keep up with business needs.

SPD Server offers dynamic cluster tables. Earlier releases of SPD Server provided a type of cluster table called the time-based partitioning table. To optimize the benefits of the clustering, the SPD Server administrator can use dynamic clusters to partition SPD Server data tables for speed and enhanced I/O processing. Clustering is performed using metadata that when combined with SPD Server functionality, provides parallel processing capabilities for loading and querying data tables. Parallel processing can accelerate performance and increase the manageability, flexibility, and scalability of very large data stores.

Dynamic Cluster Table Structure

The SPD Server dynamic cluster table can be considered as part of a hierarchy of tables with increasing sophistication:

Figure 7.1 *Figure 7.1: SAS Table, SPD Server Table, and SPD Server Cluster Table Structures.*



- **Traditional SAS tables**are single files that contain the data descriptors and the table data. Data values are the columns, and the descriptors are the metadata that describe the column and data formatting that the table uses. If a traditional SAS table contains one or more indexes, they are stored in a separate file.
- **SPD Server tables**use component files to store tables. One component file stores the stream of data values. Another component file stores the column and data descriptors. If you create an index for a column or a composite of columns, SPD Server creates two separate component files (a *.hbx file and a *.idx file) for each index.
- **SPD Server Cluster tables**are virtual table structures. SPD Server cluster tables consist of members. Each member is an SPD Server table. All members must share the same metadata formats and organization. SPD Server cluster tables use the metadata to manage the data that is contained in the members.

The SPD Server dynamic cluster table structure provides architecture that enables flexible loading and rapid storage and processing for very large data tables. Using dynamic cluster tables, loading data, removing data, and refreshing tables in very large data marts become easier and more timely. Dynamic cluster tables provide organizational features and performance benefits that traditional SAS tables and SPD Server tables do not have.

Benefits of Dynamic Cluster Tables

Overview of Dynamic Cluster Tables

Organizing SPD Server data into dynamic cluster tables creates an architecture that supports parallelism, enhanced data flexibility and manageability, and significantly improved speed in robust data warehousing environments that use large and very large data tables.

For example, you can add new data or remove historical data from very large tables by accessing only the member tables that are affected by the change. You can access the individual member tables in parallel. This strategy reduces the time that is needed for the job to complete and uses simple commands. Furthermore, a complete refresh of a dynamic cluster table can occur using a fraction of the disk space that is needed to refresh a large traditional SAS or SPD Server table that contains the same amount of data.

Parallel Loading

Because dynamic cluster tables are virtual tables that consist of numerous small SPD Server tables, the architecture enables parallel loading and processing. Cluster table loads and refreshes are broken down into multiple tasks that can be performed concurrently. Separate SAS MP CONNECT jobs manage the parallel loading and processing.

The scalability of parallel loading with dynamic cluster tables depends on the scalability of the server I/O and the number of processors on the server.

Parallel loading requires multiple concurrent writes to disk. If the I/O hardware does not scale appropriately, the loading process can degrade performance.

SPD Server can create multiple indexes on the same table in parallel, and index creation is a CPU-intensive process.

When sufficient processing power is available, parallel index creation in SPD Server is highly scalable.

The creation process for each index is multi-threaded. A single index creation can use multiple CPUs on a server if they are available, which greatly improves performance.

Fast and Economical Refreshes

Refreshing a dynamic cluster table requires only a fraction of the disk space that a traditional SPD Server table with the same amount of data would require. The dynamic cluster table architecture allows users to refresh many large tables concurrently, while conserving disk and I/O resources. With very large traditional SAS or SPD Server tables, available disk space often limits the number of tables that can be concurrently refreshed.

In the life cycle of data warehouses, tables can be refreshed to recapture disk space when rows have been updated or deleted, or to reorder data for optimized performance. However, refreshing a table can temporarily use twice the disk space of the table itself. With very large tables, disk space can be a limiting factor when updating a data warehouse or data mart. When disk space is limited on a server, the amount of data that can be refreshed at any given time is constrained. The window of time that is required to load and refresh can become huge.

Because dynamic cluster tables can be quickly unbound into smaller SPD Server tables, refreshing dynamic cluster tables does not use twice the disk space of the original table. Instead, only twice the disk space of the largest member table in the dynamic cluster table is required.

After the dynamic cluster table is unbound, disk space equal to the first member table is required to perform a refresh. A backup of the refresh is created, and then the old version is deleted, creating more available disk space. The refresh process repeats for each successive member table until all members in the dynamic cluster table have been refreshed and updated. Then, the member tables are merged into a dynamic cluster table once again.

When a server has enough disk space and I/O resources to refresh more than one member table at a time, the benefits of parallel processing can be realized.

Creating and Controlling Dynamic Cluster Tables

Creating dynamic cluster tables in SPD Server is simple and straightforward. The following operations are associated with creating and controlling dynamic cluster tables:

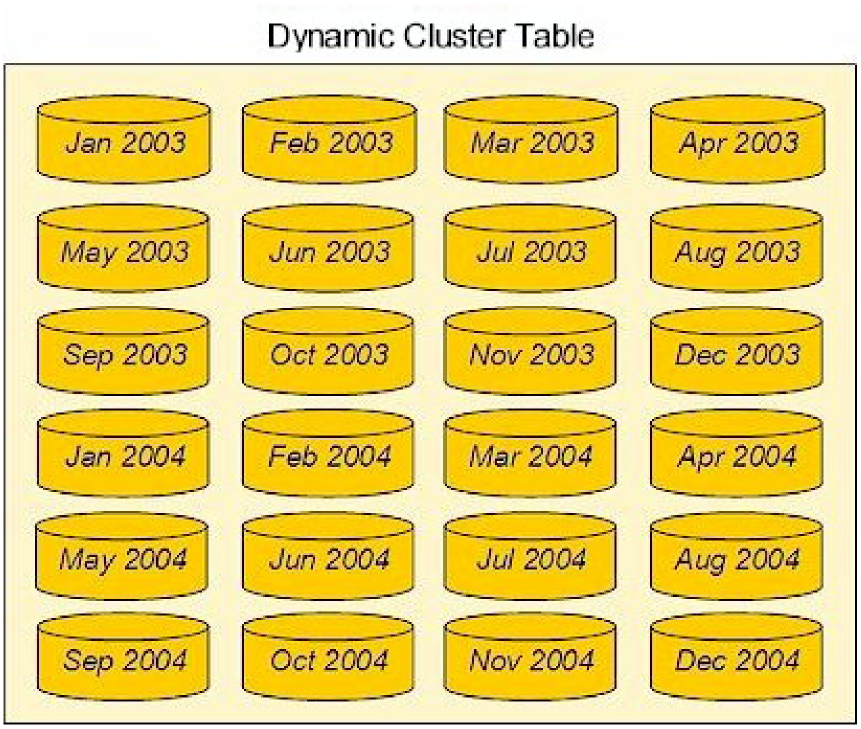
Create a Dynamic Cluster Table

To create dynamic cluster tables in SPD Server, you must have a set of related SPD Server tables that you want to cluster, such as tables that contain monthly sales histories. The SPD Server tables that you want to cluster must all be in the same domain, and must use identical table structures (columns and indexes) and compression. However, member table partition sizes and member table owners can vary. These requirements ensure the metadata compatibility that is necessary to create dynamic cluster tables in SPD Server.

Once the related SPD Server tables are organized, a simple PROC SPDO command is used to bind the tables into a dynamic cluster table.

The following graphic represents a dynamic cluster table with 24 members. Each member table is an SPD Server table that contains monthly sales transactions:

Figure 7.2 *Figure 7.2: Representative Dynamic Cluster Table*



The following code shows the PROC SPDO command syntax that is used to create dynamic cluster tables from the member tables:

```
PROC SPDO library=domain-name ;
   cluster create Sales_History
      mem=sales200301
      mem=sales200302
      mem=sales200303
      mem=sales200304
      mem=sales200305
      mem=sales200306
      mem=sales200307
      mem=sales200308
      mem=sales200309
      mem=sales200310
      mem=sales200311
      mem=sales200312
      mem=sales200401
      mem=sales200402
      mem=sales200403
      mem=sales200404
      mem=sales200405
      mem=sales200406
      mem=sales200407
      mem=sales200408
      mem=sales200409
      mem=sales200410
      mem=sales200411
      mem=sales200412
   maxslot=36 ;
quit ;
```

PROC SPDO uses a LIBRARY statement to identify the domain that contains the tables to be clustered. The **cluster create** syntax specifies the name of the dynamic cluster table to be created (Sales_History).

The **mem=** syntax identifies the members of the cluster table. The tables in the previous example represent monthly sales transactions. This example uses 24 monthly sales tables for the years 2003 and 2004.

The **maxslot=** specification specifies the maximum number of members that are allowed in the dynamic cluster table Sales_History.

The "Dynamic Cluster Table Examples " on page 86 section contains more extensive code examples of creating dynamic cluster tables.

Dynamic Cluster Table Access Control

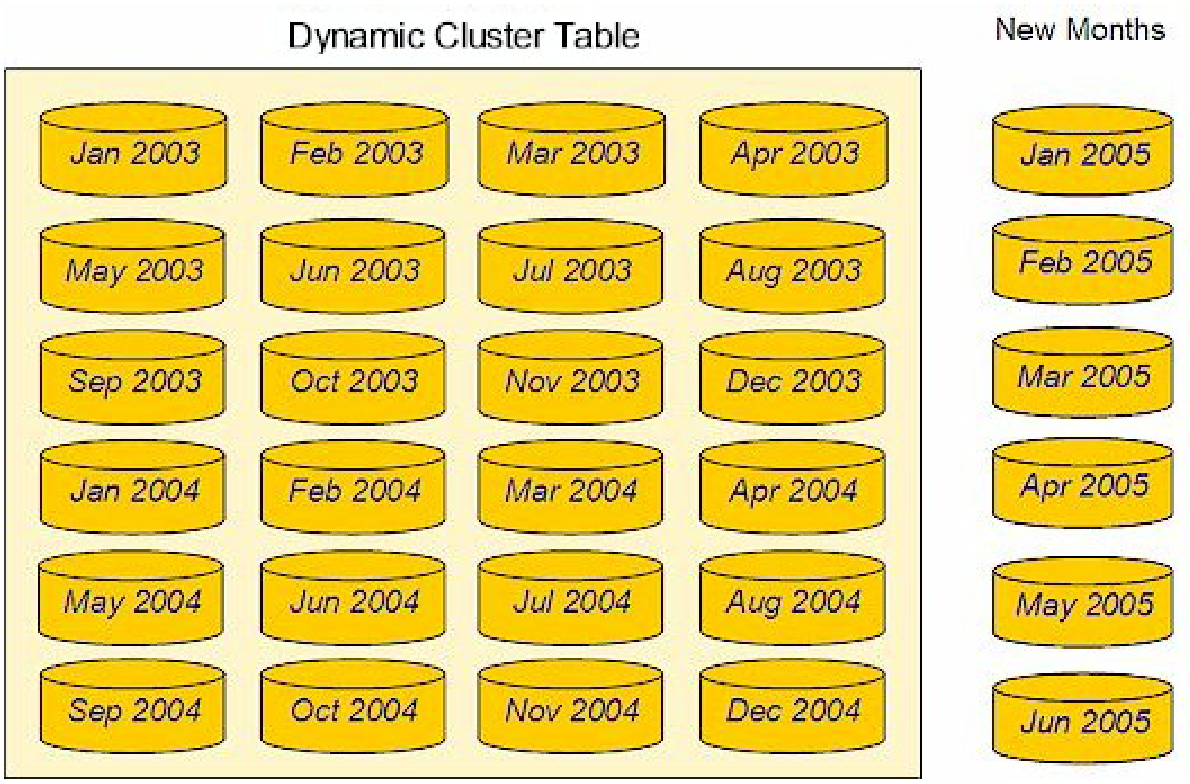
A user must have SPD Server control access on any member tables that are used in the `cluster create` or `cluster add` commands. A user must also have SPD Server control access on the dynamic cluster table itself to submit a `cluster undo` command. There is no restriction on table ownership, as long as the user has control access on all member tables. All users that have access to a domain have default control access on tables that were created by the user Anonymous within that domain. ACLs can be defined on a dynamic cluster table after it is created, and the permissions that are specified in the dynamic cluster table ACL are applied when SPD Server accesses the dynamic cluster table. Any individual ACL that is defined on a member table does not apply during the time when the member table is part of a created dynamic cluster table.

Add Tables to a Dynamic Cluster

To add tables to a dynamic cluster table, you must have an existing dynamic cluster table. The SPD Server tables that you want to add to the dynamic cluster table must all be in the same domain as the dynamic cluster table. These tables must use identical table structures (columns and indexes) and compression. However, partition sizes and owners can vary. These requirements ensure the metadata compatibility that is required to add to a dynamic cluster table.

Once the tables to be added are organized, a simple PROC SPDO command is used to add the new tables to an existing dynamic cluster table. In the following graphic, sales tables for the first six months of 2005 are set up to be added to the dynamic cluster table that contains monthly sales transaction data for 2003 and 2004:

Figure 7.3 *Figure 7.3: New Monthly Data Added to Cluster Table*



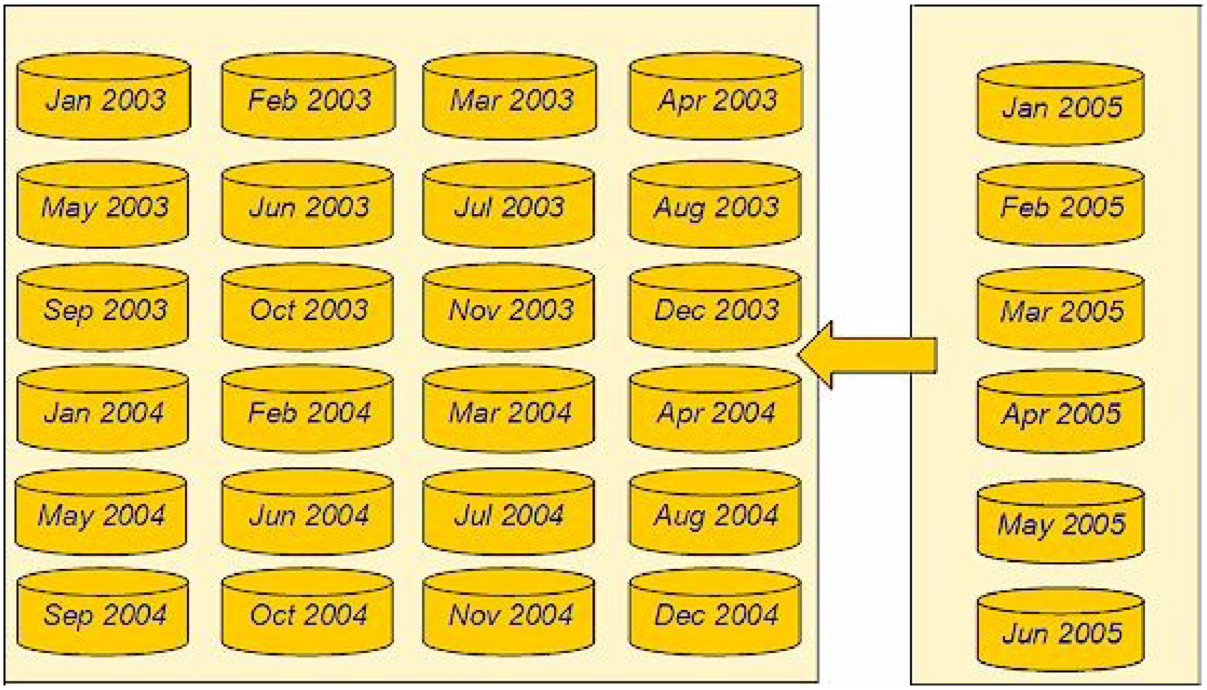
The following code shows the PROC SPDO command syntax that is used to add new tables to an existing dynamic cluster table:

```
PROC SPDO library=domain-name ;
   cluster add Sales_History
      mem=sales200501
      mem=sales200502
      mem=sales200503
      mem=sales200504
      mem=sales200505
      mem=sales200506 ;
quit ;
```

PROC SPDO uses a LIBRARY statement to identify the domain that contains the existing dynamic cluster table that you want to add to. The **cluster add** syntax specifies the name of the dynamic cluster table that you want to add to (Sales_History).

The **mem=** syntax identifies the members that form the table to be added to the existing dynamic cluster table. In the following graphic, six tables that include monthly sales transactions for the first half of 2005 are set up to be added to the existing dynamic cluster table of 2003 and 2004 sales transactions data:

Figure 7.4 *Figure 7.4: Adding Member Tables to a Dynamic Cluster Table*



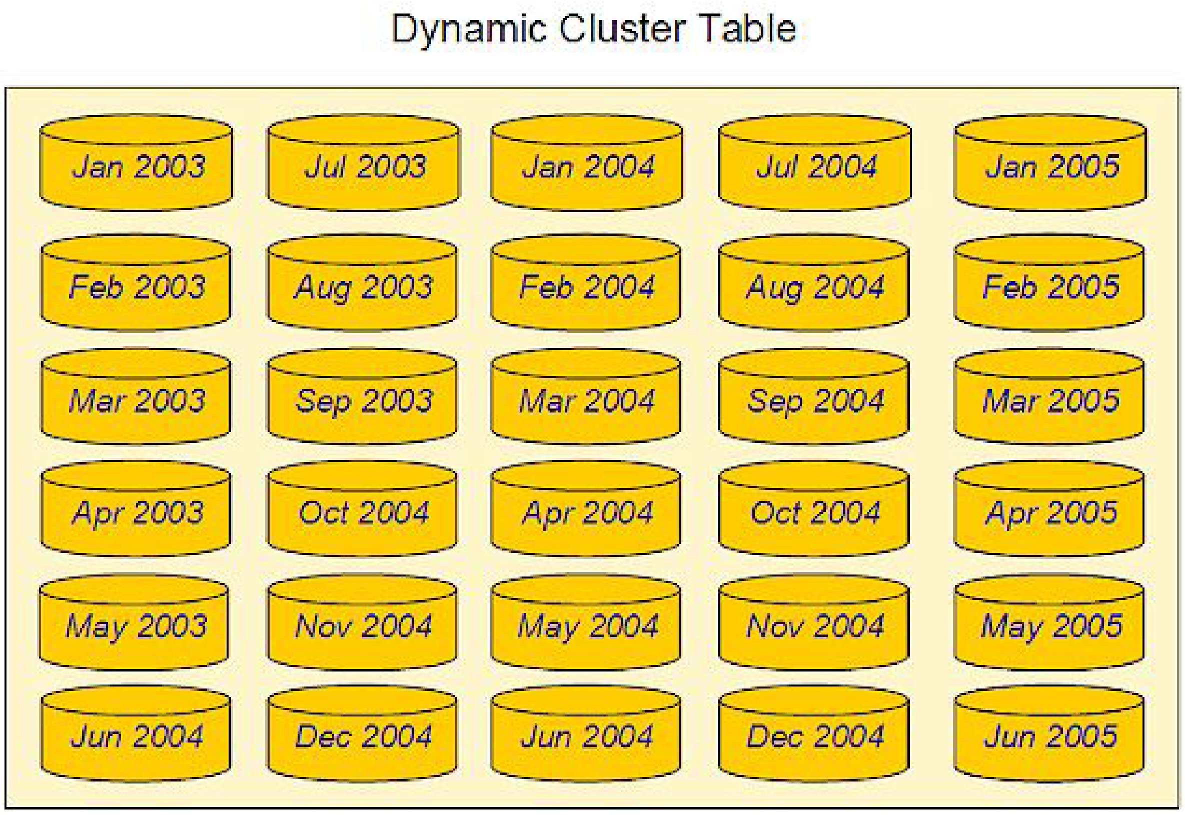
See the “Dynamic Cluster Table Examples ” on page 86 section for a more extensive code example of adding to a dynamic cluster table.

Undo Dynamic Cluster Tables

To undo a dynamic cluster table, you must have an existing dynamic cluster table. Undoing the dynamic cluster table simply reverts the table back to unbound SPD Server tables. Undoing a dynamic cluster table is required to remove a specific member table from a dynamic cluster table, to add data to a specific member table in the dynamic cluster table, or to completely refresh a specific member table that belongs to the dynamic cluster table.

The following graphic represents a dynamic cluster table with 24 members. Each member contains monthly sales transactions for the years 2003 and 2004:

Figure 7.5 *Figure 7.5: Dynamic Cluster Table with 24 Members*



PROC SPDO is used to undo the existing dynamic cluster table.

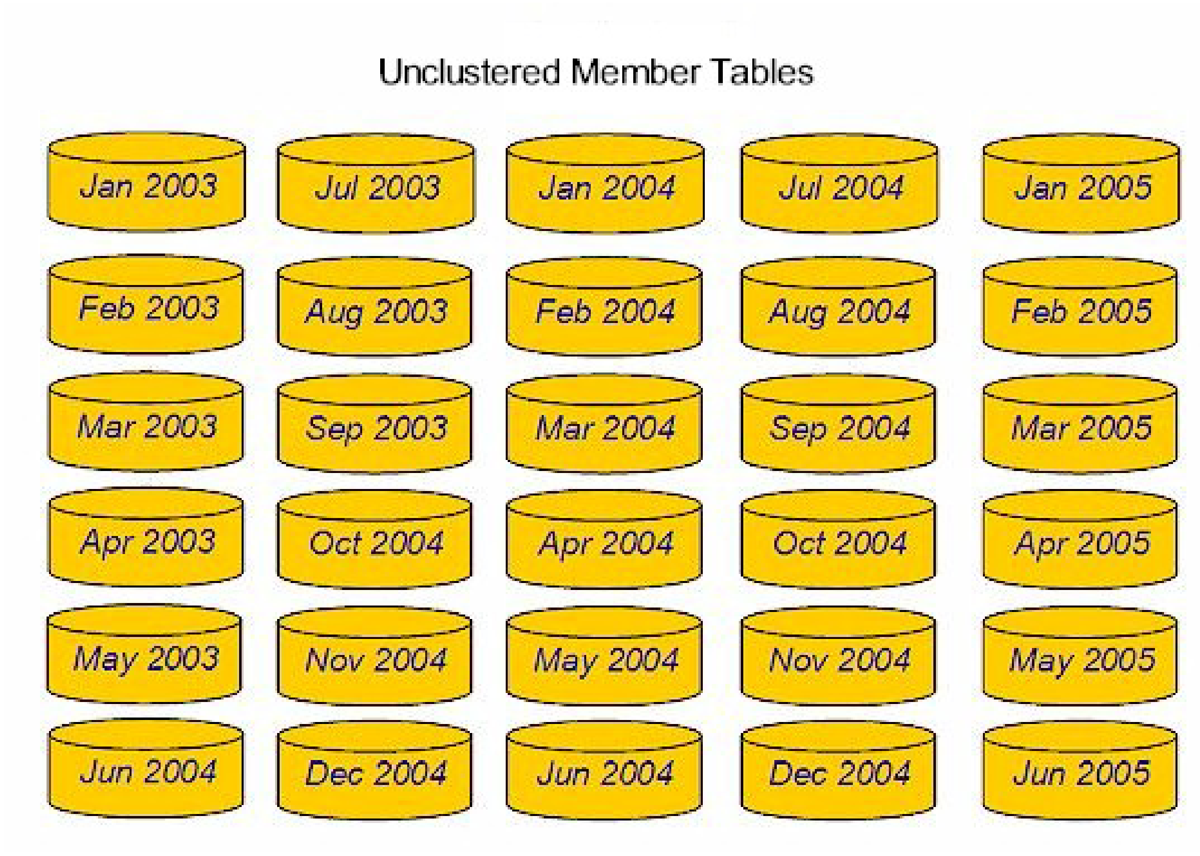
The following code shows the PROC SPDO command syntax that is used to undo an existing dynamic cluster table:

```
PROC SPDO library= domain-name ;
   cluster undo Sales_History ;
quit ;
```

PROC SPDO uses a LIBRARY statement to identify the domain that contains the existing dynamic cluster table that you want to undo. The **cluster undo** syntax specifies the name of the dynamic cluster table that you want to undo (Sales_History).

The following graphic represents the previous dynamic cluster table, now unbound.

Figure 7.6 *Figure 7.6: Unbound Dynamic Cluster Table*



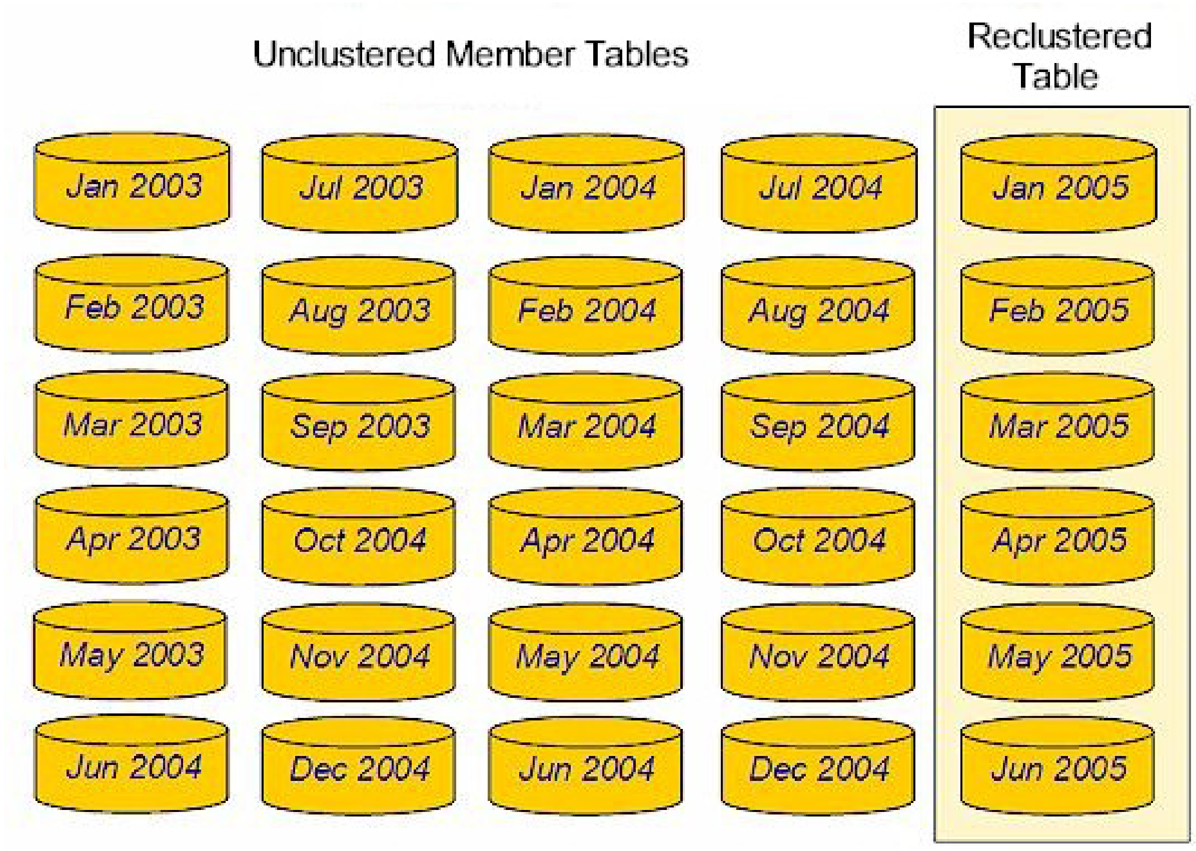
See the "Dynamic Cluster Table Examples " on page 86 section for a more extensive code example of undoing a dynamic cluster table and then refreshing it.

Refresh Dynamic Cluster Tables

To refresh a dynamic cluster table, you perform the same actions that are required to undo a dynamic cluster table. Then, you recreate the dynamic cluster table after you add a member table or change an existing member table. An example of refreshing an SPD Server dynamic cluster table is updating on a monthly basis a dynamic cluster table whose members are the 24 previous months of sales transaction data.

To refresh a dynamic cluster table, use sequential PROC SPDO commands to **`cluster undo`** and **`create cluster`** with the desired member tables. The dynamic cluster table is first undone. Table changes are made, and then the dynamic cluster table is rebound again. The following example unbinds the sales transactions tables for 2003 and 2004, and then refreshes the dynamic cluster table with sales transactions tables for the first six months of 2005:

Figure 7.7 *Figure 7.7: Refreshed Dynamic Cluster Table*



See "Dynamic Cluster Table Examples " on page 86 for a more extensive code example of unbinding a dynamic cluster table, and then refreshing it by recreating it with different member tables.

Modify Dynamic Cluster Tables

The PROC SPDO command set for dynamic clusters provides a **`cluster modify`** cluster command. The usage syntax for the **`cluster modify`** command is

```
CLUSTER MODIFY clustername
  MINMAXVARLIST=(varname1 <varname2 varname3 ...>);
```

The **`cluster modify`** command sets a MINMAXVARLIST attribute on one or more variables that belong to an existing dynamic cluster. The variable names that are specified in the **`cluster modify`** command must exist in the cluster and the variables must not have a pre-existing MINMAXVARLIST setting. When the SPD Server runs the **`cluster modify`** command, the dynamic cluster is unclustered while the variable modifications are made to the individual member tables. The cluster is recreated after the MINMAXVARLIST changes are completed. Control permission and exclusive access to the dynamic cluster is required in order to run the **`cluster modify`** command. SPD Server performs a full table scan to initialize the MINMAXVARLIST values in each member table, so the processor time that is required to perform the **`cluster modify`** command is directly related to the size of the tables that belong to the cluster. If an error occurs while the **`cluster modify`** command is running, the cluster cannot be recreated and the user will need to manually recreate the cluster using the **`cluster create`** command.

Creating Dynamic Clusters with Unique Indexes

The CLUSTER CREATE command in PROC SPDO has an option that allows you to specify whether the unique indexes that are defined in the member tables should be validated and marked as unique in the cluster. If the UNIQUEINDEX option is set to No, then unique indexes are not validated, and the cluster metadata does not mark the indexes as unique within the cluster. If the UNIQUEINDEX option is not specified, then the setting defaults to YES. In this case, the indexes are validated and marked unique within the cluster. The processing that is required to validate the unique indexes depends on the number of rows in the tables. The process can take considerable time for larger tables. If the validation process is chosen and the indexes are not unique, the CLUSTER CREATE command will fail.

```
CLUSTER CREATE clustername
MEM=member_table1
MEM=member_table2
 ...
MEM=member_table_n
MAXSLOT=n
UNIQUEINDEX=<yes|no>;
```

Dynamic Cluster BY Clause Optimization

Overview of Optimizing BY Clauses

When you use SPD Server dynamic clusters, you can create huge data sets. If the huge data sets need further manipulation by some SAS job, it might be better to sort them for more efficient processing. Traditional processing of huge data sets can overuse or overwhelm available resources. The resulting lack of available runtime or processor resources can prohibit you from running full-table scans and manipulating table rows, which are required to sort huge data sets for subsequent processing.

SPD Server provides dynamic cluster BY clause optimization to reduce the need for a large amount of processor resources when evaluating BY clauses. The dynamic cluster BY clause optimization uses SPD Server to join individually created SPD Server member data sets so that the data sets appear to be single data set, while still keeping the individual member data sets intact. The dynamic cluster BY clause optimization uses the SORTEDBY metadata of the member data sets to bypass most of the sorting that is required to perform the implicit BY clause ordering. With the SORTEDBY metadata of each member, SPD Server merges the member data sets in the dynamic cluster by using each member data set's order. No additional SPD Server work-space is required, and the ordered data set records are returned with minimum delay since member sorting is eliminated.

To use dynamic cluster BY clause optimization, you need to build the dynamic cluster table a certain way. All of the member tables in your dynamic cluster table need to be sorted by the same columns that you need to use in the BY clause. When you build your dynamic cluster table from member tables that are presorted by your BY clause columns, your dynamic cluster table can use the BY clause optimization.

When a BY clause is run that matches the SORTEDBY column order of the dynamic cluster table member tables, SPD Server performs the BY clause without using sort work-space or experiencing first-record latency. SPD Server uses the presorted member tables to perform an instantaneous interleave. By using the presorted member tables, the dynamic cluster BY clause optimization enables you to perform operations on huge data sets that would be impossible to handle otherwise.

For example, suppose that you have a system that has sufficient CPU, memory, and work-space resources to sort a 50-GB data set in a reasonable amount of time. However, suppose this system accumulates 50 GB of new data every month, so that after 12 months, the data sets require 600 GB of storage. The system cannot handle sorting 600 GB of data to process queries that are based on the previous 12-month period. If you use SPD Server to create a dynamic cluster table from the 12 50-GB member tables, you can store each rolling month of data in an SPD Server member table, and then sort it like the other dynamic cluster table member tables, and then add the new member table to the 600-GB dynamic cluster table. Now you can use the dynamic cluster BY clause optimization to run SAS steps that use BY clauses on the 600-GB cluster. For example, you can run a DATA step MERGE statement that uses the dynamic cluster table as the master source for the MERGE statement. The BY clause from the MERGE statement triggers the dynamic cluster BY clause optimization. As a result, the operation completes in the time that it takes to interleave the individual member tables, using no SPD Server work-space and without experiencing any implicit BY sort delays.

Dynamic cluster BY clause optimization allows the BY optimization to be combined with certain WHERE clauses on dynamic cluster tables. For the WHERE clause optimization to work, SPD Server must be able to determine whether the WHERE clause is trivially true or trivially false for each member table in the dynamic cluster table. To be trivially true, a WHERE clause must find the clause condition true for every row in the member table. To be trivially false, a WHERE clause must find the clause condition false for every row in the member table.

SPD Server keeps metadata about indexed values in dynamic cluster table member tables, and if the WHERE clause criteria can be determined as true or false based on the dynamic cluster table's member table metadata, the WHERE clause optimization is possible on a member-by-member basis for the entire dynamic cluster table. Suppose that member tables of a dynamic cluster table all have an index on the column QUARTER (1=Jan-Mar, 2=Apr-Jun, 3=Jul-Sep, 4=Oct-Dec). Suppose that you need to run a DATA step MERGE statement that uses the expression WHERE QUARTER=2. Because the QUARTER column is indexed in all of the member tables, SPD Server uses the BY clause optimization to determine that the WHERE clause is trivially true. SPD Server then evaluates the expression only on the member tables for April, May, and June without using any SPD Server work-space. When the WHERE clause can be determined as trivially true or trivially false for each member table of the dynamic cluster table in advance, the BY clause optimization performs the BY processing only on the appropriate member tables.

The dynamic cluster BY clause optimization is triggered when member tables all have an applicable SORTEDBY ordering for the BY clause that is asserted. When the SORTEDBY ordering is strong (validated), SPD Server does not perform checks to verify the order of BY variables that are returned from the member table. When the SORTEDBY ordering is weak (such as from a SORTEDBY assertion that was a data set option), additional checking is performed to verify the order of BY variables that are returned from the member table. If an invalid BY variable order is detected, SPD Server terminates the BY clause and displays the following error message:

```
ERROR: Clustered BY member violates weak
        sort order during merge.
```

Dynamic Cluster BY Clause Optimization Example

Consider a database of medical patient insurance claims, with quarterly claims data sets that are named ClaimsQ1, ClaimsQ2, ClaimsQ3, and ClaimsQ4. Each quarterly claims table is sorted by columns that are named PatID (for Patient ID) and ClaimID (for Claim ID). The member tables are combined into a dynamic cluster table that is named ClaimsAll. The following example shows the code:

```
DATA SPDS.ClaimsQ1;
...
run;

DATA SPDS.ClaimsQ2;
...
run;

PROC SORT DATA=SPDS.ClaimsQ1;
 BY PatID ClaimID;
run;

PROC SORT DATA=SPDS.ClaimsQ2;
 BY PatID ClaimID;
run;

PROC SPDO LIB=SPDS;
create cluster ClaimsAll;
quit;
```

Consider the DATA step MERGE statement to be submitted to the ClaimsAll dynamic cluster table:

```
DATA SPDS.ToAdd SPDS.ToUpdate;
MERGE SPDS.NewOnes(IN=NEW1)
      SPDS.ClaimsAll(IN=OLD1);
BY PatID ClaimID;

SELECT;
WHEN(NEW1 and OLD1)
 DO;
 OUTPUT SPDS.ToUpdate;
 end;
WHEN(NEW1 and not OLD1)
 DO;
 OUTPUT SPDS.ToAdd;
 end;
run;
```

If ClaimsAll were not a dynamic cluster table, the DATA step MERGE statement would create an implicit sort from the BY clause on the respective SPD Server data sets. However, ClaimsAll is a dynamic cluster table with member tables that are presorted. As a result, the dynamic cluster BY clause optimization uses BY clause processing to merge the sorted member tables instantaneously without using any SPD Server work-space or experiencing delays. The previous example merges the transaction data named NewOnes into new rows that will be appended to the data for the next quarter.

Consider that the member data sets ClaimsQ1 and ClaimsQ2 are indexed on the column Claim_Date:

```
DATA SPDS.RepClaims;
 SET SPDS.ClaimsAll;
 WHERE Claim_Date BETWEEN '01JAN2007' and '31MAR2007';
 BY PatID ClaimID;
run;
```

The WHERE clause determines whether each member table is true or false for each quarter. The WHERE clause is trivially true for the data set ClaimsQ1 because the WHERE clause is true for all dates in the first quarter. The WHERE clause is trivially false for the data set ClaimsQ2 because the WHERE clause is false for all dates in the second quarter. The BY clause optimization determines that the member table ClaimsQ1 will be processed because the WHERE clause is true for **all** of the rows of the ClaimsQ1 table. The BY clause optimization skips the member table ClaimsQ2 because the WHERE clause is false for **all** of the rows of the ClaimsQ2 table.

Suppose that the Claim_Date range is changed in the WHERE clause:

```
DATA SPDS.RepClaims;
 SET SPDS.ClaimsAll;
 WHERE Claim_Date BETWEEN '05JAN2007' and '28JUN2007';
 BY PatID ClaimID;
run;
```

When the new WHERE clause is evaluated, it is not trivially true for member tables ClaimsQ1 or ClaimsQ2. The WHERE clause is not trivially false for member tables ClaimsQ1 or ClaimsQ2 either. The WHERE clause calls dates that exist in portions of the member table ClaimsQ1, and it calls dates that exist in portions of the member table ClaimsQ2. The dates in the WHERE clause do not match **all** of the dates that exist in the member table ClaimsQ1, or **all** of the dates that exist in the member table ClaimsQ2. The dates in the WHERE clause are not totally exclusive of the dates that exist in the member tables ClaimsQ1 or ClaimsQ2. As a result, BY clause optimization will not be used when SPD Server runs the code.

Member Table Requirements for Creating Dynamic Cluster Tables

Overview of Table Requirements

When you create a dynamic cluster table, all of the member tables must have matching table, variable, and index attributes. If there are attribute mismatches, the dynamic cluster table creation fails, and SPD Server displays the following error message:

```
ERROR: Member table not compatible with other
        cluster members. Compare CONTENTS.
```

A more detailed error message is written to the SPD Server log. The SPD Server log lists which attribute is mismatched in the member tables. The following lists specify the member table attributes that must match for SPD Server to successfully create a dynamic cluster table.

- "Table Attributes " on page 82
- "Variable Attributes " on page 83
- "Index Attributes " on page 84

Table Attributes

The following table attributes must match in all member tables to successfully create a dynamic cluster table:

IDXSEGSIZE
: index segment size

OBSLEN
: observation record length

NVAR
: number of columns

NINDEXES
: number of indexes

DSORG
: data set organization

SEMTYPE
: data set semantic type

DSTYPE
: SAS data set type

LOCALE
: creation locale

LANG
: data set language tag

LTYPE
: data set language type tag

FLAGS
: compressed data set

: encrypted data set

: backup data set

: NLS variables in data set

: minmaxvarlist variables in data set

: SAS encryption password in data set

SASPW
: SAS encryption password

DS_ROLE
: data set option for ROLE

ENCODING_CEI
: encoding CEI for NLS (for compressed tables)

DISKCOMP
: compression algorithm

IOBLOCKSIZE
: I/O block size

IOBLOCKFACTOR
: I/O block factor

Variable Attributes

The following variable attributes must match in all member tables to successfully create a dynamic cluster table:

NAME	variable name
LABEL	variable label
NFORM	variable format
NIFORM	variable informat
NPOS	variable offset in record
NVARO	variable number in record
NLNG	variable length
NPREC	variable precision
FLAGS	NLS encoding supported minmaxvarlist variable
NFL	format length
NFD	format decimal places
NIFL	informat length
NIFD	informat precision
NSCALE	scale for fixed-point decimal
NTATTR	variable type attributes
TYPE	variable type
SUBTYPE	variable subtype
SORT	variable sorted status
NTYPE2	variable extended type code

Index Attributes

The following index attributes must match in all member tables to successfully create a dynamic cluster table:

NAME
index name

TYPE
index type

KEYFLAGS
unique index

nomiss index

LENGTH
index length

NVAR
number of variables in index

NVAR0
variable number in index

Querying and Reading Member Tables in a Dynamic Cluster

Dynamic clusters can be read using the MEMNUM= table option. The MEMNUM= option enables you to perform query or read operations on a single member table that belongs to a dynamic cluster. When you use the MEMNUM= option, SPD Server opens only the specified member table, instead of opening all of the member tables that belong to the cluster. You can determine the member number of a table in the cluster by issuing a **`cluster list`** statement or a **`proc contents`** command on the cluster. The SPD Server **`cluster undo`** or **`proc contents`** command output lists the member tables of the cluster in numbered order.

You can specify verbose output for the **`cluster list`** statement by using the following option syntax:

```
CLUSTER LIST clustername [/VERBOSE]
```

When you issue the /VERBOSE option with a **`cluster list`** statement, the output lists the MINMAXVARLIST information for each member table in a dynamic cluster.

The following example uses PROC SPDO to create a dynamic cluster that has a MINMAXVARLIST on the numeric column STORE_ID of each member table. Then a **`cluster list`** statement is issued using the /VERBOSE option. The **`cluster list`** output displays the dynamic cluster name, the names of each member table in the cluster, and the MINMAXVARLIST values for each member table.

```
PROC SPDO library=&libdom ;

CLUSTER CREATE ussales
  mem=ne_region
  mem=se_region
  mem=central_region
  maxslot=6 ;

CLUSTER LIST ussales/VERBOSE;
MINMAXVARLIST COUNT = 1
varname = store_id
Numeric type.

Cluster Name USSALES, Mem=NE_REGION
  Variable Name  (MIN,MAX)
  STORE_ID       ( 1, 20)

Cluster Name USSALES, Mem=SE_REGION
  Variable Name  (MIN,MAX)
  STORE_ID       ( 60, 70)

Cluster Name USSALES, Mem=CENTRAL_REGION
  Variable Name (MIN,MAX)
  STORE_ID      ( 60, 70)

NOTE: The maximum number of possible slots is 6.
```

You can specify an integer value n as an argument for the MEMNUM= table option to select the nth member of the table, or you can use the argument LASTCLUSTERMEMBER. When you use the LASTCLUSTERMEMBER argument with MEMNUM=, SPD Server selects the last member of the dynamic cluster table, without needing to count the members to determine the number (n) of the last member.

The following example uses the MEMNUM= table option to query against the member table sales200504 that belongs to the dynamic cluster table sales_history:

```
PROC SPDO library=&domain; ;
  CLUSTER CREATE sales_history
   mem=sales200501
   mem=sales200502
   mem=sales200503
   mem=sales200504
   mem=sales200505
   mem=sales200506
   maxslot=12 ;
  quit ;

 PROC PRINT data=&domain..sales_history (MEMNUM=4);
   WHERE salesdate = 30Apr2005;
 run;
```

To use the MEMNUM= table option to query the last member table in the dynamic cluster table sales200506, the query would be:

```
PROC SPDO library=&domain ;

  CLUSTER CREATE sales_history
   mem=sales200501
   mem=sales200502
   mem=sales200503
   mem=sales200504
   mem=sales200505
   mem=sales200506
   maxslot=12 ;
  quit ;

 PROC PRINT data&domain..sales_history
  (MEMNUM=LASTCLUSTERMEMBER);
   WHERE salesdate = 15Jun2005;
 run;
```

Unsupported Features in Dynamic Cluster Tables

Because of differences in the load and read structures for dynamic cluster tables, some standard features that are available in SAS tables and SPD Server tables are currently not supported in SPD Server 4.5. These features are:

- You cannot append data to a dynamic cluster table. To append data to a dynamic cluster table, the table must be unclustered, the data is appended to the individual unclustered files, and then the individual unclustered files must be reclustered.
- Record-level locking is not allowed.
- The SPD Server Backup and Restore utility is not available.
- Copying data with PROC COPY or PROC SQL is not supported.

If a task for a dynamic cluster table requires one of these features, you should undo the dynamic cluster table and create standard SPD Server tables.

Dynamic Cluster Table Examples

The following four examples show all of the fundamental operations that are required to use dynamic cluster tables:

Create a Dynamic Cluster Table Example

The following example creates a dynamic cluster table named Sales_History. The first part of the example generates dummy transaction data that is used in the rest of the example.

The example uses SPD Server tables from the domain motorcycle. Twelve individual SPD Server tables for monthly motorcycle sales during 2004 are bound into the dynamic cluster table named Sales_History. Tables are created for the first six months of motorcycle sales during 2005:

```
/* declare macro variables that will be used to  */
/* generate dummy transaction data               */
%macro var (varout,dist,card,seed,peak) ;
    %put &dist; &card; &seed; ;
    %local var1 ;

    if upcase("&dist;") = 'RANUNI'
    then do ;
       &varout; = int(ranuni(&seed;)*&card;)+1;
    end ;
    else
    if upcase("&dist;") = 'RANTRI'
    then do ;
       *%let vartri = %substr("&dist;",5,2)&card; ;
       *&varout; = int(rantri(&seed;,&peak;)*&card;)+1;
       &varout; = int(rantri(&seed;,&peak;)*&card;)+1;
    end ;
%mend ;

%macro linkvar (varin,varout,devisor) ;
    &varout; = int(&varin;/&devisor;) ;
%mend ;


/* declare main vars */
%let domain=motorcycle ;
%let host=kaboom ;
%let port=5200 ;
%let spdssize=256M ;
%let spdsiasy=YES ;

LIBNAME &domain; sasspds "&domain;"
   server=&host..;&port;
   user='anonymous'
   ip=YES ;


/* generate monthly sales data tables for */
/* 2004 and the first six months of 2005  */
data
   &domain..sales200401;
   &domain..sales200402;
   &domain..sales200403;
   &domain..sales200404;
   &domain..sales200405;
   &domain..sales200406;
   &domain..sales200407;
   &domain..sales200408;
   &domain..sales200409;
   &domain..sales200410;
   &domain..sales200411;
   &domain..sales200412;
   &domain..sales200501;
   &domain..sales200502;
   &domain..sales200503;
   &domain..sales200504;
   &domain..sales200505;
   &domain..sales200506;
   ;

drop seed bump1 bump2 random_dist ;

   seed = int(time()) ;


/* format the dummy transaction data */
```

```
format trandate shipdate paiddate yymmdd10. ;

put seed ;
   do transact = 1 to 5000 ;
      %var (customer,ranuni,100000,seed,1) ;

      %linkvar (customer,zipcode,10) ;
      %linkvar (customer,agent,20) ;
      %linkvar (customer,mktseg,10000) ;
      %linkvar (agent,state,100) ;
      %linkvar (agent,branch,25) ;
      %linkvar (state,region,10) ;

      %var (item_number,ranuni,15000,seed,1) ;

      %var (trandate,ranuni,577,seed,1) ;
      trandate = trandate + 16071 ;

      %var (bump1,ranuni,20,seed,.1) ;
      shipdate = trandate + bump1 ;

      %var (bump2,rantri,30,seed,.5) ;
      paiddate = trandate + bump2 ;

      %var (units,ranuni,100,seed,1) ;
      %var (trantype,ranuni,10,seed,1) ;
      %var (amount,rantri,50,seed,.5) ;
      amount = amount + 25 ;

      random_dist = ranuni ('03feb2005'd) ;


      /* sort the dummy transaction data into */
      /* monthly sales data tables            */

      if '01jan2004'd <= trandate <= '31jan2004'd
        then output &domain..sales200401; ;

      else if '01feb2004'd <= trandate <= '28feb2004'd
        then output &domain..sales200402; ;

      else if '01mar2004'd <= trandate <= '31mar2004'd
        then output &domain..sales200403; ;

      else if '01apr2004'd <= trandate <= '30apr2004'd
        then output &domain..sales200404; ;

      else if '01may2004'd <= trandate <= '31may2004'd
        then output &domain..sales200405; ;

      else if '01jun2004'd <= trandate <= '30jun2004'd
        then output &domain..sales200406; ;

      else if '01jul2004'd <= trandate <= '31jul2004'd
        then output &domain..sales200407; ;
      else if '01aug2004'd <= trandate <= '31aug2004'd
        then output &domain..sales200408; ;

      else if '01sep2004'd <= trandate <= '30sep2004'd
        then output &domain..sales200409; ;

      else if '01oct2004'd <= trandate <= '31oct2004'd
        then output &domain..sales200410; ;

      else if '01nov2004'd <= trandate <= '30nov2004'd
        then output &domain..sales200411; ;

      else if '01dec2004'd <= trandate <= '31dec2004'd
        then output &domain..sales200412; ;

      else if '01jan2005'd <= trandate <= '31jan2005'd
        then output &domain..sales200501; ;

      else if '01feb2005'd <= trandate <= '28feb2005'd
        then output &domain..sales200502; ;

      else if '01mar2005'd <= trandate <= '31mar2005'd
        then output &domain..sales200503; ;

      else if '01apr2005'd <= trandate <= '30apr2005'd
        then output &domain..sales200504; ;

      else if '01may2005'd <= trandate <= '31may2005'd
        then output &domain..sales200505; ;

      else if '01jun2005'd <= trandate <= '31jun2005'd
        then output &domain..sales200506; ;
      end ;
run ;


/* index the transaction data in the */
/* monthly sales data tables         */
%macro indexit (yrmth) ;
   PROC DATASETS library=&domain; nolist ;
      modify sales&yrmth; ;
      index create transact customer agent state branch trandate ;
   quit ;
%mend ;

%let spdsiasy=YES ;

%indexit (200401) ;
%indexit (200402) ;
%indexit (200403) ;
%indexit (200404) ;
%indexit (200405) ;
%indexit (200406) ;
%indexit (200407) ;
%indexit (200408) ;
%indexit (200409) ;
```

```
%indexit (200410) ;
%indexit (200411) ;
%indexit (200412) ;
%indexit (200501) ;
%indexit (200502) ;
%indexit (200503) ;
%indexit (200504) ;
%indexit (200505) ;
%indexit (200506) ;


/* Use PROC SPDO to create the dynamic cluster */
/* table sales_history                         */
PROC SPDO library=&domain; ;
   cluster create sales_history
      mem=sales200401
      mem=sales200402
      mem=sales200403
      mem=sales200404
      mem=sales200405
      mem=sales200406
      mem=sales200407
      mem=sales200408
      mem=sales200409
      mem=sales200410
      mem=sales200411
      mem=sales200412
   maxslot=36 ;
quit ;
```

Add Tables to a Dynamic Cluster Example

The following example adds member tables to the dynamic cluster table named Sales_History. The Sales_History table currently contains 12 members. Each member is an SPD Server table that contains monthly sales data. This example augments the 12 member tables for 2004 with 6 new member tables that contain sales data for January through June of 2005:

```
/* declare main vars */
%let domain=motorcycle ;
%let host=kaboom ;
%let port=5200 ;
%let spdssize=256M ;
%let spdsiasy=YES ;

LIBNAME  &domain; sasspds &domain;
   server=&host..;&port;
   user='anonymous'
   ip=YES ;

/* Use PROC SPDO to add member tables to */
/* the dynamic cluster table sales_history */

PROC SPDO library=&domain;
```

```
   cluster add sales_history
   mem=sales200501
   mem=sales200502
   mem=sales200503
   mem=sales200504
   mem=sales200505
   mem=sales200506;
quit ;

/* Verify the presence of the added tables */
PROC CONTENTS data=&domain..sales_history;
run ;
```

Undo Dynamic Cluster Table Example

The undo example is included as part of the following refresh example.

Refresh Dynamic Cluster Table Example

Refreshing SPD Server dynamic cluster tables is a combination of two tasks, UNDO CLUSTER and CREATE CLUSTER. The UNDO CLUSTER command unbinds an existing dynamic cluster table. The CREATE CLUSTER command rebinds the dynamic cluster table with updated member tables. Therefore, the following example shows both the UNDO CLUSTER and CREATE CLUSTER commands with SPD Server dynamic cluster tables.

The following example refreshes the dynamic cluster table named Sales_History. The Sales_History table received additional member tables in the previous example. The 18-member dynamic cluster table Sales_History is unbound. The 12 member tables that contain 2004 sales data are deleted when the dynamic cluster table Sales_History is recreated with only the six member tables that contain 2005 sales data. The combined actions refresh the contents of the dynamic cluster table Sales_History.

```
/* declare main vars */
%let domain=motorcycle ;
%let host=kaboom ;
%let port=5200 ;
%let spdssize=256M ;
%let spdsiasy=YES ;

LIBNAME &domain; sasspds &domain;
   server=&host..;&port;
   user='anonymous'
   IP=YES ;

/* Use PROC SPDO to undo the existing dynamic */
/* cluster table Sales_History, then rebind */
/* it with members from months in 2005 only */

PROC SPDO library=&domain;
   cluster undo sales_history ;
   cluster create sales_history
      mem=sales200501
      mem=sales200502
      mem=sales200503
      mem=sales200504
      mem=sales200505
      mem=sales200506
   maxslot=36 ;
quit ;

/* Verify the contents of the refreshed dynamic */
/* cluster table sales_history */

PROC CONTENTS data=&domain..sales_history;
run ;
```

Part 3

SPD Server SQL Features

Chapter 8
SPD Server SQL Features

SPD Server SQL Planner

SPD Server includes SQL Planner optimizations. SQL Planner optimizations improve the performance of the more frequent query types that used in data mining solutions such as Enterprise Marketing Automation. A key enhancement to the SPD Server SQL Planner is optimizing correlated queries through the use of query rewrite techniques. Correlated queries are common in business and analytic intelligence data mining. Another significant enhancement is the tighter integration of the Parallel Group-By technology in the planner. The tighter integration adds performance benefits to nested Group-By syntax.

Connecting to the SPD Server SQL Engine

Implicit Pass-Through Connection

You can use an implicit pass-through connection to pass implicit SQL statements to the SPD Server SQL Engine. When you use an implicit pass-through connection, the SAS SQL planner parses SQL statements to determine which, if any, portions can be passed to the SPD Server SQL Engine. In order for a submitted SQL statement to take advantage of implicit pass-through SQL, the tables that are referenced in the SQL statement must be SPD Server tables, and the SPD Server SQL engine must be able to successfully parse the submitted SQL statement.

An example of an SPD Server implicit pass-through connection is available in the Help section in this document on how to “Specify SQL Options Using Implicit Pass-Through Code” on page 100.

Explicit Pass-Through Connection

You can use an explicit pass-through connection to pass explicit SQL statements to the SPD Server SQL Engine. When you use an explicit pass-through connection, you decide explicitly which SQL statements are passed to the SPD Server SQL Engine. The explicit pass-through connection passes the entire SQL statement as written to the SPD Server SQL Engine, which parses and plans the SQL statement. All tables that are referenced in the SQL statement must be SPD Server tables or an error occurs.

An example of an SPD Server implicit pass-through connection is available in “Specify SQL Options Using Explicit Pass-Through Code” on page 100.

LIBNAME Syntax to Specify a Libref

Below is a LIBNAME statement that associates a libref, the SASSPDS engine, and an SPD Server domain.

```
LIBNAME libref
SASSPDS <'SAS-data-library'> <SPD Server-options>;
```

Use the following arguments:

libref
: a name that is up to eight characters long and that conforms to the rules for SAS names.

SASSPDS
: the name of the SPD Server engine.

'*SAS-data-library*'
: the logical LIBNAME domain name for an SPD Server data library on the host machine. The Name Server resolves the domain name into the physical path for the library.

SPD Server-options
: one or more SPD Server options.

Libref Statements

Whenever you issue a CONNECT statement to an SPD Server SQL server with the DBQ option, by default you define a primary LIBNAME domain. The software uses the primary domain to resolve table references in SQL statements executed for that connection.

You can also use the libref statement to assign secondary LIBNAME domains for the SPD Server SQL Server. The additional libref statements assign explicit LIBNAME domains, allowing the software to specify two-part table names for SQL statements executed for the connection.

```
PROC SQL;
 execute(libref librefname
 <enginename>
  engopt= ' ')
by sasspds;
```

Libref Clauses

The ENGNAME Clause

Specifies the name of an alternate SAS I/O engine to service the libref's access to data. If you do not specify an alternate SAS I/O engine, the default is **spdseng**, which accesses SPD Server tables.

The ENGOPT Clause

Specifies options that configure the libref to access a specific data source or storage domain. Use single or double quotes around the clause. (If you have nested quotes within a clause, alternate between single- and double-quoted expressions.) The available options depend on the current value of the ENGNAME option. For the default **spdseng**, you can specify any SPD Server CONNECT or LIBNAME engine option with the exception of **prompt**, **newpasswd**, and **chngpass**. Use the same keyword or value syntax required by the CONNECT statement.

Note: If you specify the SAS I/O engine **spdseng** and use explicit options in your CONNECT statement, these options become default ENGOPT clause options. Explicit options can also be specified using the ENGOPT clause. Explicit options specified in an ENGOPT clause override default values or declarations made in previous CONNECT statements.

Libref Examples

Libref for Another Domain but the Same CONNECT Statement User

In this example the client connects to the SPD Server SQL server using the engine **sasspds.** The domain is **mydomain**, the server machine is called **namesvrID**, and the port number is **namesvrPortNum**. The execute statement assigns the libref cookie to another domain, dough. After the libref is executed, the user issuing the connect statement can now access either the default domain **mydomain** or the secondary domain **dough**.

```
PROC SQL;
connect to sasspds
 (dbq='mydomain'
  host='namesvrID'
  serv='namesvrPortNum'
  user='neraksr'
  passwd='siuya');
execute(libref cookie
 engopt='
 dbq="dough"')
by sasspds;
```

In the example above, the libref is **cookie**, and the secondary domain named is **dough**. The intent of the example is to show how the CONNECT and libref statements work in conjunction to access multiple domains for the same user.

Libref to Same Domain but Different CONNECT Statement User

This example assigns a libref to the domain specified by the CONNECT statement but for another user (different SPD Server User ID).

```
PROC SQL;
execute(libref samslib
 engopt='
 user="sam"
 passwd="samspwd"')
by sasspds;
```

Secondary Libref Using a Different Host

This example assigns a secondary libref to a different host machine.

```
PROC SQL;
execute(libref sam2
 engopt='
 host="flex"
 dbq="samsplace"')
by sasspds;
```

Specifying SPD Server SQL Planner Options

The SPD Server SQL Planner provides reset options that you can use to configure the behavior of the SQL Planner and the SPD Server facilities that function through the SQL Planner, such as the SPD Server Parallel Group-By facility, the SPD Server Parallel Join facility, and the SPD Server STARJOIN facility. You can specify SPD Server SQL reset options using either explicit pass-through or implicit pass-through code.

Specify SQL Options Using Explicit Pass-Through Code

The example below shows how to use an **execute(reset <reset-options>)** statement in explicit SPD Server pass-through SQL code to invoke an SQL Planner, Parallel Group-By facility, Parallel Join facility, or STARJOIN facility reset option.

Most SQL Planner reset option usage examples in this document use explicit pass-through code. See the implicit pass-through code example below to see how SQL reset options can be declared using an implicit **%let spdssqlr=** statement instead of an explicit **execute(reset <reset-options>)** statement.

```
/* Explicit Pass-Through SQL Example */
/* to invoke an SQL Reset Option */

PROC SQL ;

connect to sasspds (
  dbq=domain-name
  server=<host-name>.<port-number>
  user='username') ;

execute(reset <reset-options>)
  by sasspds ;

execute(SQL statements)
  by sasspds ;

disconnect from sasspds ;
 quit ;
```

Specify SQL Options Using Implicit Pass-Through Code

The example below shows how to use a **%let spdssqlr=<reset-options>** statement in implicit SPD Server pass-through SQL code to invoke an SQL Planner, Parallel Group-By facility, Parallel Join facility, or STARJOIN facility reset option.

Most SQL Planner reset option usage examples in this document use explicit pass-through code. The implicit pass-through code example below shows how SQL reset options can be declared using an implicit **%let spdssqlr=** statement instead of an explicit **execute(reset <reset-options>)** statement.

```
/* Implicit Pass-Through SQL Example */
/* to invoke an SQL Reset Option */
```

```
%let spdssqlr=<reset-options> ;

PROC SQL ;
SQL statements ;

quit ;
```

Important SPD Server SQL Planner Options

_Method

The SQL **_method** option is one of the most important reset options. The _method reset option provides a method tree in the output that shows how the SQL was executed.

The following methods are displayed in the SQL _method tree:

sqxcrta
: Create table as Select.

sqxslct
: Select rows from table.

sqxjsl
: Step Loop Join (Cartesian Join).

sqxjm
: Merge Join execution.

sqxjndx
: Index Join execution.

sqxjhsh
: Hash Join execution.

sqxsort
: Sort table or rows.

sqxsrc
: Read rows from source.

sqxfil
: Filter rows from table.

sqxsumg
: Summary Statistics (with GROUP BY).

sqxsumn
: Summary Statistics (not grouped).

sqxuniq
: Distinct rows only.

sqxstj
: STARJOIN

sqxxpgb
: Parallel Group-By

sqxxpjn
: Parallel Join with Group-By. The SAS log displays the name of the parallel join method that was used.

sqxpll
: Parallel Join without Group-By

Reading the Method Tree

A method tree is produced in your output when the **_method**reset option is specified for the SQL Planner. The SQL Planner method tree is read from bottom row to top row. Below is an example that shows how to interpret the method tree by substituting the type of method that was used in each step.

```
PROC SQL ;
create table tbl1 as
 select *
  from path1.dansjunk1 a,
   path1.dansjunk2 b,
   path1.dansjunk3 c
  where a.i = b.i
   and a.i = c.i ;
quit ;
```

Here is the example Method Tree that was printed:

```
SPDS_NOTE: SQL execution methods chosen are:
<0x00000001006BBD78> sqxslct
<0x00000001006BBBF8>     sqxjm
<0x00000001006BBB38>               sqxsort
<0x0000000100691058>                   sqxsrc
<0x0000000100667280>               sqxjm
<0x0000000100666C50>                       sqxsort
<0x0000000100690BD8>                           sqxsrc
<0x00000001006AE600>                       sqxsort
<0x0000000100694748>                           sqxsrc
```

Reading from bottom to top, you can review the sequence of methods that were invoked.

```
SPDS_NOTE: SQL execution methods chosen are:
<0x00000001006BBD78> step-9
<0x00000001006BBBF8>     step-8
<0x00000001006BBB38>               step7
<0x0000000100691058>                    step-6
<0x0000000100667280>                 step-5
<0x0000000100666C50>                        step-4
<0x0000000100690BD8>                            step-3
<0x00000001006AE600>                       step-2
<0x0000000100694748>                              step-1
```

In step 1, **sqxsrc** reads rows from the source. In step 2, **sqxsort** sorts the table rows. Then in steps 3 and 4, more rows are read and sorted. In step 5, the tables are joined by **sqxjm**, and so on.

BUFFERSIZE=

The SPD Server query optimizer considers a hash join when an index join is eliminated as a possibility. With a hash join, the smaller table is reconfigured in memory as a hash table. SQL sequentially scans the larger table and row-by-row performs a hash lookup against the small table to form the result set. On a memory-rich system, consider increasing the **BUFFERSIZE=** option to increase the likelihood that a hash join is chosen. The default **BUFFERSIZE=** setting is 64K. You can specify the amount of memory that you want SPD Server to use for hash joins.

Usage:

```
/* Increase buffersize from 64K   */
execute(reset buffersize=1048576)
  by sasspds ;
```

EXEC/NOEXEC

You use the SPD Server SQL Planner **EXEC/NOEXEC** option to turn SPD Server SQL execution on or off.

Usage:

```
/* This explicit Pass-Through SQL */
/* prints the method tree without */
/* executing the SQL code. */

PROC SQL ;
connect to sasspds
  (dbq=domain
   server=<host-name>.<port-number>
   user='username') ;

execute (reset _method noexec)
 by sasspds ; /* turns SQL exec off */

execute (SQL statements)
 by sasspds ;

disconnect from sasspds ;
quit ;
```

INDEXSELECTIVITY=

The **INDEXSELECTIVITY=** option enables you to tune the SQL join planner strategy for more efficient or robust index join methods. The **INDEXSELECTIVITY=** setting is a continuous value between 0 and 1 that acts as a minimum threshold value for the SPD Server duplicity ratio when selecting a join method. The SPD Server duplicity ratio is a heuristic that acts as a measure of the cardinality of the inner table index, relative to the frequency of index values as they occur in the outer table. Both **INDEXSELECTIVITY=** and the SPD Server duplicity ratio are continuous values between 0 and 1. SPD Server compares the calculated duplicity ratio for an SPD Server index join to the value that is specified in the **INDEXSELECTIVITY=** option. If the calculated duplicity ratio is greater than or equal to the value that is specified in the **INDEXSELECTIVITY=** option, the index join method is chosen. The default setting for the **INDEXSELECTIVITY=** option is 0.7.

How is the SPD Server duplicity ratio calculated? The duplicity ratio of an indexed column is calculated as the number of unique values in the index column, divided by the number of rows in the outer table. As the value of the duplicity ratio approaches 0, indicating low cardinality, the greater the number of duplicate values that exist in the rows of the outer table. As the value of the duplicity ratio approaches 1, indicating high cardinality, the fewer the number of duplicate index values in the rows of the outer table. For example, a duplicity ratio of 1/1, or 1, represents a unique index value for every row in the outer table, a unique index. A duplicity ratio value of 1/2, or 0.5, represents a unique index value for every two rows in the outer table. A duplicity ratio value of 1/4, or 0.25, represents a unique index value for every four rows in the outer table. The default setting of **INDEXSELECTIVITY=** is 0.7, representing a unique index value for every 1.43 rows in the outer table.

For example, consider an outer table that contains 100 rows that match join key values in the inner table, and a calculated SPD Server duplicity ratio of 0.7 (a unique index value per 1.43 rows in the outer table,) the expected result set would be 100*1.43, or 143 rows.

From an efficiency perspective, higher cardinality and index duplicity ratios are considered better for an index join. Duplicity ratios near 1 mean more efficient processing during probes between the outer table rows and the inner table index, because each probe has fewer rows to retrieve. This in turn minimizes the work the SPD Server index must do to find and retrieve the matching rows during the join operation, resulting in an optimized index join.

You can use **INDEXSELECTIVITY=** to configure the index join to be more or less tightly constrained by the number of duplicate values in the join table rows. Increasing the value of **INDEXSELECTIVITY=** makes the duplicity criteria more selective by decreasing the allowable average number of rows per probe of the inner table. Setting **INDEXSELECTIVITY=** equal to 1.0 allows a join with a unique index only. Setting **INDEXSELECTIVITY=** to a value greater than 1.0 allows no index joins. Decreasing the value of **INDEXSELECTIVITY=** makes the duplicity criteria more forgiving by increasing the allowable average number of rows per probe of the inner table. Setting **INDEXSELECTIVITY=** equal to 0.0 allows joins with any amount of duplicity.

Usage:

```
execute(reset indexselectivity=<0.0 ... 1.0>)
  by sasspds ;
```

INOBS

Use the **INOBS** option to specify the specific number of observations that you want to read from input tables.

Usage:

```
execute(reset inobs=<n>)
  by sasspds ;
```

where the integer value <*n*> is the desired number of observations.

MAGIC

You use the SPD Server SQL Planner **MAGIC** reset option that controls how the SPD Server SQL planner executes join statements. The Magic option has three settings, 101, 102, and 103.

Usage:

```
execute(reset magic=<101/102/103>)
  by sasspds ;
```

MAGIC=101
: SPD Server performs sequential loop joins. Sequential loop joins are brute force joins that match every row from the first table to every row of the second table.

MAGIC=102
: SPD Server performs sort merge joins. Sort merge joins force a sort on all tables that are involved in the join.

MAGIC=103
: SPD Server performs hash joins. Hash joins require SPD Server to create a memory table in order to perform the join. The size of the memory table is limited based on memory available.

OUTOBS

Use the **OUTOBS** option to specify the specific number of observations that you want to create or print in your output.

Usage:

```
execute(reset outobs=<n>)
  by sasspds ;
```

where the integer value <**n**> is the desired number of observations.

OUTRSRTJNDX/NOOUTRSRTJNDX

Use the **OUTRSRTJNDX/NOOUTRSRTJNDX** option to configure sort behavior for an SPD Server join index. **OUTRSRTJNDX** sorts the outer table for a join index by the join key. This is the default SPD Server setting. **NOOUTRSRTJNDX** does not sort the outer table for a join index.

Usage:

```
/* Disable outer table      */
/* sorting for a join index */
execute(reset nooutrsrtjndx)
  by sasspds ;


/* Enable outer table       */
/* sorting for a join index */
execute(reset outrsrtjndx)
  by sasspds ;
```

PRINTLOG/NOPRINTLOG

You use the PRINTLOG/NOPRINTLOG option of the SPD Server SQL Planner to turn the printing of the SQL statement text to the SPD Server log on or off.

Usage:

```
PROC SQL ;
connect to sasspds
    (dbq=domain
     server=<host-name>.<port-number>
     user='username') ;

/* turn SQL statement printing on */
execute (reset printlog)
by sasspds ;

/* all statements will be printed to SPD Server log */
execute (SQL statements)
by sasspds ;

/* turn SQL statement printing off */
execute (reset noprintlog)
by sasspds ;

disconnect from sasspds ;
quit ;
```

SASVIEW/NOSASVIEW

Use the **SASVIEW/NOSASVIEW** option to enable or disable SAS PROC SQL views that use an SPD Server LIBNAME. SAS PROC SQL views use a generic transport format to represent numeric values, which SPD Server converts to native numeric values. When extremely large or extremely small numeric values are conveyed in a SAS PROC SQL view to SPD Server, some precision might be lost in extreme values during the SPD Server numeric conversion.

Usage:

```
/* Disable SAS PROC SQL views     */
/* that use an SPD Server LIBNAME */
execute(reset nosasview)
  by sasspds ;

/* Enable SAS PROC SQL views that */
/* use an SPD Server LIBNAME      */
execute(reset sasview)
  by sasspds ;
```

If SAS PROC SQL views are disabled and SPD Server pass-through SQL uses a view that was created by PROC SQL, SPD Server rejects the PROC SQL statement and inserts the following error message in the SAS log::

```
SPDS_WARNING: SAS View and SASVIEW Reset Option equals No.
SPDS_ERROR: An error has occured.
```

If SAS PROC SQL views are enabled and SPD Server pass-through SQL uses a view that was created by PROC SQL, SPD Server prints the following note in the SAS log:

```
SPDS_NOTE: SPDS using SAS View in transport mode.
```

SQLHIMEM

Use the **SQLHIMEM** option to dynamically allocate or deallocate large memory blocks to and from the SPD Server SQL proxy address space. Providing the SPD Server SQL proxy address space with large memory blocks enhances processing during memory-intensive operations such as large table sorts and joins. By deallocating the large memory blocks after the memory overhead is no longer needed, memory resources for the SPD Server process resident set size, the SPD Server SQL proxy address space, and the SPD Server swap space are freed, which allows the resources to be used elsewhere by the system. If the **SQLHIMEM** option is not specified, system memory calls malloc() and free() are used to allocate and deallocate memory from the process heap. The trade-off with using **SQLHIMEM** instead of malloc() and free() is the need to use **SQLHIMEM** to allocate and deallocate memory for each SQL operation for a given user, as opposed to acquiring the SQL proxy address space once via malloc() and retaining the use of the memory resources.

Usage:

```
execute(reset SQLHIMEM) by sasspds;
```

By default, SPD Server does not use **SQLHIMEM**.

UNDO_POLICY=

Use the **UNDO_POLICY** option in SPD Server PROC SQL and RESET statements to configure SPD Server PROC SQL error recovery. When you update or insert rows in a table, you might receive an error message that states that the update or insert operation cannot be performed. The **UNDO_POLICY** option specifies how you want SPD Server to handle rows that were affected by INSERT or UPDATE statements that preceded a processing error.

Usage:

```
/* Do not undo any updates or inserts */
execute(reset undo_policy=none)
  by sasspds ;

/* Permit row inserts and updates to  */
/* be done up to the point of error   */
execute(reset undo_policy=required)
  by sasspds ;
```

UNDO_POLICY=NONE
    is the default setting for SPD Server. It does not undo any updates or inserts.

UNDO_POLICY=REQUIRED
    undoes all row updates or inserts up to the point of error.

UNDO_POLICY=OPTIONAL
Undoes any updates or inserts that it can undo reliably.

If the UNDO policy is not REQUIRED, you get the following warning message for an insert into the table:

```
WARNING: The SQL option UNDO_POLICY=REQUIRED is not in effect. If an
error is detected when processing this insert statement, that error
will not cause the entire statement to fail.
```

Additional SQL Reset Options

More detailed information about the available SQL reset options for the SPD Server SQL Parallel Join, Parallel Group-By, STARJOIN, and Correlated Query facilities can be found in this document as follows:

- "Parallel Join SQL Options " on page 110
- "Parallel Group-By SQL Options " on page 116
- "STARJOIN Options " on page 117
- "Correlated Query Options " on page 133

Parallel Join Facility

Overview of the Parallel Join Facility

The Parallel Join facility is a feature of the SPD Server SQL planner that decreases the required processing time when creating a pair-wise join between two SPD Server tables. The processing time savings is created when SPD Server performs the pair-wise join in parallel.

The SQL planner first searches for pairs when SPD Server source tables are to be joined. When a pair is found, the planner checks the join syntax for that pair to determine whether it meets all of the requirements for the Parallel Join facility. If the join syntax meets the requirements, the pair of tables will be joined by the Parallel Join facility.

Criteria to use the Parallel Join Facility

The criteria to use the SPD Server Parallel Join facility can be more complex than simply requiring a pair-wise join of two SPD Server tables. The Parallel Join facility can handle multiple character columns, numeric columns, or combinations of character and numeric columns that are joined between pairs of tables. Numeric columns do not need to be of the same width to act as a join key, but character columns must be of the same width in order to be a join key. Columns that are involved in a join cannot be derived from a SAS CASE statement, and cannot be created from character manipulation functions such as SUBSTR, YEAR, MONT, DAY, and TRIM.

Parallel Join Methods

Parallel Sort-Merge Method

The parallel sort-merge join method first performs a parallel sort to order the data, and then merges the sorted tables in parallel. During the merge, the facility concurrently joins multiple rows from one table with the corresponding rows in the other table. You can use the parallel sort-merge join method to execute any join that meets the requirements for parallel join.

The parallel sort-merge method is a good all-around parallel join strategy that requires no intervention from the user. The tables for the sort-merge method do not need to be in the same domain. The performance for the sort-merge method is not affected by the distribution of the data in the sort key columns.

The sort-merge method begins by completely sorting the smaller of the two tables being joined, while also performing concurrent partial parallel sorts on the larger table. If both tables are very large and sufficient resources are not available to do the complete sort on the smaller table, the performance of the parallel sort-merge method can degrade. The parallel sort-merge method is also limited when performing an outer, left, or right join in parallel. Only two concurrent threads can be used when performing parallel outer, left, or right joins. Inner joins are not limited in the parallel sort-merge method and can use more than two concurrent threads during parallel operations.

Parallel Range Join Method

The parallel range join method uses a join index to determine the ranges of rows between the tables that can be joined in parallel. The parallel range join method requires you to create a join index on the columns to be joined in the tables that you want to merge. The join index divides the two tables into a specified number of near-equal parts, or ranges, based on matching values between the join columns. The Parallel Join facility recognizes the ranges of rows that contain matching values between the join columns, and then uses concurrent join threads to join the rows in parallel. The SPD Server parallel sort then sorts the rows within a range.

The parallel range join method can be performed only on tables that are in the same domain. If either of the two tables are updated after the join index is created, the join index must be rebuilt before the parallel range join method can be used. The parallel range join method performs best when the columns of the tables that are being joined are sorted. If the columns are not relatively sorted, then the concurrent join threads can cause processor thrashing. Processor thrashing occurs when unsorted rows in a table require SPD Server to perform increasingly larger table row scans, which can consume processor resources at a high rate during concurrent join operations.

More detailed information about creating join indexes is available in Chapter 17, "SAS Scalable Performance Data (SPD) Server Index Utility Ixutil," of the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.

How does the SPD Server Parallel Join facility choose between the sort-merge method and the range join method? If a join index is available for the tables to be joined, the Parallel Join facility chooses the parallel range join method. If a join index does not exist, or if the join index has not been rebuilt since a table was updated, the Parallel Join facility defaults to using the parallel sort-merge method.

Parallel Joins with Group-By

A powerful feature of the SPD Server Parallel Join facility is its integration with the SPD Server Parallel Group-By facility. If the result of the parallel join contains a group-by statement, the partial results of the parallel join threads are passed to the Parallel Group-By facility, which performs the group-by operation in parallel. In the following example, SPD Server performs both a parallel join and parallel group-by operation.

```
LIBNAME path1 sasspds .... IP=YES;

PROC SQL;
create table junk as
 select a.c, b.d, sum(b.e)
 from path1.table1 a,
  path1.table2 b
 where a.i = b.i
 group by a.d, b.d;
quit;
```

When you use the SPD Server Parallel Join facility, you are not restricted to using the parallel group-by method only on single tables.

Parallel Join SQL Options

PLLJOIN/NOPLLJOIN

The **PLLJOIN/NOPLLJOIN** option enables and disables the SPD Server Parallel Join facility.

Usage:

```
execute(reset noplljoin)
  by sasspds ; /* disables Parallel Join */
```

CONCURRENCY

The **CONCURRENCY=<n>** option sets the concurrency level that is used by the SPD Server Parallel Join facility, where the integer n specifies the number of levels. In most cases, changing the default SPD Server concurrency setting (half of the available number of processors) is not recommended.

Usage:

```
execute(reset concurrency=4)
  by sasspds ; /* enables 4 concurrency levels */
```

PLLJMAGIC

The **PLLJMAGIC** option specifies how SPD server performs parallel joins.

Usage:

```
execute(reset plljmagic=<100/200>)
  by sasspds ;
```

PLLJMAGIC=100 forces a parallel range join when the range index is available.

PLLJMAGIC=200 forces a parallel merge join.

Parallel Join Example 1

The first parallel join example is a basic SQL query that creates a pair-wise join of two SPD Server tables, **table1** and **table2**.

```
LIBNAME path1 sasspds .... IP=YES;

PROC SQL;
create table junk as
 select *
  from path1.table1 a,
  path1.table2 b
  where a.i = b.i;
 quit;
```

Parallel Join Example 2

The next parallel join example is an SQL query that uses more than two SPD Server tables. In this example, the SQL planner performs a parallel join on **table1** and **table2**, and then use a non-parallel method to join the results of the first join and **table3**. A non-parallel join method is used for the second join, because the criteria for a parallel join was not met. A parallel join can be performed only on a pair-wise join of two SPD Server tables, and the query calls three SPD Server tables.

```
LIBNAME path1 sasspds .... IP=YES;

PROC SQL;
create table junk as
 select *
  from path1.table1 a,
  path1.table2 b,
  path1.table3 c
 where a.i = b.i and b.i = c.i;
quit;
```

Parallel Join Example 3

Multiple parallel joins can be used in the same SQL query, as long as the SQL planner can perform the query using more than one pairwise join. In the next parallel join example, a more complex query contains a union of two separate joins. Both joins are pair-wise joins of two SPD Server tables. There is a pair-wise join between **table1** and **table2**, and then a pair-wise join between **table3** and **table4** is performed concurrently, using the Parallel Join facility.

```
PROC SQL;
create table junk as
 select *
  from path1.table1 a,
  path1.table2 b
  where a.i = b.i
  union

 select *
  from path1.table3 c,
path1.table4 d
where c.i = d.i;
quit;
```

Parallel Group-By Facility

Overview of the Parallel Group-By Facility

SPD Server SQL Planner optimizations improve the performance of the more frequent query types used in data mining solutions. One of the SQL planner optimizations integrated into SPD Server is tighter integration of the Parallel Group-By capability. Parallel Group-By is a high performance parallel summarization of data executed using SQL. Parallel Group-By is often used in SQL queries (through the use of sub queries) to apply selection lists for inclusion or exclusion. The tighter integration adds performance benefits to nested Group-By syntax.

Parallel Group-By looks for specific patterns in a query that can be performed using parallel processing summarization. Parallel Group-By works against single tables that are used to aggregate data. Parallel processing summarization is limited to the types of functions it can handle.

The Parallel Group-By support in SPD Server has been expanded in many areas. Parallel Group-By is integrated into the WHERE clause planner code so that it will boost the capabilities of the SPD Server SQL engine. Any section of code that matches the Parallel Group-By trigger pattern will use it.

Enhanced Group-By Functions

Parallel Group-By now supports the following functions in syntax: COUNT, FREQ, N, USS, CSS, AVG, MEAN, MAX, MIN, NMISS, RANGE, STD, STDERR, SUM, VAR. These functions all can accept the DISTINCT term. The listed functions are the minimum summary functions that are required in order to support the SAS Enterprise Marketing Automation tool suite.

Table Aliases Supported

Table aliases are now supported in SPD Server in order to better support front end tools such as SAS Enterprise Marketing Automation. Tools such as SAS Enterprise Marketing Automation generate SQL queries that use table aliases. Table aliases enable both shorter coding syntax and a method to select a specific column in a query that has two tables that share common column names.

Nested Queries Meet Group-By Syntax Requirements

Since the Parallel Group-By functionality is integrated into the SPD Server WHERE clause planner, now many sections of queries can take advantage of performance enhancements such as parallel processing. Some common performance enhancements are sub-queries that generate value lists in an IN clause, views that now conform to Parallel Group-By syntax, and views that contain nested Group-By syntax.

General Syntax:

SELECT '*project-list*' **FROM** '*table-name*' ;

WHERE [*where-expression*];

GROUP BY [*groupby-list*];

HAVING [*having-expression*];

ORDER BY [*orderby-list*];

project-list
: Items must be either column names (which must appear in the groupby-list) or aggregate (summary) functions involving a single column (with the exception of count(*), which accepts an asterisk argument. At least one aggregate function must be specified. Project items can be aliased (for example, select avg(salary) as avgsal from) and these aliases can appear in any *where-expression*, *having-expression*, *groupby-list* or *orderby-list*. The following aggregate functions are supported: count, avg, avg distinct, count distinct, css, max, min, nmiss, sum, sum distinct, supportc, range, std, stderr, uss, var. Mean is a synonym for avg. Freq and n are synonyms for count except they do not accept the asterisk argument.

table-name
: Table names can be one- or two-part identifiers (for example, mytable or foo.mytable), the latter requiring a previous libref statement to define the domain identifier (for example, foo).

The *where-expression* is optional.

The optional *groupby-list* must be column names or projected aliases.

The optional *having-expression* must be a Boolean expression composed of aggregate functions, GROUP BY columns, or constants.

The optional *orderby-list* must be projected column names or aliases or numbers that represent the position of a projected item (for example, select a, count (*) order by 2).

Since the Parallel Group-By functionality is integrated into the SPD Server WHERE clause planner, now many sections of queries can take advantage of performance enhancements such as parallel processing. Some common performance enhancements are sub-queries that generate value lists in an IN clause, views that now conform to Parallel Group-By syntax, and views that contain nested Group-By syntax.

Formatted Parallel Group Select

By default, the columns of a group-by statement are grouped by their unformatted value. SQL pass-through parallel GROUP BY provides the capability to also group data by the columns output data format. For example, you could group by the date column of a table with an input format of mmddyy8 and an output format of monname9. Suppose the column has dates 01/01/04 and 01/02/04. Grouping by the unformatted value would put these dates into two separate groups. However, grouping by the formatted month name, would put these values into the same month grouping of January.

You enable or disable pass-through formatted parallel GROUP BY with the following execute commands:

```
PROC SQL;
connect to sasspds
```

```
  (dbq=.......);

 /* turn on formatted parallel group-by */
 execute(reset fmtgrpsel)
  by sasspds;

 select *
 from connection
 to sasspds
  (select dte
   from mytable
   groupby dte);

 /* turn off formatted parallel group-by */
 execute(reset nofmtgrpsel)
  by sasspds;

 select *
 from connection
 to sasspds
  (select dte
   from mytable
   groupby dte);

 quit;
```

The example code below is extracted from a larger block of code, whose purpose is to make computations based on user-defined classes of age, such as Child, Adolescent, Adult, and Pensioner. The code uses SQL Parallel Group-By features to create the user-defined classes and then uses them to perform aggregate summaries and calculations.

```
/* Use the parallel group-by feature with the   */
/* fmtgrpsel option. This groups the data based */
/* on the output format specified in the table. */
/* This will be executed in parallel.           */

PROC SQL;
connect to sasspds
 (dbq="&domain"
  serv="&serv"
  host="&host"
  user="anonymous");

 /* Explicitly set the fmtgrpsel option */

 execute(reset fmtgrpsel)
  by sasspds;

 title 'Simple Fmtgrpsel Example';
 select *
 from connection to sasspds
  (select age, count(*) as count
   from fmttest group by age);

 disconnect from sasspds;
 quit;
```

```
PROC SQL;
connect to sasspds
 (dbq="&domain"
  serv="&serv"
  host="&host"
  user="anonymous");

 /* Explicitly set the fmtgrpsel option */

 execute(reset fmtgrpsel)
  by sasspds;

 title 'Format Both Columns Group Select Example';

 select *
 from connection to sasspds
  (select
    GENDER format=$GENDER.,
    AGE format=AGEGRP.,
   count(*) as count
   from fmttest
   formatted group by GENDER, AGE);

 disconnect from sasspds;

 quit;

PROC SQL;
connect to sasspds
 (dbq="&domain"
  serv="&serv"
  host="&host"
  user="anonymous");

 /* Explicitly set the fmtgrpsel option */

 execute(reset fmtgrpsel)
  by sasspds;

 title1 'To use Format on Only One Column With Group Select';
 title2 'Override Column Format With a Starndard Format';

 select *
 from connection to sasspds
  (select
   GENDER format=$1.,
   AGE format=AGEGRP.,
   count(*) as count
   from fmttest
   formatted group by GENDER, AGE);

 disconnect from sasspds;

 quit;
```

```
/* A WHERE clause that uses a format to subset  */
/* data is pushed to the server. If it is not   */
/* pushed to the server, the following warning  */
/* message will be written to the SAS log:      */
/* WARNING: Server is unable to execute the     */
/* where clause.                                */

data temp;
set &domain..fmttest;
 where put
  (AGE,AGEGRP.) = 'Child';
run;
```

For the complete code example, see “User-Defined Formats Example” on page 277.

Parallel Group-By SQL Options

SPD Server provides the following Parallel Group-By SQL reset options:

GRPSEL/NOGRPSEL

The **GRPSEL/NOGRPSEL** option enables or disables the SPD Server Parallel Group-By facility.

Usage:

```
/* Disable Parallel Group-By */
execute(reset nogrpsel)
 by sasspds ;
```

FMTGRPSEL/NOFMTGRPSEL

The **FMTGRPSEL/NOFMTGRPSEL** option enables or disables the SPD Server Parallel Group-By use of formats.

Usage:

```
/* Disable Parallel Group-By */
/* use of formats.           */
execute(reset nofmtgrpsel)
 by sasspds ;
```

SCANGRPSEL/NOSCANGRPSEL

Use the **SCANGRPSEL/NOSCANGRPSEL** option to turn the SPD Server index scan facility on and off. The default SPD Server setting uses the index scan facility.

Usage:

```
/* Disable index scan facility */
execute(reset noscangrpsel)
 by sasspds ;

/* Enable index scan facility */
```

```
execute(reset scangrpsel)
 by sasspds ;
```

SPD Server STARJOIN Facility

The SPD Server's enhanced SQL planner includes the STARJOIN facility. The SPD Server STARJOIN facility validates, optimizes, and executes SQL queries on data that is configured in a star schema. Star schemas consist of two or more normalized dimension tables that surround a centralized fact table. The centralized fact table contains data elements of interest derived from the dimension tables.

In data warehouses with large numbers of tables and millions or billions of rows of data, properly constructed star joins can minimize overhead data redundancy during query evaluation. If the SPD Server STARJOIN facility is not enabled, or of SPD Server SQL does not detect a star schema, then the SQL is processed using pair-wise joins.

How do star joins differ from pair-wise joins? In SPD Server 4.4, properly configured star joins require only three steps to complete, regardless of the number of dimension tables. SPD Server pair-wise joins require one step for each table to complete the join. If a star schema consisted of 25 dimension tables and one fact table, the STARJOIN is accomplished in three steps; joining the tables in the star schema using pair-wise joins requires 26 steps.

When data is configured in a valid SPD Server star schema, and the STARJOIN facility is not disabled, the SPD Server STARJOIN facility can produce quicker and more processor-efficient SQL query performance than would be realized using SQL pair-wise join queries.

For more information about the STARJOIN facility, see "STARJOIN Facility Reference" on page 118.

STARJOIN Options

Use the SPD Server SQL STARJOIN facility options to specify how SPD Server implements Star Joins.

NOSTARJOIN

Use the **NOSTARJOIN** option to disable or enable the SPD Server STARJOIN facility.

Usage

```
execute(reset nostarjoin=<1/0>)
  by sasspds ;
```

NOSTARJOIN=0 enables the SPD Server STARJOIN facility.

NOSTARJOIN=1 disables the SPD Server STARJOIN facility.

STARMAGIC

Use the **STARMAGIC** option to modify the behavior of the SPD Server STARJOIN and override some internal heuristics in order to favor a particular join strategy in the planner. The values are bit flags in the STARJOIN code that can be added together to result in a variety of controls.

Usage

```
execute(reset starmagic=<1/2/4/8/16>)
  by sasspds ;
```

STARMAGIC=1 forces all dimension tables to be classified as Phase I tables.

STARMAGIC=2 is currently not used.

STARMAGIC=4 requires an exact match on the FACT composite index in order to meet Phase I conditions for STARJOIN.

STARMAGIC=8 disables the IN-SET STARJOIN strategy. The IN-SET strategy is enabled by default.

STARMAGIC=16 disables the COMPOSITE STARJOIN strategy. The COMPOSITE strategy is enabled by default.

DETAILS

Use the **DETAILS** option to print details about your SPD Server STARJOIN facility settings. All internal STARJOIN debugging information is tied to the stj$ DETAILS key. Issuing the stj$ reset option displays available information as SPD Server attempts to validate a join sub-tree. The RESET DETAILS="stj$" option is very useful for debugging STARJOIN and SQL statement execution.

Usage

```
execute(reset details="stj$")
  by sasspds ;
```

STARJOIN Facility Reference

Overview: SPD Server STARJOIN Facility

SPD Server provides an enhanced SQL planner that includes the STARJOIN facility. The SPD Server STARJOIN facility validates, optimizes, and executes SQL queries on data that is configured in a star schema. Star schemas consist of two or more normalized dimension tables that surround a centralized fact table. The centralized fact table contains data elements of interest that are derived from the dimension tables.

In data warehouses with large numbers of tables and millions or billions of rows of data, a properly constructed STARJOIN can minimize overhead data redundancy during query evaluation. If the SPD Server STARJOIN facility is not enabled, or if SPD Server SQL does not detect a star schema, then the SQL is processed using pairwise joins.

How does a STARJOIN differ from a pairwise join? In SPD Server, a properly configured STARJOIN requires only three steps to complete, regardless of the number of dimension tables. SPD Server pairwise joins require one step for each table to complete the join. If a star schema consists of 25 dimension tables and one fact table, the STARJOIN is accomplished in three steps; joining the tables in the star schema using pairwise joins requires 26 steps.

When data is configured in a valid SPD Server star schema, and the STARJOIN facility is not disabled, the SPD Server STARJOIN facility can produce quicker and more processor-efficient SQL query performance than SQL pairwise joins.

Star Schemas

Overview of Star Schemas

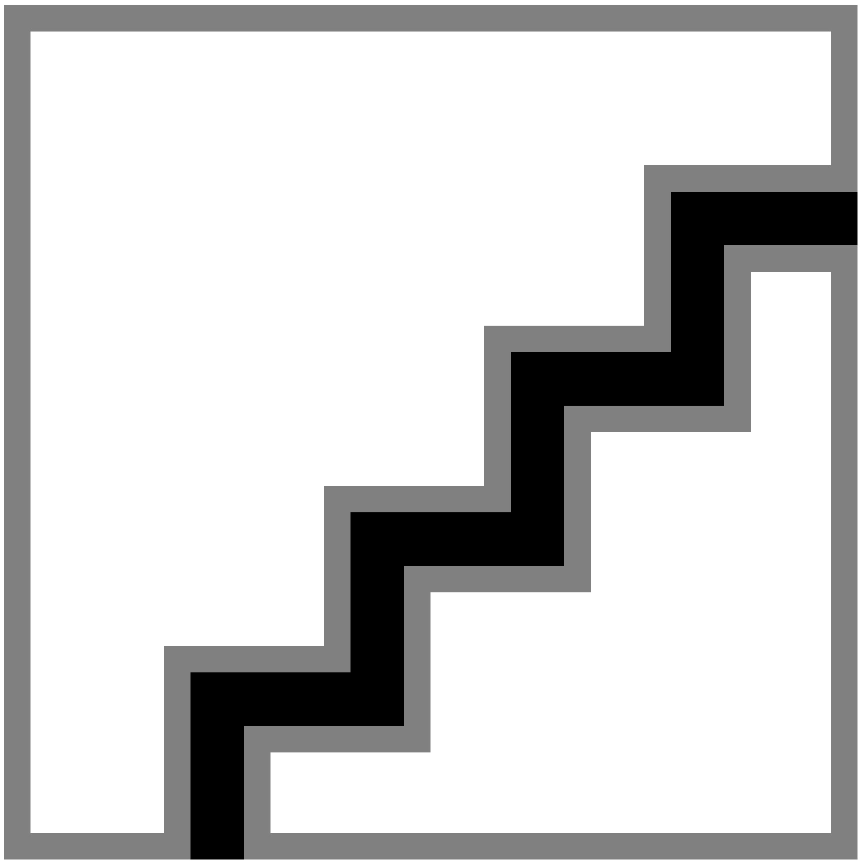
To exploit the power of the SPD Server STARJOIN facility, the data must be configured as a star schema, and it must meet specific SPD Server SQL star schema requirements.

Star schemas are the simplest data warehouse schema, consisting of a central fact table that is surrounded by multiple normalized dimension tables. Fact tables contain the measures of interest. Dimension tables provide detailed information about the attributes within each dimension. The columns in fact tables are either foreign key columns that define the links between the fact table and individual dimension tables, or they are columns that calculate numeric values that are based on foreign key data.

Figure 1 is an example of a simple star schema. The dimension tables Products, Supplier, Location, and Time surround the fact table Sales.

Figure 8.1 *Example Star Schema*

XisError: Missing Graphic - see TransformMessages.log for more information



The dimension tables, fact table, and keys in Figure 1 are used in the examples in this document.

Dimension Tables Information

Products is a table of products, with one row per unique product SKU. The row for each unique SKU contains information such as product name, height, width, depth, weight, pallet cube, and so on. The example Products table contains 1,500 rows.

Supplier is a table of the suppliers that supply the products. The row for each unique supplier contains information such as supplier name, address, state, contact representative, and so on. The example Supplier table contains 25 rows.

Location is a table of the stores selling the products. The row for each unique location contains information such as store number, store name, store address, store manager, store sales volume, and so on. The Location table contains 500 rows.

Time is a sequential sales transaction table. Each row in the Time table represents one day out of a rolling three-year, 365-day-per-year calendar. The row for each day contains information such as date, day of week, month, quarter, year, and so on. The Time table contains 1,095 rows.

Fact Table Information

The fact table Sales is a table that combines information from the four dimension tables, Products, Supplier, Location, and Time. Its foreign keys are imported, one from each dimension table: PRODUCT_CODE from Products, STORE_NUMBER from Location, SUPPLIER_ID from Supplier, and SALES_DATE from Time. The fact table Sales might have other columns with facts or information that are not found in any dimension table. Examples of fact table columns that are not foreign keys from a dimension table are columns such as QTY_SOLD or NET_SALES. The fact table in this example could contain as many as 1,500 x 25 x 500 x 1,095 = 20,531,250,000 rows.

SPD Server STARJOIN Requirements

For SPD Server SQL to take advantage of the STARJOIN planner, the following conditions must be true:

- STARJOIN optimization must be enabled in SPD Server.
- The SPD Server star schema must use a single central fact table.
- All dimension tables in the SPD Server star schema must be connected to the fact table.
- SPD Server dimension tables can appear in only one join condition.
- SPD Server fact tables are equally joined to dimension tables.
- SPD Server SQL infers fact tables by topology (common equally joined predicates).
- Dimension tables that have no subsetting require a simple index on the dimension table's join column.

When SPD Server SQL is submitted that does not meet these STARJOIN conditions, SPD Server reverts to performing the requested SQL task using SPD Server's pairwise join strategy. The “SPD Server STARJOIN Examples” on page 129 section of this document provides three examples that show valid, invalid, and restricted candidates for the SPD Server STARJOIN facility.

Enabling STARJOIN Optimization in SPD Server

SPD Server STARJOIN optimization is enabled by default. The “SPD Server STARJOIN RESET Statement Options” on page 127 section provides detailed information about statement options that enable or disable the STARJOIN facility in SPD Server.

Invoking the SPD Server STARJOIN Facility

SPD Server knows when to use the STARJOIN facility because it is topology based. SPD Server invokes STARJOIN based on the SQL that is submitted. When SQL is submitted and STARJOIN optimization is enabled, SPD Server checks the submitted SQL for admissible STARJOIN patterns. SPD Server SQL identifies a fact table by scanning for a common equally joined table among multiple join predicates in a WHERE clause. When SPD Server SQL detects patterns that have multiple equally joined operators sharing a common table, the common table becomes the star schema's fact table.

When an SQL statement that is submitted to SPD Server uses structures that indicate the presence of a star schema, the STARJOIN validation checks begin.

SPD Server STARJOIN Optimization

Overview of STARJOIN Optimization

The SPD Server STARJOIN optimization process searches for the most efficient SQL strategy to use for computations. The STARJOIN optimization process consists of three steps, regardless of the number of dimension tables that are joined to the fact table in the star schema.

1. Classify dimension tables that are called by SQL as Phase I tables or Phase II tables.
2. Phase I probes fact table indexes and selects a STARJOIN strategy.
3. Phase II performs index lookups and joins subsetted fact table rows with Phase II tables.

Classify Dimension Tables That Are Called by SQL as Phase I Tables or Phase II Tables

After the STARJOIN planner validates the join sub-tree, join optimization begins. Join optimization is the process that searches for the most efficient SQL strategy to use when joining the tables in the star schema.

The first step in optimization is to examine the dimension tables that were called by SQL for structures that SPD Server can use to improve performance. Each dimension table is classified as a Phase I table or a Phase II table. The structure of a dimension table and whether the submitted SQL filters or subsets the table's contents determine its classification. SPD Server uses different processes to handle Phase I and Phase II dimension tables.

Phase I tables can improve performance. A Phase I table is a dimension table that is either very small (nine rows or less), or a dimension table whose SQL queries contain one or more filtering criteria that is expressed with a WHERE clause. A Phase II table is any dimension table that does not meet Phase I criteria. Rows in Phase II tables that are referenced in the SQL query are not subsetted.

Consider the star schema that is illustrated in Figure 8.1 on page 119, with the fact table Sales and the dimension tables Products, Supplier, Location, and Time.

Suppose a submitted SQL query requests transaction reports from the fact table Sales for all stores where the location is the state of North Carolina, for the time period of the month of January, for all products, and for all suppliers. The SQL query subsets the Location and Time tables, so SPD Server classifies the Location and Time tables as Phase I tables. The query requests information from all of the rows in the Product and Supplier tables. Because those tables are not subsetted by a WHERE clause in the submitted SQL, STARJOIN classifies the Products and Supplier tables in this query as Phase II tables.

Now, using the same star schema, add more detail to the SQL query. Set up a new query that requests transaction reports from the fact table Sales for all stores where the location is the state of North Carolina, for the time period of the month of January, and for products where the supplier is from the state of North Carolina. The subsetted dimension tables Location, Time, and Supplier are classified as Phase I tables. The Products table, unfiltered by the submitted SQL query, is classified as a Phase II table.

Dimension tables are classified as Phase I or Phase II tables because the two types of tables require different index probe methods.

Phase I Probes Fact Table Indexes and Selects a STARJOIN Strategy

Phase I uses the SQL join keys from the subsetted Phase I dimension tables to get a smaller set of candidate rows to query in the central fact table. After optimizing the candidate rows in the fact table, the Phase I index probe examines index structures to determine the best STARJOIN strategy to use. There are two SPD Server STARJOIN strategies: the IN-SET strategy and the COMPOSITE strategy. In all but a few cases, the IN-SET strategy is the most robust and efficient processing strategy. The user can determine which strategy SPD Server chooses by providing the required table index types in the submitted SQL.

Phase I creates the smaller set of candidate rows in the central fact table by eliminating fact table rows that do not match the SQL join keys from the subsetted Phase I dimension tables. For example, if the SQL query requests information about transactions that occurred only in North Carolina store locations, the candidate rows that are retained in the fact table uses the SQL that subsets the Location dimension table:

```
WHERE location.STATE = "NC";
```

If the Sales fact table contains sales records for all 50 states, Phase I uses the SQL that subsets the Location dimension table to eliminate the sales records of all stores in states other than North Carolina from the fact table candidate rows. The example is simple, but powerful -- reducing the fact table candidate row set to transactions from only North Carolina stores eliminates massive amounts of nonproductive data processing.

The Phase I index probe inventories the number and types of indexes on the fact table and dimension tables as it attempts to identify the best STARJOIN strategy. To use the STARJOIN IN-SET strategy, Phase I must find simple indexes on all SQL join columns in the fact table and dimension tables. Otherwise, to use the STARJOIN COMPOSITE strategy, Phase I searches for the best composite index that is available on the fact table. The best composite index for the fact table is the composite index that spans the largest set of join predicates from the aggregated Phase I dimension tables.

Based on the fact table and dimension table index probe, SPD Server selects the STARJOIN strategy using the following logic:

- If one or more simple indexes are found on fact table and dimension table SQL join columns, and no spanning composite indexes are found on the fact table, SPD Server selects the STARJOIN IN-SET strategy.
- If an optimal spanning composite index is found on the fact table, and no simple indexes are found on fact table and dimension table SQL join columns, SPD Server selects the STARJOIN COMPOSITE strategy.
- If both simple and spanning composite indexes are found, SPD Server generally selects the STARJOIN IN-SET strategy, unless the composite index is an exact match for all of the Phase I join predicates, and only lesser matches are available with the IN-SET strategy.
- If no suitable indexes are found for either STARJOIN strategy, SPD Server does not use STARJOIN; it joins the sub-tree using the standard SPD Server pairwise join.

The IN-SET and COMPOSITE join strategies have some underlying differences.

The IN-SET join strategy uses an IN-SET transformation of dimension table metadata to produce a powerful compound WHERE clause to be used on the STARJOIN fact table. The "IN" part of the term "IN-SET" refers to an IN specification in the SQL WHERE clause. The IN-SET is the set of values that populate the contents of the SQL IN query expression. For example, in the following SQL WHERE clause, the cities **Raleigh**, **Cary**, and **Clayton** are the values of the IN-SET:

```
WHERE location.CITY in ("Raleigh", "Cary", "Clayton");
```

For the IN-SET strategy, Phase I dimension tables are subsetted, and then the resulting set of join keys form the SQL IN expression for the fact table's corresponding join column. You must have simple indexes on all SQL join columns in both the fact table and dimension tables before STARJOIN Phase I can select the IN-SET strategy.

If the dimension table Location has six rows for Raleigh, Cary, and Clayton, then six STORE_NUMBER values are applied to the IN-SET WHERE clause that is used to select the candidate rows from the central fact table. The STARJOIN IN-SET facility transforms the dimension table's CITY values into STORE_NUMBER values that can be used to select candidate rows from the Sales fact table. The transformed WHERE clause to be applied to the fact table might resemble the following code:

```
WHERE fact.STORE_NUMBER in
 (100,101,102,103,104,105,106);
```

You can use IN-SET transformations in a star schema that has any number of dimension tables and a fact table. Consider the following example dimension table subsetting statement:

```
WHERE location.CITY in
("Raleigh","Cary","Clayton")
 and Time.SALES_WEEK = 1;
```

Because the Sales fact table has no matching CITY column to join with the Location dimension table, and no matching SALES_WEEK column to join with the Time table, the IN-SET strategy uses transformations to create a WHERE clause that the Sales fact table can resolve:

```
WHERE fact.STORE_NUMBER in
 (100,101,102,103,104,105,106)
and Time.SALES_DATE in
 ('01JAN2005'd,'02JAN2005'd,'03JAN2005'd,
  '04JAN2005'd,'05JAN2005'd,'06JAN2005'd,
  '07JAN2005'd,);
```

The advantage of the STARJOIN facility is that it handles all of the transformations on a fact table, from dimension table subsetting to IN-SET WHERE clauses.

The COMPOSITE join strategy uses a composite index on the fact table to exhaustively probe the full Cartesian product of the combined join keys that is produced by the aggregated dimension table subsetting. SPD Server compares the composite indexes on the fact table to the theoretical composite index that is made from all of the join keys in the Phase I dimension tables. Phase I selects the best composite index on the fact table, based on the join requirements of the dimension tables.

A disadvantage of using the COMPOSITE join strategy is that when more than a few join keys exist, the Cartesian product map can become large geometric matrixes that can interfere with processing performance. You must have a composite index on the fact table that consists of Phase I dimension table join columns before STARJOIN Phase I can select the COMPOSITE join strategy.

If any Phase I dimension tables contain join predicates that do not have supporting simple or composite indexes on the fact table, those Phase I dimension tables are dropped from Phase I processing and are moved to the Phase II group.

Phase II Performs Index Lookups and Joins Subsetted Fact Table Rows with Phase II Tables

Phase I optimizes the join strategies between the Phase I dimension tables and the candidate rows from the fact table . After Phase I terminates, Phase II takes over. Phase II completes the indicated joins between the candidate rows from the fact table and the corresponding rows in the subsetted Phase I dimension tables. After completing the joins with the Phase I dimension tables, Phase II performs index lookups from the fact table to the Phase dimension II tables. Phase II dimension tables should have indexes created on all columns that join with the fact table.

When SPD Server completes the STARJOIN Phase I and Phase II tasks, the STARJOIN optimizations have been performed, the STARJOIN strategy has been selected, and the subsetted dimension tables and fact table joins are ready to run and produce the desired SQL results set.

Indexing Strategies to Optimize STARJOIN Query Performance

Overview of Indexing Strategies

Once the baseline criteria to create an SQL STARJOIN in SPD Server have been satisfied, you can configure indexing to influence which strategy the SPD Server STARJOIN facility chooses.

With the IN-SET strategy, the SPD Server STARJOIN facility can use multiple simple indexes on the fact table. The IN-SET strategy is the simplest to configure, and usually provides the best performance. To configure your work to choose the STARJOIN IN-SET strategy, create a simple index on each fact table and dimension table SQL column that you want to use in a join relation. Creating simple indexes prevents STARJOIN Phase I from rejecting a Phase I dimension table so that it becomes a non-optimized Phase II table. In addition, simple indexes facilitate the Phase II fact-table-to-dimension-table join lookup.

Consider the following SQL code for a star schema with one fact table and two dimension tables:

```
PROC SQL;
select F.FID, D1.DKEY, D2.DKEY
from fact F, DIM1 D1, DIM2 D2
where D1.DKEY EQ F.D1KEY
and D2.DKEY EQ F.D2KEY
and D1.REGION EQ 'Midwest'
and D2.PRODUCT EQ 'TV';
```

Indexing to Optimize the IN-SET Join Strategy

The SPD Server IN-SET join strategy is the preferred strategy for almost every STARJOIN. If you want the previous example code to trigger the IN-SET STARJOIN strategy, create simple indexes on the join columns for the star schema's fact table and dimension tables:

- On the fact table F, create simple indexes on columns F.D1KEY and F.D2KEY.
- On the dimension tables D1 and D2, create simple indexes on columns D1.DKEY and D2.DKEY.

Other fact table and dimension table indexes might exist that could filter WHERE clauses, but those simple indexes are the indexes that are needed to enable the STARJOIN IN-SET join strategy.

Indexing to Optimize the COMPOSITE Join Strategy

Using the COMPOSITE join strategy, the dimension tables with WHERE clause subsetting are collected from the set of equally joined predicates. A fact table composite index is needed for the fact table columns that correspond to the subsetted dimension table columns. The composite index on the fact table is necessary to facilitate the dimension tables' Cartesian product probes on the fact table rows. The STARJOIN optimizer code looks for the best composite index, which is based on the best and simplest left-to-right match of the columns in the COMPOSITE join.

If the subsetting in a STARJOIN is limited to a single dimension table, then the COMPOSITE join strategy can be enabled by creating a simple index on the join column of the single dimension table. That index is used to perform the Phase II index lookup on the fact table candidate rows. The fact table candidate row set is the result of the Phase I composite index probe.

For the previous example code to trigger the COMPOSITE STARJOIN strategy, create a composite index named COMP1 on the fact table for each of the dimension table keys: F.COMP1=(D1KEY D2KEY).

Other fact table and dimension table indexes might exist that could filter WHERE clauses, but the COMPOSITE index named COMP1 is the type of index that is needed to enable the STARJOIN COMPOSITE join strategy.

Although the COMPOSITE join strategy might appear to be a simpler configuration, the strongest utility of the COMPOSITE join strategy is limited to join relations between the fact table and dimension tables that are based on a Cartesian matrix of outcomes. As the number of dimension tables and join relations increases, the resulting Cartesian matrixes increase geometrically in size and can become unmanageable. The superior performance of the IN-SET strategy is so dramatic and robust that you should consider using the COMPOSITE join strategy only if you have good evidence that it compares favorably with the IN-SET strategy.

Example: Indexing Using the IN-SET Join Strategy

The example star schema in Figure 8.1 on page 119 has four dimension tables (Supplier, Products, Location, and Time) and one fact table (Sales) with simple indexes on the SUPPLIER_ID, PRODUCT_CODE, STORE_NUMBER, and SALES_DATE columns in the Sales fact table.

Consider the following SQL query to create a January sales report for an organization's North Carolina stores:

```
PROC SQL;
select
 sum(s.sales_amt) as sales_amt
 sum(s.units_sold) as units_sold
 s.product)code,
 t.sales_month

from
 spdslib.sales s,
 spdslib.supplier sup,
 spdslib.products p,
 spdslib.location l,
 spdslib.time t
```

```
where
 s.store_number = l.store_number
and s.sales_date = t.sales_date
and s.product_code = p.product_code
and s.supplier_id = sup.supplier_id
and l.state = 'NC'
and t.sales_date
 between '01JAN2005'd and '31JAN2005'd;

quit;
```

During optimization, the STARJOIN planner examines the WHERE clause subsetting in the SQL to determine which dimension tables qualify as Phase I tables and which are Phase II tables.

The WHERE clause subsetting of the STATE column of the Location dimension table (**where ... l.state = 'NC'**) and the subsetting of the SALES_DATE column of the Time dimension table (**where ... t.sales_date between '01JAN2005'd and '31JAN2005'd**) cause SPD Server to process the Location and Time tables as Phase I tables. The remaining dimension tables Supplier and Products are processed as Phase II tables.

SPD Server STARJOIN uses the Phase I dimension tables to reduce the rows in the fact table to candidate rows that contain the matching criteria. The values in each dimension table key are used to create a list of values that meet the subsetting criteria of the fact table.

For example, the previous SQL query is intended to create a January sales report for stores located in North Carolina. Note that the WHERE clause in the SQL code joins the Location and Sales tables on the STORE_NUMBER column. Suppose that there are 10 unique North Carolina stores, with consecutively ordered STORE_NUMBER values that run from 101 to 110. When the WHERE clause is evaluated, the results will include a list of the 10 North Carolina store IDs that existed in January 2005.

With simple indexes on the fact table and dimension tables for the STORE_NUMBER column, STARJOIN chooses the IN-SET strategy. Subsetting the STATE column values to '**NC**' enables STARJOIN to build the set of store numbers that are associated with North Carolina locations. STARJOIN can use the set of North Carolina store numbers to generate a **where ... in** SQL expression. SQL uses the **where ... in** expression to efficiently subset the candidate rows in the fact table before the final SQL expression evaluation.

In other words, STARJOIN uses a matrix of database relationships and index combinations to reorganize the SQL expression for more internal processing that can take advantage of the IN-SET join strategy. For the previous example code, the internal STARJOIN SQL reorganization resembles the following example code. The WHERE clause IN-SET statements for the STORE_NUMBER and TIME columns can be rapidly processed to subset the candidate rows in the Sales fact table. (The optimized code sections are highlighted.)

```
PROC SQL;
 select sum(s.sales_amt) as sales_amt
 sum(s.units_sold) as units_sold
s.product)code,
 t.sales_month

from spdslib.sales s,
 spdslib.supplier sup,
 spdslib.products p,
 spdslib.location l,
 spdslib.time t
 where s.store_number = l.store_number
 and s.sales_date = t.sales_date
 and s.product_code = p.product_code
 and s.supplier_id = sup.supplier_id
 and s.store_number in (101,102,103,104,105,106,107,108,109,110)
 and s.time
 in ('01JAN2005'd,'02JAN2005'd,'03JAN2005'd,
     '04JAN2005'd,      <...>, '28JAN2005'd,
     '29JAN2005'd,'30JAN2005'd,'31JAN2005'd);

quit;
```

After Phase I completes the candidate row optimization on the Sales fact table, Phase II processes the optimized query from the fact table outward. Phase II uses the values in the fact table's subsetted candidate rows to perform index lookups on the dimension tables' contents to complete the join in the most efficient manner.

SPD Server STARJOIN RESET Statement Options

Overview of STARJOIN Reset Statement Options

SPD Server recognizes several RESET statements that can configure or provide information about the STARJOIN facility in SPD Server SQL.

RESET NOSTARJOIN=[0/1]

The NOSTARJOIN option suppresses the use of the SPD Server STARJOIN optimizer in the planning and running of SQL statements that have valid STARJOIN patterns or star schemas. The statements NOSTARJOIN and NOSTARJOIN=1 are equivalent. When NOSTARJOIN is enabled, SPD Server ignores STARJOIN and uses pairwise joins to plan and run SQL statements. The default setting is NOSTARJOIN=0, meaning that in SPD Server, STARJOIN is enabled unless reset, and STARJOIN optimization occurs when SQL recognizes a valid SPD Server pattern or star schema.

RESET STARMAGIC=nnn

STARMAGIC is the STARJOIN counterpart to the SQL MAGIC number option. You can set magic numbers that direct STARJOIN to override internal heuristics, which results in enhanced join strategies. The STARMAGIC option uses bit flags to configure the STARJOIN code. You can select different controls by adding the values for the different bit flags in the following STARMAGIC set:

Table 8.1 *STARMAGIC Bit Flags*

Value	Meaning
1	forces all dimension tables to be classified as Phase I tables.
2	obsolete; not used.
4	requires exact matches on the fact table composite index to meet STARJOIN Phase I conditions.

Value	Meaning
8	disables IN-SET join strategy. (Default setting is enabled.)
16	disables COMPOSITE join strategy. (Default setting is enabled.)

RESET DETAILS="stj$"

All internal STARJOIN debugging information is tied to RESET DETAILS="stj$". Issuing this statement displays available information as SPD Server attempts to validate a join sub-tree. The RESET DETAILS="stj$" statement is useful for debugging STARJOIN and SQL statements.

Example: STARJOIN RESET Statements

The following example connects to sasspds, and then issues the "stj$" RESET option to display all available information as SPD Server attempts to validate the join sub-tree for the submitted SQL on a star schema. The STARMAGIC=16 setting disables the STARJOIN COMPOSITE join strategy (STARJOIN COMPOSITE joins are enabled by default in SPD Server). The NOSTARJOIN=0 setting means that STARJOIN is enabled (or resets a disabled STARJOIN facility) and ensures that STARJOIN optimization occurs if SQL recognizes a valid SPD Server pattern or star schema. (The STARJOIN facility is enabled by default in SPD Server.)

After submitting the following SQL statements, the code disconnects from sasspds and quits:

```
PROC SQL;
  connect to sasspds
    (dbq="star"
     server=sunburn.5007
     user='anonymous');

  execute (reset
     DETAILS="stj$"
     STARMAGIC=16
     NOSTARJOIN=0)

  by sasspds;

  execute (
     ...
     SQL statements
     ...);
  by sasspds;

  disconnect from sasspds;
quit;
```

SPD Server STARJOIN Examples

Example 1: Valid SQL STARJOIN Candidate

The following code is an example of an SQL submission that SPD Server is able to use as a star schema. The submission is a valid candidate because:

- a single central fact table, Sales, exists
- the dimension tables Time, Products, Location, and Supplier all join with the fact table Sales
- each dimension table appears in only one join condition
- all dimension tables link to the fact table using equally joined operators

```
PROC SQL;
  create table Sales_Report as
  select a.STORE_NUMBER,
         b.quarter
         c.month,
         d.state,
         e.SUPPLIER_ID

  sum(a.total_sold) as tot_qtr_mth_sales
  from   Sales a,
         Time b,
         Products c,
         Location d,
         Supplier e

  where a.sales_date   = b.sales_date
    and a.STORE_NUMBER = d.store_number
    and a.PRODUCT_CODE = c.product_code
    and a.SUPPLIER_ID  = d.supplier_id
    and b.quarter in (3, 4)
    and c.PRODUCT_CODE in (23, 100)

  group by b.quarter,
           a.STORE_NUMBER,
           b.month;
  quit;
```

Example 2: Invalid SQL STARJOIN Candidate

The following code is an example of an SQL submission that SPD Server is not able to use as a star schema because no single central fact table can be identified. Changes to the previous code example are highlighted:

```
  PROC SQL;
  create table Sales_Report as
  select a.STORE_NUMBER,
         b.quarter
         c.month,
         d.state,
         e.SUPPLIER_ID

  sum(a.total_sold) as tot_qtr_mth_sales
```

```
from   Sales a,
       Time b,
       Products c,
       Location d,
       Supplier e

where a.sales_date   = b.sales_date
  and a.STORE_NUMBER = d.store_number
  and a.PRODUCT_CODE = c.product_code
  and c.SUPPLIER_ID  = d.supplier_id
  and b.quarter in (3, 4)
  and c.PRODUCT_CODE in (23, 100)

group by b.quarter,
         a.STORE_NUMBER,
         b.month;
quit;
```

SPD Server is not able to use the SQL submission in this example as a star schema This submitted code joins the dimension tables for Time, Products, and Location to the Sales table, but the table for Supplier is joined to the Sales table through the Products table. As a result, the topology does not define a single central fact table.

Example 3: STARJOIN Candidate with Created or Calculated Columns

The STARJOIN facility in SPD Server supports calculated or created columns. The following code is an example of an SQL submission that creates columns, but still uses STARJOIN optimization, if the central fact table and the dimension tables contain indexes on the join columns for the STARJOIN:

```
PROC SQL;
create table &Regional_Report as
select case d.state
  when 1 then 'NC'
  when 2 then 'SC'
  when 3 then 'GA'
  when 4 then 'VA'
  else '  '

  end as state_abv,
  b.quarter,
  sum (a.tot_amt) as total_amt

from wk_str_upd_t a,
     week_t b,
     location_t d,

where a.we_dt       = b.we_dt
  and a.chn_str_nbr = d.chn_str_nbr
  and b.quarter     = 2

group by d.state,
         b.quarter
  having d.state in (1,2,3,4);
quit;
```

The highlighted code creates a column called **state_abv**. The SPD Server STARJOIN facility supports created columns if the appropriate indexes on the join columns exist in the fact table and dimension tables.

SPD Server Index Scan

SPD Server SQL provides users with the capability to use lightning-fast index scans on large tables. Rather than scanning entire tables, which can have million or billions of rows, SPD Server SQL is able to scan cached index metadata instead of sequentially scanning entire large tables. SPD Server SQL provides enhanced index scan support for the following functions:

min, max, count, nmiss, range uss, css, std, stderr,and **var.** All of the functions can accept the DISTINCT term as well.

All index scan capabilities listed above are available for both standard SPD Server tables as well as clustered tables, with the exception of the DISTINCT qualifier. The DISTINCT index scan function is not available in clustered tables.

The count(*) function is the only function included with the index scan support enhancement that does not require an index on the table. For example,

```
select count(*) from tablename;
```

returns the number of rows in the large table tablenamewithout performing a row scan of the table. Table metadata is able to return the correct number of rows. As a result, the response is as fast as an index scan, even on an unindexed table in this case.

Count(*) functions with WHERE clauses require an index for each column referenced in the WHERE clause, in order for the index scan feature to provide the performance enhancement. For example, suppose SPD Server table Foo has indexes on numeric columns a and b. The following count(*) functions benefit from SPD Server index scan support:

```
select count(*)
  from Foo
    where a = 1;

select count(*)
  from Foo
    where a LT 4
    and b EQ 5;

select count(*)
  from Foo
    where a in (2,4,5)
    or b in (10,20,30);
```

All functions other than count(*) require an index on function columns in order to exploit the index scan performance savings. Minimal WHERE clause support is available for these queries, as long as all functions use the same column, and the WHERE clause is a simple clause that uses the LT, LE, EQ, GE, GT, IN, or BETWEEN operator for that column. For example, suppose that the SPD Server table Bar has indexes on numeric columns x and y. The following SQL submissions are able to exploit the performance gains of index scans:

```
select min(x),
       max(x),
       count(x),
       nmiss(x),
       range(x),
       count(distinct x)
 from Bar;
select min(x),
       max(x),
       count(x),
       nmiss(x),
       range(x),
       count(distinct x)
 from Bar
  where x between 5 and 10;

select min(x),
       max(x),
       count(x),
       nmiss(x),
       range(x),
       count(distinct x)
 from Bar
  where x gt 100;

select min(x),
       min(y),
       count(x),
       count(y)
 from Bar;
```

If any one function in a statement does not meet the index scan criteria, all functions in that statement revert to being resolved by table scan instead of index scan. Suppose the SPD Server table Oops has indexes on numeric columns x and y. Column z is not indexed. Then, the SPD Server SQL statement below

```
select min(x),
       min(y),
       count(x),
       count(y),
       count(z)
 from Oops;
```

is entirely evaluated by table scan; index scanning is not performed on any of the functions. To take advantage of index scans, the statement above could be resubmitted as

```
select min(x),
       min(y),
       count(x),
       count(y)
from Oops;

select count(y)
from Bar;
```

The functions min(x), min(y), count(x), and count(y) are evaluated using index scan metadata and exploit the performance gains. The function count(y) continues to be evaluated by table scan. The count(*) function can be combined with other functions and benefit from index scan performance gains. Continuing to use the SPD Server table Oops with indexes on numeric columns x and y, the following SPD Server SQL statement benefit from index scan performance:

```
select min(x),
       range(y),
       count(x),
       count(*)
from Oops;
```

SPD Server Index Scan is an extension to the SPD Server Parallel Group-By facility. The query must first be accepted by Parallel Group-By to be evaluated for an Index Scan. The section on "Parallel Group-By Facility " on page 112 contains more detailed information. When SPD Server uses the Index Scan optimization, the following message is printed to the SAS log:

```
SPDS_NOTE: Metascan used to resolve this query.
```

Optimizing Correlated Queries

Intelligent storage must have the ability to interpret and process complex requests such as correlated queries. A correlated query is a select expression where a predicate within the query has a relationship to a column that is defined in another scope. Today's business and analytic intelligence tools often generate SQL queries that are nested 3 or 4 layers deep. Queries with cross-nested relationships consume significant processor resources and require more time to complete processing. New algorithms in the SQL Planner of SPD Server implement techniques that significantly improve the performance of correlated queries for patterns that permit query rewrites or query de-correlation.

The SQL Planner improves correlated query performance by changing complex rules about nested relationships into a series of simple steps. SPD Server can process the simple steps much faster than it can process the complex rules that arise with multiple levels of nesting. When a query with multiple levels of nesting is submitted to the SQL Planner, the planner examines the relationships between nested and unnested sections of the query. When a complex nested relation ship is found, the SQL Planner restructures or recodes the SQL query into a simpler form using temporary SPD Server tables.

Development work continues to improve the range of sub-expressions that are addressed by the SPD Server SQL rewrite facility. For more information, see Chapter 8, "SPD Server SQL Query Rewrite Facility" in the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.

Correlated Query Options

The following are SPD Server SQL options for use with correlated query rewrites:

_QRW/NO_QRW

Use the **_QRW/NO_QRW** option to configure SPD Server to enable or disable the query rewrite facility diagnostic output. Specifying this SPD Server RESET option enables or disables various debugging and tracing outputs from the query rewrite facility. The debugging and tracing outputs are generated when the SPD Server query rewrite facility detects sub-expressions that it rewrites and executes the SQL code. The SQL code produces the intermediate results and final rewritten SQL statement. By default, the SPD Server **_QRW** option for diagnostic output is not enabled.

SPD Server provides alternate expressions that do the same thing as the **_QRW/NO_QRW** option. They are the **_QRW=1/_QRW=0** option and the **NO_QRW=0/NO_QRW=1** option.

Usage:

```
/* Enable query rewrite diagnostics */
execute(reset _qrw)
 by sasspds ;

/* A second way to enable    */
/* query rewrite diagnostics */
execute(reset _qrw=1)
  by sasspds ;

/* A third way to enable     */
/* query rewrite diagnostics */
execute(reset no_qrw=0)
 by sasspds ;

/* Disable query rewrite diagnostics */
execute(reset no_qrw)
  by sasspds ;

/* A second way to disable query */
/* rewrite diagnostics           */
execute(reset _qrw=0)
 by sasspds ;

/* Another way to disable query */
/* rewrite diagnostics          */
execute(reset no_qrw=1)
  by sasspds ;
```

_QRWENABLE/NO_QRWENABLE

Use the **_QRWENABLE/NO_QRWENABLE** option to completely disable the SPD Server query rewrite facility. Disabling the query rewrite facility prevents the rewrite planner from intervening in the SQL flow and from making any optimizing rewrites. This option is not normally specified unless you want to test if an SQL statement would run faster without rewrite optimization, or if you suspect that the resulting row set that you get from a query rewrite evaluation is incorrect.

SPD Server provides an alternate expression that does the same thing as the **_QRWENABLE/NO_QRWENABLE** option. It is the **_QRWENABLE=1/_QRWENABLE=0** option. The query rewrite facility is enabled in SPD Server by default.

Usage:

```
/* Disable query rewrite */
/* facility              */
execute(reset no_qrwenable)
 by sasspds ;
```

```
/* A second way to disable */
/* query rewrite facility  */
execute(reset _qrwenable=0)
  by sasspds ;

/* Enable query rewrite  */
/* facility              */
execute(reset _qrwenable)
  by sasspds ;

/* A second way to enable  */
/* query rewrite facility  */
execute(reset _qrwenable=1)
  by sasspds ;
```

SPD Server Views

SPD Server supports the creation of SQL views. A view is a virtual table based on the result set of an SQL statement. An SPD Server view can reference only SPD Server tables. You should use SPD Server explicit pass-through SQL syntax to create SPD Server views, as follows:

```
EXECUTE(Create view <viewname> as Select) BY [sasspds|alias];
```

Creating an SQL view results in a view file that is created in the specified domain with the name <viewname>.view.0.0.0.spds9. After an SQL view is created, the SPD Server view can be used in SPD Server SQL queries as a table.

View Access Inheritance

SPD Server uses view access inheritance to control access to tables that are referenced by SPD Server views. View access inheritance allows a user that has rights to a given view to also have rights to access the individual component tables that comprise the view.

For example, user Stan creates tables **`WinterSales`** and **`SpringSales`**, and then Stan creates a view that joins the two tables. Stan gives user Kyle Read access to the view. Using view access inheritance, because Kyle has Read access to the view of the joined tables, Kyle also has Read access to the individual component tables **`WinterSales`** and **`SpringSales`**.

```
/* User Stan creates tables WinterSales and SpringSales.   */
/* Only user Stan can read these tables directly.          */

LIBNAME Stan sasspds 'temp' user='Stan';
DATA Stan.WinterSales;
INPUT idWinterSales colWinterSales1 $ colWinterSales2 $ ... ;
...
;

DATA Stan.SpringSales;
```

```
INPUT idSpringSales colSpringSales1 $ colSpringSales2 $ ... ;
...
quit;

/* Stan creates view WinterSpring to join tables WinterSales */
/* and SpringSales. Stan gives user Kyle read access to the  */
/* view.  Because Kyle has rights to read view WinterSpring, */
/* he also has read access rights to the individual tables   */
/* that Stan used to create the view WinterSpring. Kyle can  */
/* only read the tables WinterSales and SpringSales through  */
/* the view WinterSpring. If Kyle tries to directly access   */
/* the table WinterSales or the table SpringSales, SPD       */
/* Server does not comply and issues an access failure       */
/* warning.                                                  */

PROC SQL;
CONNECT TO sasspds(dbq='temp' user='Stan';
EXECUTE(create view WinterSpring as
        SELECT * from SpringSales, WinterSales
        WHERE SpringSales.id = WinterSales.id);
quit;

PROC SPDO lib=Stan;
SET ACLUSER;
SET ACLTYPE=view;
ADD ACL WinterSpring;
MODIFY ACL WinterSpring / Stan=(y,n,n,n);
quit;
```

SPD Server view access inheritance is available only when it is invoked with SPD Server explicit pass-through SQL syntax. If the view is accessed directly through SAS SQL or a SAS DATA step, the user of the view must also have direct access to the component tables that are referenced in the view. In this case, the ACL credentials of the user of the view are applied to the component view tables. This restriction limits the usefulness of SPD Server views that are accessed via SAS SQL to cases where a SAS SQL user creates a virtual table to simplify SQL coding.

Even though the SPD Server view access inheritance feature is not available for views issued from a SAS session, the ability to create and use views on SPD Server tables from SAS is a significant improvement in functionality.

Materialized Views

SPD Server allows users to create an SQL view as a materialized view. What makes a materialized view different from an SQL view? For a materialized view, the results of the view statement are computed and saved in a temporary SPD Server table at the time the view is created. For a standard SQL view the results are computed each time the view is referenced in a subsequent SQL statement. As long as there are no changes to any of the input tables that the view consists of, the materialized view returns the results from the temporary table when the view is referenced in an SQL statement. If any of the input tables that comprise the view are modified, the materialized view will recompute the results the next time the view is referenced and refreshes the temporary table with the new results. The materialized view temporary results table exists for as long as the view is in existence. When a view is dropped or deleted, then the temporary results table is also deleted.

Materialized Views Operating Details

A materialized view can be created only at the time the SQL view is created. This feature is available only through the SPD Server 4.5 SQL pass-through facility. A new keyword Materialized is added to the Create View syntax that identifies the view to be created as a materialized view. When a materialized view is created, the Create View operation does not complete until the temporary results table is populated. This can add substantial time to the execution of Create View.

Each time a created materialized view is referenced in an SQL statement, there is a check to determine whether any of the input tables used to produce the temporary results table have been modified. If not, the temporary table is substituted in place of the vie w file within the SQL statement. If any of the input tables have been modified, the SQL statement executes without this substitution so it acts as if it is a standard SQL view reference. There is also a background thread launched at this time that is independent of the SQL statement execution, which refreshes the temporary results table. Until this refresh is completed, any incoming references to the view is treated as standard view references.

Creating a standard SQL view results in a view file being created in the specified domain with the name <viewname>.view.0.0.0.spds9. Creating a materialized view results in an additional SPD Server table being created in the same domain as the view file with the name format <.viewname>.mdfspds9 and corresponding .dpf files <.viewname>.dpfspds9. The materialized view table is not visible or accessible to the user by using PROC DATASETS or other SAS procedures. If one or more simple indexes are defined on any of the input tables that are used to create the results table, the indexes are also created on the materialized view table, as long as the column that was indexed in the input table also exists in the materialized view table.

User Interface for Materialized Views

To create a materialized view, use the following SQL pass-through syntax.

```
EXECUTE (Create Materialized View <viewname> as Select ) BY [sasspds | alias];
```

All other references to the view follow the existing SQL syntax, whether it is a standard SQL view or a materialized view. The Materialized keyword is used only in the Create statement. For example, to drop a materialized view, you would use the following syntax.

```
EXECUTE (Drop View <viewname> ) BY [sasspds | alias];
```

If any of the input tables to a materialized view are modified, the next time the view is referenced, a refresh is performed on the materialized view table. You can use an **spdsserv.parm** file option setting to specify the time delay before the materialized view table is refreshed.

```
MVREFRESHTIME=<number-of-seconds> ;
```

Where <number-of-seconds> specifies the number of seconds before the refresh starts. You can set the MVREFRESHTIME= option to any integer value between 0 and 86400. The default MVREFRESHTIME= specification is 30 seconds.

The reason that a time delay might be necessary before refreshing a materialized view table is to prevent processor thrashing. Processor thrashing might occur if you refresh the materialized view table when other processes are concurrently processing updates to the tables that are used in the view. If your computing environment does not perform multiple concurrent table updates, then you can set MVREFRESHTIME=0 and eliminate any time delay associated with materialized view refreshes.

Benefits of Materialized Views

Creating a materialized view instead of a Standard SQL view can provide enormous performance benefits when the view is referenced in an SQL statement. For views that contain costly operations such as multiple table joins or operations on very large tables, the execution time for queries containing a materialized view can be orders of magnitude less than a standard view. If the results table produced by the view is relatively small in comparison with the input tables, the execution time for queries using a materialized view might be a few seconds versus several minutes for a standard view.

For example, if it takes on average 20 minutes to produce the result set from a view and the result is in the order of thousands of rows or less, a query referencing the materialized view now takes seconds. Previously using the standard view operation s, every time the view was referenced would result in 20 minutes of execution time. The performance benefits should be measured on a case by case basis.

The decision of whether to use a standard view or a materialized view can be primarily driven by how often the input tables to the view are updated versus how often the view is referenced in an SQL statement. If a view is being referenced at least twice be fore any updates can occur, then the materialized view should provide superior performance. In cases where the defined view can be created very quickly, there is probably not a need for using a materialized view. If the input tables are frequently updated in comparison to how often the view is referenced, a standard view would probably be more efficient.

Materialized View Example

The following code shows the creation and use of a materialized view. The input tables X and Z are created with X having three columns a,b,c and Z having four columns a,b,c,d respectively.

```
data mydomain.X;
  do a = 1 to 1000;
     b = sin(a);
     c = cos(a);
  output;
end;
run;

data mydomain.Z;
  do a = 500 to 1500;
     b = sin(a);
     c = cos(a);
     d = mod(a,99);
  output;
end;
run;

PROC SQL;
connect to sasspds (dbq='mydomain'
  host='myhost'
  serv='myport'
  user='me'
  passwd='mypasswd');

execute (create materialized view XZVIEW as
         select *
           from Z
           where a in
             (select a from X))
           by sasspds;

         select *
           from connection
           to sasspds
            (select *
              from XZVIEW
              where d >90);

execute (drop view XZVIEW);
quit;
```

SPD Server SQL Extensions

SPD Server SQL furnishes several extensions to the SQL language. These extensions are not a part of standardized industry SQL, but they are an integral part of the SPD Server system. These extensions enable systemic data management unique to the SPD Server. The SPD Server SQL uses a special pass-through facility that uses these extensions for data manipulation and extraction operations. The following section discusses the roles of the following extensions, which enable SPD Server's SQL pass-through facility. In addition to the extensions in this section, users should know "Libref Statements" on page 98 and "Libref Clauses" on page 98.

BEGIN and END ASYNC OPERATION Statements

Overview of BEGIN and END ASYNC Operation Statements

Asynchronous statements are a useful technique you can use to harness the multi-processor power of SPD Server. Asynchronous statements enable execution of multiple, independent threads at the same time. The BEGIN ASYNC OPERATION and END ASYNC OPERATION statements enable you to delimit one or more statements for asynchronous, parallel execution. Since the statements execute in parallel, they must not depend on another, because there is no way to guarantee which statement will finish before another statement executes. SPD Server software initiates thread execution according to the order of the statements in the block.

Usage:

```
execute ([ BEGIN | END ] ASYNCH OPERATION);
```

Illegal ASYNC Block Statements

The statements in this Illegal ASYNC Block example have illegal interdependencies and cannot be expected to work correctly:

```
/* Example of Illegal ASYNC Block Code   */

      PROC SQL;
         connect to sasspds
            (dbq="my-domain"
             server=host.port
             user='user-name'
             password='user-password'
             other connection options);

         execute(begin async operation)
            by sasspds;

         execute(create table T1 as
            select *
            from SRC1)
            by sasspds;

         execute(create unique index I1 on
            T1(a,b))
            by sasspds;

         execute(end async operation)
            by sasspds;

         disconnect from sasspds;
      quit;
```

The example violates the interdependency rule. The **create index** statement assumes table **T1** exists and is complete. However, table **T1** is created from table **SRC1**, and might not be complete before the asynchronous create index statement executes. Hence, index **I1** is dependent on a complete table **T1**. The resultant data would not be reliable. The purpose of this example is to illustrate the concept of interdependency, and how **not** to write an ASYNC block.

Legal ASYNC Block Statements

The statements in this Legal ASYNC Block example have no interdependencies.

```
/*  Example of Legal ASYNC Block Code            */
/* Creates some tables in the first ASYNC block  */
/*                                               */

   PROC SQL;
      connect to sasspds
        (dbq="path1"
         server=host.port
         user='anonymous');

      execute(begin async operation)
         by sasspds;

      execute(create table state_al as
        select *
        from allstates
        where state='AL')
        by sasspds;

      execute(create table state_az as
        select *
        from allstates
```

```
          where state='AZ')
          by sasspds;
          ...

        execute(create table state_wy as
           select *
           from allstates
           where state='WY')
           by sasspds;

        execute(end async operation)
          by sasspds;

  /*                                              */
  /* Create some indexes in the second ASYNC block  */
  /*                                              */

        execute(begin async operation)
          by sasspds;

        execute(create index county on
                state_al(county))
          by sasspds;

        execute(create index county on
                state_az(county))
          by sasspds;
          ...

        execute(create index county on
                state_wy(county))
          by sasspds;

        execute(end async operation)
          by sasspds;

       disconnect from sasspds;
      quit;
```

Why does the second example work correctly? First, each table is created independently. Second, there is a 'synchronization point': the first **END ASYNC** operation. This point ensures that all the tables are created before the second ASYNC statement block begins. (You can also achieve results that are similar to this example by using the LOAD statement).

Using Librefs in an ASYNC Block Statement

To refer to a two-part table name inside an ASYNC block, you must re-execute the libref statement issued before entering the block. Conversely, if you issue a libref statement inside the ASYNC block, it does not extend outside the ASYNC block. An ASYNC block creates a distinct scope for the libref. To work correctly, a libref statement must be located inside the ASYNC block, and the libref statement must precede the first SQL statement that references it.

```
  /*  Example of Legal Code using LIBREFs in an ASYNC Block  */
  /*  Create some tables in the first ASYNC block            */
```

```
PROC SQL;
   connect to sasspds
     (dbq="path1"
      server=host.port
      user='anonymous');

   execute(begin async operation)
      by sasspds;

   execute(libref path1 engopt='dbq="path1"
      server=host.port
      user="anonymous"')
      by sasspds;

   execute(libref path2 engopt='dbq="path1"
      server=host.port
      user="anonymous"')
      by sasspds;

   execute(create table path1.southeast as
      select a.customer_id,
             a.region,
             b.sales
      from   path1.customer a,
             path2.orders b
      where  a.customer_id = b.customer_id
      and    a.region='SE')
      by sasspds;

            ....

    execute(create table path1.northeast as
        select a.customer_id,
               a.region,
               b.sales
        from   path1.customer a,
               path2.orders b
        where  a.customer_id = b.customer_id
        and    a.region='NE')
        by sasspds;

   execute(end async operation)
      by sasspds;

   disconnect from sasspds;
quit;
```

Using SQL Options in an ASYNC Block Statement

SPD Server SQL options must be set globally for all execute statements in the ASYNC block. These options must be set using an execute statement before the BEGIN ASYNC operation. This example uses code blocks from the preceding example to show how to print a method tree without executing the SQL.

```
/*                                            */
/*  Example of Legal SQL Options in ASYNC Block  */
/*                                              */

   PROC SQL;
       connect to sasspds
          (dbq="path1"
           server=host.port
           user='anonymous');

       execute(reset noexec _method)
         by sasspds;

       execute(begin async operation)
         by sasspds;

       execute(libref path1
          engopt='dbq="path1"
          server=host.port
          user="anonymous"')
          by sasspds;

       execute(libref path2
          engopt='dbq="path1"
          server=host.port
          user="anonymous"')
          by sasspds;

       execute(create table path1.southeast as
          select a.customer_id,
                 a.region,
                 b.sales
          from   path1.customer a,
                 path2.orders b
          where  a.customer_id = b.customer_id
          and    a.region='SE')
          by sasspds;

               ....

       execute(create table path1.northeast as
          select a.customer_id,
                 a.region,
                 b.sales
          from   path1.customer a,
                   path2.orders b
          where  a.customer_id = b.customer_id
          and    a.region='NE')
          by sasspds;

       execute(end async operation)
         by sasspds;

       disconnect from sasspds;
   quit;
```

LOAD Statement

The LOAD statement enables table creation (with one or more indexes) with a single statement. The data source for the statement is a SELECT clause. The **SELECT** list in the clause defines the columns for the new table. All characteristics of the columns (variables) in the select list are preserved, becoming permanent attributes of the new table's column definitions. The target table for the LOAD TABLE statement must be on the local machine.

Usage:

```
execute (LOAD TABLE table spec
    < WITH index spec
    < WITH index spec>>
  by sasspds;
```

In the following example, each execute statement creates a table for one U.S. state using a global table called STATE that contains many states. The first execute statement uses LOAD to create table STATE_AL (Alabama), and creates an index on the COUNTY column. The structure of the state table STATE_AL and the data in the state table both come from the global table STATE. The data in STATE_AL is the subset of all records from STATE where the STATE column variable equals 'AL'. LOAD creates a table for states from Alabama to Wyoming, with each state's table indexed by county and mirroring the structure of the parent table STATE.

```
    execute(load table state_al
        with index county
        on (county) as
        select *
        from state
        where state='AL')
      by sasspds;

    execute(load table state_az
        with index county
        on (county) as
        select *
        from state
        where state='AZ')
      by sasspds;

        ...

    execute(load table state_wy
        with index county
        on (county) as
        select *
        from state
        where state='WY')
      by sasspds;
```

In general, the LOAD statement is faster than a corresponding create table / create index statement pair, because it builds the table and associated index(es) asynchronously using parallel processing.

COPY Statement

The COPY table statement creates a copy of an SPD Server table with or without the table index(es). For the COPY table statement to work, the source and target tables must be on the local machine. By default, the software creates an index(es). The COPY table statement is faster than either of the following CREATE and LOAD statements:

```
create table ...
as select ...
create index ...
```

or

```
load table ...
with index...
as select ...
```

The COPY statement is faster than the two above statements because it uses a more direct access path than the SQL SELECT clause when accessing the data.

In the example that follows, two new tables are created: T_NEWand T2_NEW. The first table, T_NEW, is created with index structures identical to table T_NEW. The second table, T2_NEW, is unindexed regardless of the structure of table T2_OLD.

```
     execute(copy table t_new
             from t_old)
        by sasspds;

     execute(copy table t2_new
             from t2_old
             without indexes)
        by sasspds;
```

The COPY statement also supports an ORDER BY clause that you use to create a new table with a sort order on one or more columns of the new table. While COPY TABLE does not support all of the options of PROC SORT, you can achieve substantial performance gains when creating this ordered table by using COPY with an ORDER BY clause when appropriate.

The next example copies the table T_OLD to T_NEW using the order by clause. The data is ordered by columns: x in ascending order, y in descending order, and z in ascending order. The results are the same if you run PROC SORT on the columns using the same BY clause. The syntax of the COPY ORDER BY follows the normal SQL ORDER BY clause, but the column identifiers that you can specify are restricted. You can specify only actual table columns when using the COPY ORDER BY clause.

```
execute(copy table t_new
       from t_old
       order by x, y desc, z asc)
  by sasspds;
```

Differences between SAS SQL and SPD Server SQL

This section overviews some of the functional differences between SAS SQL and SPD Server SQL. A great deal of SAS SQL functionality is integrated into SPD Server. Exceptions between SAS and SPD Server SQL are listed below.

Reserved Keywords

SPD Server uses keywords to initiate statements or refer to syntax elements. For example, the words **where** and **group** can be used only in certain ways, because there are WHERE and GROUP BY clauses. Keywords are treated as reserved words. That means you cannot use keywords when naming a libref, a table, a column or an index.

In contrast, SAS allows keywords in some, but not all, syntax locations. The documentation chapter Chapter 9, "SPD Server SQL Syntax Reference Guide," on page 151 contains a list of "(Reserved) Keywords" on page 154.

Table Options and Delimiters

SPD Server SQL uses brackets to delimit table options. SAS SQL uses parentheses as delimiters. You can place table options in a create table statement. You must put table options within parentheses to delimit column definitions within a table.

Mixing Scalar Expressions and Boolean Predicates

SPD Server SQL does not allow mixing scalar expressions with Boolean predicates. SAS SQL does allow mixing scalar expressions with Boolean predicates in most places. The Help section on "Scalar Expressions Contrasted with Boolean Predicates" on page 153 contains more information about permissible expression content.

INTO Clause

SPD Server SQL does not support the INTO clause, as in

```
select a, b into :var1, :var2 from t where a > 7;
```

In contrast, SAS SQL supports the INTO clause.

Tilde Negation

SPD Server SQL supports the use of the tilde only to negate the 'equals' operator, ~= (not equals). SAS SQL allows broader use of the tilde ('~') character, where the tilde is synonymous with not and can be combined with various operators. For example, SAS SQL can use the tilde with 'between' ~ between (not between). SPD Server does not recognize that expression.

Nested Queries

SAS SQL permits sub-queries without parentheses delimiters in more places than SPD Server SQL. SPD Server SQL uses parentheses to explicitly group sub-queries or expressions that are nested within a query statement whenever possible. Queries with nested expressions execute more reliably and are also easier to read.

USER Value

SAS SQL permits sub-queries without parentheses delimiters in more places than SPD Server SQL. SPD Server SQL uses parentheses whenever possible to explicitly group sub-queries or expressions that are nested within a query statement. Queries with nested expressions execute more reliably and are easier to read.

SPD Server SQL does not support the USER keyword in the INSERT statement. For example, the following query fails in SPD Server SQL:

```
insert into t1(myname) values(USER);
```

Supported Functions

SPD Server SQL supports the following functions:

```
abs, addr, arcos, arsin, atan, band, betainv, blshift, bnot, bor, brshift, bxor,
byte, ceil, cinv, collate, compbl, compound, compress, cos, cosh, css, cv,
daccdb, daccdbsl, daccsl, daccsyd, dacctab, date, datejul, datepart, datetime,
day, dcss, depdb, depdbsl, depsl, depsyd, deptab, dequote, dhms, digamma, dmax,
dmean, dmin, drange, dstd, dstderr, dsum, duss, dvar, erf, erfc, exp, finv,
fipname, fipnamel, fipstate, floor, fnonmiss, fuzz, gaminv, gamma, hms, hour,
int, intck, intnx, intrr, irr, ispexec, isplink, kurtosis, left, length, lgamma,
log, log10, log2, lowcase, max, mdy, mean, min, minute, mod, month, mort, n,
netpv, nmiss, npv, ordinal, poisson, probbeta, probbnml, probchi, probf, probgam,
probhypr, probit, probnegb, probnorm, probt, qtr, quote, range, ranuni, rank,
recip, repeat, reverse, right, round, saving, second, sign, signum, sin, sinh,
skewness, sqrt, std, stderr, stfips, stname, stnamel, substr, sum, tan, tanh,
time, timepart, tinv, today, tranwrd, trigamma, trim, upcase, uss, var, weekday,
year, zipfips, zipname, zipnamel, and zipstate.
```

Ranuni functions can show slight variation from run to run due to the impact of parallel processing.

Note that **date, int, left, right** and **trim** are reserved keywords. Therefore, they must be preceded by a backslash in SPD Server SQL queries:

```
select \date() from t ;
```

Part 4

SPD Server SQL Reference

Chapter 9

SPD Server SQL Syntax Reference Guide

Overview

This chapter describes the SQL syntax that is allowed with the Scalable Performance Data (SPD) Server. SPD Server SQL is a dialect of SQL. That is, it combines SQL-92, SAS SQL and extensions that are specific to SPD Server. Whenever possible, SPD Server attempts to conform to SAS SQL.

Document Conventions

Productions

The syntax uses building blocks that are called productions. Productions are denoted by the symbol ::= . To the left of the equal sign is a production name; to the right of the equal, or on the next line, is a list of production constructs. If a production has more than one possible construct, the alternatives are separated by a vertical bar |. Read productions top-down. For example, reading the delete statement, there are literal keywords and two subproductions, a table specification and the WHERE clause.

Production Links / References

Subproductions that are referenced within a production definition are HTML links to their definitions. You can navigate the links with an HTML browser.

Literal Text

Traversing down a syntax tree leads to leaf or terminal definitions. The definitions are composed either of keywords (select), identifiers (names of tables, columns, and so on) or symbols (punctuation, operators, and so on.). Keywords and identifiers are shown with bold, capitalized text. In contrast, symbols are shown with single quotation marks and are bold.

Optional Text

Optional syntax is delimited by square brackets, [and]. Optional lists (syntax elements that are repeated) are denoted by [and]*. The * signifies zero or more occurrences of the bracketed syntax.

Selection Lists

Selection lists, that allow you to choose from a list of alternative syntax elements, are denoted by braces { and }. These elements are separated by a vertical bar |. The selection list itself is **not** optional; you must choose at least one element. If you must choose one or more of the elements, the list is terminated with a }+. The + indicates one or more occurrences of the delimited syntax.

Note: The browser displays links best with underscores. To view underscores using Netscape, refer to the option under the File Command: Options/General Preferences/ Appearance.

SQL Syntax Definitions

Statement (Query)

One or more syntax elements terminated by a semicolon.

Scalar Expressions Contrasted with Boolean Predicates

Scalar expressions represent a single data value, either a numeric or a string from a constant specification. Examples include: 1, 'hello there', '31-DEC-60'd), a function (that is, avg(a*b)), a column/variable (that is foo.bar), the case expression, or even a subquery which returns a single run-time value. Boolean predicates are either true or false. They are used in WHERE clauses, having clauses and in the case expression. You cannot select predicates, nor can you assign them to columns (that is, in an update statement). Scalar expressions and Boolean predicates **cannot** be used interchangeably, although SAS SQL does allow you to mix the expressions.

Strings

SPD Server SQL strings are character streams that are delimited by either single or double quotation marks. If you use a single quotation mark to begin a string, you must use a single quotation mark to terminate the string. To embed a single quotation mark in a string, use two single quotation marks together. For example,

```
SELECT 'it''s a wonderful life' from mytable.
```

You can use double quotation marks in the same manner. There is another way to embed a single quotation mark without doubling the character. You can use double-quotation marks as delimiters. For example,

```
SELECT it's a wonderful life from mytable.
```

In some of the syntax specifications that follow, a user-defined or database-specific string is noted. Delimit these strings with a bracket or parenthesis. Characters between the delimiters are considered part of the string up to, but not including, the matching delimiter.

```
CONNECT to sasspds(
  user='john'
  passwd='foobar'
  options=(a b c)
  );
```

The dbms_options string is

```
user='john'
passwd='foobar'
options=(a b c).
```

In this example, the first right-parenthesis is considered part of the string. It is not the matching termination delimiter.

Identifiers

Identifiers are the names of librefs, tables, indexes and columns, as well as table and column aliases.

(Reserved) Keywords

Keywords are used to initiate statements and syntax elements. For example, WHERE or GROUP BY clauses. Keywords are also reserved. They cannot be used for identifiers because this use introduces ambiguity. For example, select unique from; is a valid but ambiguous statement. Below is a list of current SPD Server keywords. Some words have been reserved for future enhancements to SPD Server SQL:

```
add, all, alter, and, any, as, asc, async, begin, between, both, by,
calculated, cascade, case, char, character, column, connect,
connection, contains, contents, copy, corr, corresponding, create,
cross, date, dec, decimal, default, delete, desc, describe,
dictionary, disconnect, distinct, double, drop, else, end, engname,
engopt, eq, except, execute, exists, false, float, for, format, from,
full, ge, grant, group, gt, having, in, index, indexes, informat,
inner, insert, int, integer, intersect, into, is, join, label, le,
leading, left, libref, like, load, lower, lt, match, missing, modify,
natural, ne, no, not, notin, null, num, numeric, on, operation, option,
or, order, outer, overlaps, partial, precision, privileges, public,
real, references, reset, restrict, revoke, right, select, set,
smallint, some, table, then, to, trailing, trim, true, union, unique,
unknown, update, upper, using, validate, values, varchar, verbose,
view, when, where, with, without, yes
```

SQL Statements

Alter Table Statement

The Alter table statement changes a table definition.

```
alter table statement ::=
    ALTER TABLE table spec
    { { ADD|MODIFY|ALTER [ COLUMN ] column def list } |
      { DROP [ COLUMN ] column name list }
    }+ ';'
```

Connect Statement

The Connect statement creates a Pass-Through connection.

```
connect statement ::=
    CONNECT TO libref name [ [ AS ]
    alias name ] '('
    dbms options ')' ] ';'
```

Create Index Statement

The Create Index statement creates an index on a table.

```
create index statement ::=
    CREATE [ UNIQUE ] INDEX index name ON
    table spec '(' column name list ')' ';'
```

Create Table Statement

The Create Table statement creates a table definition.

```
create table statement ::=
    CREATE TABLE table spec
    { '(' column def list ')' | AS
select spec | LIKE
table spec } ';'
```

Create View Statement

The Create View statement creates a view upon one or more tables.

```
create view statement ::= CREATE VIEW
table spec AS
select spec ';'
```

Delete Statement

The Delete statement deletes records.

```
delete statement ::= DELETE FROM
table spec [
where clause ] ';'
```

Describe Table Statement

The Describe Table statement describes a table definition.

```
describe table statement ::=
    DESCRIBE TABLE table spec [ [','] 
    table spec ]* ';'
```

Describe View Statement

The Describe View statement describes a view definition.

```
describe view statement ::=
    DESCRIBE VIEW table spec [ [',']
    table spec ]* ';'
```

Disconnect Statement

The Disconnect statement is a Pass-Through statement.

```
disconnect statement ::= DISCONNECT FROM
libref name ';'
```

Drop Index Statement

The Drop Index statement drops an index from a table.

```
drop index statement ::=
    DROP INDEX index name [ [','] 
    index name ]* FROM
    table spec ';'
```

Drop Table Statement

The Drop Table statement drops a table definition.

```
drop table statement ::= DROP TABLE
table spec [ [',']
table spec ]* ';'
```

Drop View Statement

The Drop View statement drops a view definition.

```
drop view statement ::=
    DROP VIEW table spec [ [',']
    table spec ]* ';'
```

Execute Statement

The Execute statement is a Pass-Through statement.

```
execute statement ::= EXECUTE '('
passthru spec ')' BY
libref name ';'
```

Insert Statement

The Insert statement adds records.

```
insert statement ::=
    INSERT INTO table spec [ '('
    column name list ')' ]
    insert source ';'
```

Reset Statement

The Reset statement resets session options and flags.

```
set option statement ::=
    { SET OPTION  | RESET }
    { identifier
 [ '=' { constant |
identifier |
truth value
| DEFAULT } ] }+
```

Select Statement

The Select statement retrieves information.

```
select statement::=
select spec ';'
```

Update Statement

The Update statement updates records.

```
update statement ::=
    UPDATE table spec
    SET column name '='
scalar expr [ ','
column name '='
scalar expr ]*
    [ where clause ] ';'
```

Validate Statement

The Validate statement validates a given select specification.

```
validate statement ::= VALIDATE
select spec ';'
```

NEW SQL Statements

Async Operation Statement

```
async operation statements ::= { BEGIN | END } ASYNC OPERATION ';'
```

Copy Table Statement

```
copy table statement ::=
    COPY TABLE table spec FROM
    table spec [ WITHOUT INDEXES ] [ORDER BY
    column name
    [ ASC | DESC ] [','
    column name [ ASC | DESC ]]] ';'
```

Create Materialized View Statement

```
create materialized view statement ::= CREATE MATERIALIZED VIEW
table spec AS
select spec ';'
```

Libref Statement

```
libref statement ::=
    LIBREF libref name [ ENGNAME '='
    identifier ] [ ENGOPT '='
    string ] ';'
```

Load Table Statement

```
load table statement ::=
    LOAD TABLE table spec [ WITH
    with index spec [ ','
    with index spec ]* ]
    AS select spec ';'
```

SQL Building Blocks

Alias Name

```
alias name ::=
identifier
```

Atomic Expression

```
atomic expr ::=
constant |
column spec
```

Between Predicate

```
between pred ::=
scalar expr [ NOT ] BETWEEN
scalar expr AND
scalar expr
```

Boolean Expression

```
boolean expr ::=
  | [ NOT ] { predicate | '('
  boolean expr ')' } [ IS [ NOT ]
  truth value ]
  | boolean expr { AND | OR }
  boolean expr
```

Case Expression

```
case expr ::=
    CASE { WHEN boolean expr THEN
    scalar expr }+ [ ELSE
    scalar expr ] END
  | CASE scalar expr { WHEN
  scalar expr THEN
  scalar expr }+ [ ELSE
  scalar expr ] END
```

Column Definition

```
column def ::=
column name
data type [
column modifier ]* [ NOT NULL ]
```

Column Definition List

```
column def list ::=
column def [ ','
column def ]*
```

Column Modifier

```
column modifier ::=
    FORMAT '=' <quoted or nonquoted SAS format specification>
  | LABEL '=' string
```

Column Name

```
column name ::= identifier
```

Column Name List

```
column name list ::=
column name [ [','] 
column name ]*
```

Column Specifications

```
column spec ::=
    [ CALCULATED ] column name
  | table alias'.'
  column name
```

Comparative Operators

```
comp operator ::=
 | EQ | '='
 | NE | '^=' | '~=' | '!=' | '<>'
 | LT | '<'
 | GT | '>'
 | LE | '<='
 | GE | '>='
```

Comparison Predicates

```
comparison pred ::=
scalar expr {
```

```
comp operator
scalar expr }+
```

Connection String

```
connection string ::= <user-defined
string delimited by ending/matching parenthesis>
```

Constant

```
constant ::=
  | number | missing value
  | string | date/time string
  | NULL
```

Contains Predicate

```
contains pred ::=
scalar expr { CONTAINS | '?' }
scalar expr
```

Data Types

```
data type ::=
    { CHAR[ACTER] | VARCHAR } [ '('unsigned ')' ]
  | { INT[EGER] | SMALLINT }
  | { NUM[ERIC] | DEC[IMAL] | FLOAT }
  [ '(' unsigned [ ',' unsigned ] ')' ]
  | REAL | DOUBLE PRECISION | DATE
```

Date / Time String

```
date/time string ::=
string{D|T|DT}
```

DBMS Options

```
dbms options ::= <user-defined
string delimited by ending/matching parenthesis>
```

Digits (Numeric)

```
digit ::= '0' <through> '9'
```

Exists Predicate

```
exists pred ::= EXISTS subquery
```

Function Arguments

```
function args ::=
    scalar expr [ ',' scalar expr ]* | DISTINCT scalar expr | [ DISTINCT ] '*'
```

Function Expressions

```
function expr ::=
func name '('
function args ')'
```

Function Name

```
function name ::=
identifier
```

Identifier

```
identifier ::= ['\']{
letter|<underscore>}{
```

```
letter|
digit|<underscore>}*
```

In Predicate

```
in pred ::=
    scalar expr { [ NOT ] IN | NOTIN } {
    subquery | '('
    constant [ ','
    constant ]* ')' }
```

Index Name

```
index name ::=
identifier
```

Insert Set List

```
insert set list ::= SET
set value list [ SET
set value list ]*
```

Insert Source

```
insert source ::=
  | insert values list
  | insert set list
  | query expr
```

Insert Value

```
insert value ::= VALUES '('
scalar expr [ ','
scalar expr ]* ')'
```

Insert Values List

```
insert values list ::=
insert value [
insert value ]*
```

Letter (Alpha)

```
letter ::= 'a' <through> 'z' <or> 'A' <through> 'Z'
```

Libref Name

```
libref name ::=
identifier
```

LIKE Predicate

```
like pred ::=
scalar expr [ NOT ] LIKE
scalar expr
```

Missing Value

```
missing value ::= '.'[
letter]
```

Null Predicate

```
null pred ::=
scalar expr IS [ NOT ] { NULL | MISSING }
```

Number

```
number ::=
    {unsigned|{
    digit}+'.'[{
    digit}+]|'.'{
    digit}+}[{'e'|'E'}['+'|'-']{
    digit}+]
```

ORDER BY Clause

```
order by clause ::=
    ORDER BY atomic expr [ ASC | DESC ] [ ','
    atomic expr [ ASC | DESC ] ]*
```

Pass-Through Specification

```
passthru spec ::=
   <database-specific string delimited by ending/matching parenthesis>
```

Predicate Types

```
predicate ::=
  | comparison pred
  | between pred
  | in pred
  | like pred
  | null pred
  | quantified comparison pred
  | exists pred
  | contains pred
  | soundslike pred
```

Quantified Comparison Predicate

```
quantified comparison pred ::=
    scalar expr
    comp operator { ALL | SOME | ANY }
    subquery
```

Query Expression

```
query expr ::=
    query spec
  | query expr { [ OUTER ] UNION | EXCEPT | INTERSECT } [ CORRESPONDING ] [ ALL ]
  query expr
```

Query Specification

```
query spec ::=
    SELECT [ DISTINCT | UNIQUE ] select item [ ','
    select item ]*
    FROM table ref [ ','
    table ref ]*
    [ WHERE boolean expr ]
    [ GROUP BY scalar expr [ ','
    scalar expr ]* ]
    [ HAVING boolean expr ]
```

Scalar Expression

```
scalar expr ::=
  | atomic expr
  | function expr
  | '(' scalar expr ')'
  | subquery
  | scalar expr { '+' | '-' | '*' | '/' | '||' | '**' }
scalar expr
  | { '+' | '-' } scalar expr
  | case expr
```

Select Item

```
select item ::=
    '*'
  | identifier'.*'
  | scalar expr [ [ AS ]
  identifier ] [
  column modifier ]*
```

Select Specification

```
select spec ::=
query expr [
order by clause ]
```

Set Value List

```
set value list ::=
column name '='
scalar expr [ ','
column name '='
scalar expr ]*
```

Soundslike Predicate

```
soundslike pred ::=
scalar expr '=*'
scalar expr
```

String

```
string ::=
<a single- or double-quoted
literal string -- see Strings>
```

Subquery

```
subquery ::= '('
query expr
```

Table Alias

```
table alias ::=
identifier
```

Table Join

```
table join ::=
    table ref [ INNER | { LEFT | RIGHT | FULL }
    [ OUTER ] ] JOIN table ref
    { ON boolean expr | USING '('
column name list ')' }
  | '(' table join ')'
```

Table Name

```
table name ::=
identifier
```

Table Options

```
table options ::= <user-defined
 string delimited by ending/matching bracket>
```

Table Reference

```
table ref ::=
    table spec [ [ AS ]
    identifier ]
  | subquery [ [ AS ]
  identifier ] [ '('
  column name list ')' ]
  | CONNECTION TO identifier '('
  connection string ')' [ [ AS ]
  identifier ]
  | table join
```

Table Specification

```
table spec ::=
  | table name [ '['
  table options ']' ]
  | libref name'.'
  table name [ '['
  table options ']' ]
```

Truth Value

```
truth value ::= { TRUE | YES } | { FALSE | NO }
```

Unsigned

```
unsigned ::= {
digit }+
```

WHERE Clause

```
where clause ::= WHERE
boolean expr
```

With Index Specification

```
with index spec ::= [ UNIQUE ] INDEX
index name ON '('
column name list ')'
```

Chapter 10

SAS Scalable Performance Data (SPD) Server SQL Access Library API Reference

Introduction

This chapter describes the Scalable Performance Data Server SQL access library API (Application Programming Interface) and provides some simple examples. This chapter refers to the Scalable Performance Data Server SQL access library as SPQL. Read this chapter if you want a library that provides a C-language compatible interface to write user applications to access an SPD Server SQL server. Because the library was designed for multi-threaded applications, the code is thread safe.

Overview of SPQL Usage

SPQL enables you to write application: programs that can connect to and access Scalable Performance Data Server (SPD Server) hosts using the SQL language. SPQL is oriented toward connection, allowing you to submit SQL statements to one or more SPD Server SQL servers that execute SQL statements on your behalf.

SPQL API Description

The C-language H file spql.h is provided for customer-written applications. This chapter describes the API functions, their use, and restrictions.

SPQL API Functions

The SPQL API functions include

spqlinit()

Initializes the SPQL library for operation.

```
int spqlinit(void)
```

Usage: Performs a one-time initialization which enables the SPQL library to function. For this reason, you must call spqlinit() at least once to activate an SPQL program. Do not make other SPQL API calls before calling this function. If you do, the results are unpredictable. When spqlinit() successfully completes, you can safely proceed to use the SPQL API in a multi-threaded context.

Note: Spqlinit()-is not a thread-safe function. Call it only within a single-threaded context in your application. Alternatively, call it within an application-controlled mutex region.

Parameters: None

Returns: 0 if successful; SPQL_INITFAILED if the initialization fails.

spqlterm()

Is the termination counterpart of the spqlinit() function.

```
int spqlterm(void)
```

Usage: Terminates the SPQL library session, disconnecting all active SPD Server SQL server connections and freeing up the memory resources associated with the SPQL run-time library executables.

Parameters: None

Returns: 0 if successful.

spqlconnect()

Establishes a connection to a specified SPD Server SQL server.

```
int spqlconnect(char *constr, void **contok)
```

Usage: Establishes a connection to the SPD Server SQL server. The **constr** parameter specifies all the connection information needed to establish the connection to an SPD Server SQL server. When a connection is made successfully, a connection, token (**contok**) is returned to the caller.

Parameters:

char *constr
: A null-terminated string identifying the SPD Server SQL server to connect to for this session. The syntax for the string is identical to that used for the SAS PROC SQL Pass-Through CONNECT statement.

void **contok
: Returns a connection token if the connection successfully completes. You must retain the token; use it in subsequent SPQL library operations that you perform using the connection.

Returns: 0 if successful; SPQL_NOMEM if unable to allocate memory for the connection token; SPQL_CONFAILED if unable to connect successfully to the SPD Server SQL server.

spqldisconnect()

Terminates a connection from the SPD Server SQL server specified with a spqlconnect().

```
int spqldisconnect(void *contok)
```

Usage: Disconnects from a specified SPD Server SQL server. The caller passes the connection token which was returned from an **spqlconnect()** call. Then, the SPD Server SQL server associated with the connection is disconnected from the caller, and the memory associated with connection token is returned to the system.

Parameters:

void *contok
: Connection token previously obtained from **spqlconnect()**.

Returns: 0 if successful.

spqlperform()

Submits an SQL statement for execution on a given connection.

```
   int spqlperform(void *contok, char *stmtbuf, int stmtlen,
                   int *actions, void **stmttok);
```

Usage: Performs specified SQL statement and informs caller of the results. The **actions** parameter returns a value of 0 if no additional action is required. If actions are required to complete the statement, one or more of the following bit flags are returned.

```
   Flag          Action
   ----------    ---------------
```

```
SPQLDATA      Data is returned(see spqlfetch())
SPQLCOLINFO   Column information is returned(see spqlcolinfo())
```

Parameters:

void *contok
The connection used to execute the SQL statement.

char *stmtbuf
A buffer that holds the SQL statement to perform.

int stmtlen
The length of the SQL statement in buffer; -1 if null-terminated.

int *actions
Returns post-processing notification bit flags.

void **stmttok
Returns a statement token to use in post-processing the SQL statement results. See post-processing action definitions for use of statement token.

Returns: 0 if the SQL statement is successfully prepared or executed; SPQL_BADSTMT if the SQL statement specified in the statement buffer is prepared incorrectly; SPQL_NOMEM if **spqlperform** cannot allocate memory for the statement token.

spqlfreestok()

Generates a free statement token from spqlperform().

```
int spqlfreestok(void *stmttok);
```

Usage: Free resources used for the statement token from **spqlperform()**. Call **spqlfreestok()** after the data or information from the statement token has been extracted. You can call this function before all selected rows from the spqlperform() are read. If you do, the remaining unread rows (from the previous select) are discarded.

Parameters:

void *stmttok
Statement token to free

Returns: 0 if successful.

spqltabinfo()

Gets table information from a statement token.

```
int spqltabinfo(void *stmttok, spqltinfo_t **tinfo)
```

Usage: Interrogates the statement token for table information. Upon return of the call, updates **tinfo** with the pointer to the spqltinfo_t structure in the statement.

Note: Treat the structure accessed by the returned pointer as read-only memory.

Parameters:

void *stmttok
The statement token to use to access table information from a 'select'.

spqltinfo **tinfo
Returns pointer to **spqltinfo_t** structure into the statement token memory.

Returns: 0 for successful completion.

spqlcolinfo()

Gets column information from a statement token.

```
int spqlcolinfo(void *stmttok, int *ncols, spqlcinfo_t **colvec)
```

Usage: Interrogates token for column information. Upon return of the call, updates **ncols** with the column count selected in the statement and updates **colvec** with the pointer to the vector of **spqlcol_t** structures in the statement.

Note: Treat structures accessed by the returned pointer as read-only memory.

Parameters:

void *stmttok
    The statement token to use to access column information from 'select'.

int *ncols
    Returns in the statement token the number of columns selected.

spqlcinfo **colvec
    Returns in the statement token a pointer to the array of **spqlcinfo_t** structures.

Returns:0 if successful.

spqlfetch()

Gets row data from a statement token.

```
int spqlfetch(void *stmttok, void **bufptr, int *bufsize)
```

Usage: Fetches the rows returned from executing a statement. Each call to spqlfetch returns a row from a statement to the caller's buffer. If **bufptr** contains a NULL value, the routine returns a pointer to a buffer containing the next row. If the value is not NULL, it assumes that the buffer is owned by the caller and returns the data to the caller's buffer. In either case, **bufsize** is updated with the row length returned. Callers that use locate-mode spqlfetch semantics (that is, who specify **bufptr** as NULL), should NEVER FREE the memory pointer returned by spqlfetch. A call to spqlfetch(), after all rows for the statement are returned, returns a **bufsize** of 0.

Parameters:

void *stmttok
    The statement token to use to access row data from 'select'.

void **bufptr
    Contains a pointer to the caller's row buffer to fill with row data. If it is NULL on entry, it returns a pointer to the internal statement buffer.

int *bufsize
    Returns the size of the row buffer that was returned to the caller.

Returns: 0 if successful; SPQL_ENDDATA if the statement has no more rows to return; SPQL_FETCHFAILED if there is an unexpected failure while fetching the next row buffer.

spqlgmsg()

Accesses thread-specific error or diagnostic message buffer contents.

```
int spqlgmsg(char **mbuf)
```

Usage: Returns a pointer to the threads error or diagnostic message buffer. Call **spqlgmsg()** to get any diagnostic messages if you encounter an error executing an SPQL function. If there is message information, spqlgmsg() returns the message pointer in the **mbuf** parameter as well as the length of the message (the function return value).

Parameters:

char **mbuf
: Returns a pointer to the thread's error or diagnostic message buffer. If mbuf is NULL, there is no message information. The call also returns the length of the thread's error or diagnostic message buffer. A 0 indicates that no message exists.

SPQL Function Return Codes

Some SPQL functions generate return codes, allowing you to check the value and take appropriate action in your application code. Typically, the application action taken upon receiving an error code is a call to spqlgmsg() to get the contents of the diagnostic buffer. The program can then display the buffer's contents to the user or write the contents to a log. The return codes in this section are classified by their state: **positive** [(WARNING), (SUCCESS)] or **negative** [(ERROR)].

SPQL_SUCCESS(==0)

Successful completion of the SPQL function call.

SPQL_ENDDATA(WARNING)

All rows selected were read from the statement token.

SPQL_INITFAILED(ERROR)

Initialization failure. (It is unsafe for your application to make additional SPQL calls if this error occurs.)

SPQL_NOMEM

Unable to allocate memory for some type of SPQL data structure. Check the diagnostic buffer for details.

SPQL_CONFAILED(ERROR)

Unable to make a connection to an SPD Server SQL server. Check the diagnostic buffer for details.

SPQL_BADSTMT(ERROR)

SQL statement is incorrectly formatted for submission to sqlprepare(). Either the statement is blank (all white space) or contains contiguous non-white space characters.

Part 5

SPD Server Reference

Chapter 11

Optimizing SPD Server Performance

SPD Server Performance and Usage Tips

SPD Server gives good performance when run using default configuration settings. To realize the full benefits of SPD Server's design and capabilities, you must configure some of the software's options to modify the default behaviors. The configuration changes will depend on the computing environment, table size and complexity, and indexing structures.

You use SAS macro variables that are specific to SPD Server and SAS statement options (LIBNAME options and table options) to configure SPD Server for optimum performance.

Symmetric Multiple Processor (SMP) Utilization

A cornerstone of SPD Server's power is the ability to perform parallel processing. Parallel processing uses multiple processors to execute more than one set of instructions, or threads, concurrently. SPD Server is oriented to exploit parallelism whenever it can improve transaction times and processor utilization.

A fundamental question about parallelism is whether using additional CPUs on a specific problem will deliver data faster. Extra CPUs do not guarantee faster results every time. The amount of CPU-intensive work that a thread must do needs to last long enough to justify the cost of the thread. The cost of the thread is creating it, managing it, and interacting with other threads involved in the same parallel algorithm.

If not properly matched to the workload, the parallel algorithm can use more CPU time without reducing data delivery time. Additional threads can create conflicting demands for critical system resources such as physical memory. Excessive execution times can occur if too many threads attempt to access a large table at the same time, because many threads demand large amounts of physical memory. Extreme resource constraints can result in slower overall processing.

SPD Server focuses on the following areas to speed overall processing using parallelism:

- User-definable parallel execution blocks for SQL Pass-Through statements
- Parallel aggregation for common summary functions when performing SELECT [...] GROUP BY statements
- WHERE clause evaluation for indexed and non-indexed strategies
- Overlapped table and concurrent index updates when appending to tables
- Index creation when creating multiple indexes
- Optimize PROC SORT BY clauses
- Pipelined read-ahead when concurrently accessing multiple tables

File System Performance Concepts

Overview of File System Performance

SPD Server uses several file types in its data storage model. Data objects in SPD Server consist of one or more component files. Each component file is itself a collection of one or more disk files. These are called the partitions of the component.

Component files create partitions when any of the following conditions is true:

- The current partition exceeds the user-specified PARTSIZE= value: Subsequent partitions are allocated in cyclical fashion across the set of directories that are specified in the DATAPATH= statement for the LIBNAME domain. Partitioning uses file-level striping to create PARTSIZE-sized files that complement disk-level striping that your operating system's volume manager software creates. SPD Server uses a default PARTSIZE= setting of 16 MB. PARTSIZE= determines a unit of work for parallel operations that require full table scans. Examples of parallel operations that require full table scans are WHERE clause evaluation and SQL GROUP-BY summarization. Trade-offs are balancing increased numbers of files used to store the table versus the work savings realized through parallel partitions. Extra partitions means that files are opened to process a table, but with fewer rows in each partition.
- The current partition exceeds the RLIMIT_FILESIZE value: In UNIX systems, RLIMIT_FILESIZE is a system parameter that defines the maximum size of a single disk file. In Windows, SPD Server uses a default RLIMIT_FILESIZE value of 2 GB.
- The current partition exceeds the space on the file system where it has been created.

Defining Directories

SPD Server allows the user to define a set of directories that contain component files and their partitions. Normally, a single directory path is constrained by some volume limit for the file system, or the maximum amount of disk space that the operating system understands.

Most UNIX and Windows systems offer a volume manager utility. You can use volume manager utilities to create file systems (volumes) that are greater than the available space on a single disk. System administrators can use these utilities to create large, multi-gigabyte volumes. These volumes can be spread across a number of disk partitions, or even span multiple disk devices. Volume manager utilities generally support creation of disk volumes that implement one of the common RAID (redundant arrays of inexpensive disks) configuration levels.

Disk Striping

A defining feature of all RAID levels is disk striping. Striping organizes the linear address space of a volume into pieces that are spread across a collection of disk drive partitions. For example, a user can configure a volume across two 1 GB partitions on separate disk drives A and B with a stripe size of 64K bytes. 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
- use multiple kernel threads to drive a block of I/O

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

RAID Levels

The following is a brief summary of RAID levels relevant to SPD Server:

RAID-0
: High performance with low availability. Physically losing a disk means that data is lost. No redundancy exists to recover volume stripes on a failed disk.

RAID-1
: Disk mirroring for high availability. Every block is duplicated on another mirror disk, sometimes referred to as shadowing. In the event one disk is lost, the mirror disk is still likely to be intact, preserving the data. RAID-1 can also improve read performance since a device driver has two potential sources for the same data. The system can choose the drive that has the least load or latency at a given point in time. The down side to RAID-1: it requires twice the number of disk drives as RAID-0 to store a given amount of data.

RAID-5
: High performance and high 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. RAID-5's downside is resource utilization; RAID-5 requires extra CPU cycles and extra disk space to transform and manage data using the ECC model.

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 in a configuration known as RAID-1+0 (sometimes referred to as RAID-10). Advantages are the same as for RAID-1 and RAID-0. The only disadvantage is the requirement for twice as much disk as the pure RAID-0 solution. Generally, this configuration tends to be a top performer if you have the disk resources to pursue it.

Regardless of RAID level, disk volumes should be hardware striped when using the SPD Server software. This is a significant way to improve performance. Without hardware striping, I/O will bottleneck and constrain SPD Server performance.

Transient Storage

You should configure a RAID-0 volume for WORKPATH= storage for your SPD Server. When sizing this RAID-0 volume, keep in mind that the WORKPATH= that you set up a given SPD Server host must be shared by all of its SQL and LIBNAME proxy processes that exist at a given point in time. The SPD Server Frequently Asked Questions (FAQ) is a good source of information about estimating disk space requirements for WORKPATH=.

Consider using one or more RAID-0 volumes to locate the database domains that will support TEMP=YES LIBNAME assignments. This LIBNAME statement option creates a temporary storage domain that exists only for the duration of the LIBNAME assignment. This is the SPD Server equivalent of the SAS WORK library. All data objects (tables, catalogs, utility files) that are created in the TEMP=YES temporary domain are automatically deleted when you end the SAS session.

LIBNAME Domains

LIBNAME domains define the primary directory path and can, if desired, define other directories for placing the data and index components of SPD Server tables. The METAPATH=, DATAPATH=, and INDEXPATH= LIBNAME definition options determine the placement of SPD Server's component and partition files.

Data and Index Separation

The section on “File System Performance Concepts” on page 175 discussed how distributing I/O load across different disk drives can improve performance. Further load distribution can be achieved by separating data and index components of SPD Server tables. To do this, use the DATAPATH= and INDEXPATH= options when configuring LIBNAME domains.

For example, when performing complex WHERE clause evaluations, multiple threads are active on index component files and the data component file at the same time. Splitting the index and data file components onto different volumes can improve performance by reducing disk contention and increasing the level of parallelism down to the disk access level.

A word of caution when using DATAPATH= and INDEXPATH= options to distribute the data and index components: take extra care when performing and restoring disk backups of SPD Server tables using a system backup and restore utility. When making a backup, ensure that the metadata, data, and index component partition files are of the same generation and are in their respective directories.

When restoring a backup, restore the component partitions to the same directories where they were created. To avoid this restore problem, create symbolic links with the original directory path that point to the restore directories. Of course, if the components are not separated using the path options, this restore issue does not apply.

The backup and restore issues are not an issue when using the SPD Server Backup and Restore Utilities. These utilities resolve any component files when backing up or restoring tables. For more information, see Chapter 19, "SPD Server Backup and Restore Utilities" in the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide*.

Configuring a LIBNAME Domain

Suppose a user has four volumes designated. Volumes exist for (1) SPD Server metadata, (2) data components, (3) index components, and (4) proxy working storage, as follows

- **/dmart_domain** is a 4 GB volume
- **/dmart_data** is a 40 GB volume
- **/dmart_index** is a 40 GB volume
- **/spds_work** is a 10 GB volume

The user wants to configure a LIBNAME domain called **dmart** to use **/dmart_domain** for the primary directory, with data components going to **/dmart_data**, and index components going to **/dmart_index**. The **/spds_work** volume should be configured for proxy working storage.

The configuration is made in two steps:

1. In the server parameter file (-parmfile) enter the following line:

   ```
   WORKPATH=/spds_work;
   ```

2. In the SPD Server LIBNAME file (-libnamefile) enter the following domain definition:

   ```
   ibname=dmart
      path=/dmart_domain
      roptions="datapath=('/dmart_data')
      indexpath=('/dmart_index')";
   ```

Loading Data into an SPD Server Host

SPD Server's emphasis on complete LIBNAME compatibility means that when you access SPD Server, the standard procedures used to create tables in SAS apply to SPD Server tables as well.

Using SAS, you can load data into SPD Server tables using DATA step programs, PROC COPY or PROC APPEND, and SCL applications. You can also use SQL Pass-Through to load SPD Server tables. The SPD Server SQL extensions for the LOAD TABLE and COPY TABLE statements provide further support.

Use LOAD TABLE to load a table from the projected columns of an SQL SELECT statement and create indexes, all in a single pass. LOAD TABLE exploits multi-thread table I/O and index creation. The multi-thread table I/O and index creation overlaps with the SELECT statement that extracts the data from its source tables.

Use COPY TABLE to copy an existing SPD Server table to a new table and include indexes as part of the copy operation. It offers the same parallel table and index I/O and overlapped input as the LOAD TABLE command.

The COPY TABLE and LOAD TABLE statements work only for source and target tables on the local machine.

Table Loading Techniques

The SAS data storage model adds rows to a data set one at a time. The SPD Server I/O engine buffers rows to be added from the SAS application and performs block adds using a highly efficient pipelined append protocol when communicating with the proxy.

Parallel Table Load Technique Using PROC APPEND

To achieve significant improvements in building a table, create the empty table first, defining indexes on the desired columns. Then, use PROC APPEND to populate the table and indexes. The example below demonstrates this technique.

```
/* Create an empty SPD Server table with the same   */
/* columns and column attributes as the existing    */
/* SAS table.                                       */
data spdslib.cars;
set somelib.cars(obs=0);
run;

/* Create indexes for the empty table so the indexes */
/* are appended in parallel with the table appends.  */

PROC DATASETS lib=spdslib;
   modify cars;
   index create make;
   index create origin;
   index create mpg;
quit;

/* PROC APPEND SAS table Cars to SPD Server table  */
/* Cars. The append to the SPD Server table and    */
/* its indexes will occur in parallel.             */

PROC APPEND
   base=spdslib.cars
   data=somelib.cars;
run;
```

Parallel Table Load Technique Using SQL Pass-Through

If you are using SQL Pass-Through, consider using the LOAD TABLE command to perform the same operation. LOAD TABLE encapsulates the sequence of SAS DATA and PROC steps into an even more powerful technique for gaining maximum performance when loading a new table. The following example demonstrates the same table construction using LOAD TABLE and SQL Pass-Through:

```
/* Create a copy of the SPD Server table Cars and   */
/* its index from Example 1 to another SPD Server   */
/* table carload using Pass-Through LOAD command.   */
/* The table creation of the SPD Server table       */
/* carload and its indexes will occur in parallel.  */
```

```
execute(
load table carload with
   index make
     on (make),
   index origin
     on (origin),
   index mpg
     on (mpg)
   as select *
   from cars
) by sasspds;
```

Parallel Pass-Through Table Load and Data Subset

```
/* Create a subset of the SPD Server table Cars  */
/* from Example 1 to another SPD Server table    */
/* Fordcar using the Pass-Through LOAD command.  */
/* The table creation of the SPD Server table    */
/* Fordcar and its indexes occurs in parallel.   */

execute(
load table fordcar with
   index origin
     on (origin),
   index mpg
     on (mpg)
   as select *
   from cars
   where make="ford"
) by sasspds;
```

Parallel Pass-Through Table Copy

```
/* Create a copy of the SPD Server table Cars and  */
/* all its indexes from Example 1 to another Data  */
/* Server table Copycars using the Pass-Through    */
/* COPY command.  The table creation of the Data   */
/* Server table Copycars and its indexes will      */
/* occur in parallel.                              */

execute(
copy table copycars
  from cars
) by sasspds;
```

Loading Indexes in Parallel

A significant strength of SPD Server is efficient creation, maintenance, and use of table indexes. Indexing can greatly speed the evaluation of WHERE clause queries. The index can also be a source of sort order when performing BY clause processing. The index is also used directly by some SAS applications. For example, PROC SQL uses indexes to efficiently evaluate equijoins.

Parallel Index Creation

SPD Server supports parallel index creation using asynchronous index options. To enable asynchronous parallel index creation, either submit the SPDSIASY=YES macro variable before creating an index in SAS, or use the ASYNCINDEX=YES table option.

Both the macro variable and the table option apply to the DATA step INDEX= processing as well as to PROC DATASETS INDEX CREATE commands. Either method allows all of the declared indexes to be populated with a single scan of the table. A single scan is a substantial improvement over making multiple passes through the data to build each index serially.

As always, there is a price for parallelism. To create multiple indexes requires enough WORKPATH= disk space to create all of the key sorts at the same time. The PROC DATASETS structure has the flexibility to allow batched parallel index creation by using multiple MODIFY groups. The Parallel Index Creation example below inserts INDEX CREATE statements between two successive MODIFY statements resulting in a parallel creation group.

Parallel Index Creation Example

```
DATA foo.patient_info;
   length
     last_name $10
     first_name $20
     patient_class $2
     patient_sex $1;

   patient_no=10;
     last_name="Doe";
     first_name="John";
     patient_class="XY";
     patient_age=33;
     patient_sex="M";

run;

%let spdsiasy=YES;
PROC DATASETS lib=foo;
   modify patient_info;
      index create
        patient_no
        patient_class;
   modify patient_info;
      index create
        last_name
        first_name;
   modify patient_info;
      index create
        whole_name=(last_name first_name)
        class_sex=(patient_class patient_sex);
  quit;
```

Indexes for PATIENT_NO and PATIENT_CLASS are created in parallel, indexes for LAST_NAME and FIRST_NAME are created in parallel, and indexes for WHOLE_NAME and CLASS_SEX are created in parallel.

Parallel Index Updates

SPD Server also supports parallel index updates during table append operations. Multiple threads enable overlap of data transfer to the proxy, as well as updates of the data store and index files. SPD Server decomposes table append operations into a set of steps that can be performed in parallel. The level of parallelism attained depends on the number of indexes that are present on the table. The more indexes you have, the greater the exploitation of parallelism during the append processing. As with parallel index creation, parallel index updates use WORKPATH= disk space for the key sorts that are part of the index append processing.

Truncating Tables

The Truncate command is a PROC SPDO command that allows the deletion of all rows in a table without deleting the table structure or metadata. The PROC SPDO truncate command is shaded for emphasis in the code example below.

```
%let host=kaboom ;
%let port=5191 ;
%let domain=path2 ;

LIBNAME &domain sasspds "&domain"
   server=&host..&port
   user='anonymous'
   ip=YES ;
/* create a table */
data &domain..staceys_table ;

   do i = 1 to 100 ;
   output ;
end ;
run ;

* verify the contents of the created table */

PROC CONTENTS data=&domain..staceys_table ;
run ;

/* SPDO Truncate command deletes the table  */
/* data but leaves the table structure in   */
/* place so new data can be appended        */
PROC SPDO lib=&domain ;
set acluser ;
Truncate staceys_table ;

quit ;
```

```
* verify that no rows or data remain in    */
/* the structure of staceys_table           */
PROC CONTENTS data=&domain..staceys_table ;
run ;
```

Optimizing WHERE clauses

Overview of Optimizing WHERE Clauses

SPD Server includes more advanced methods to optimize WHERE clauses. Before SPD Server 4.0, the rule-based, heuristic WHERE clause planner WHINIT was used to manually tune queries for performance. SPD Server provides dynamic WHERE clause costing, an automatic feature that can replace the need to manually tune queries. SPD Server dynamic WHERE-costing uses factors of duplicity and distribution to calculate relative processor costs of various WHERE clause options. SPD Server users can use server parameter commands in the **spdsserv.parm** file or macro variables to turn dynamic WHERE-costing on and off. If dynamic WHERE-costing is turned off, SPD Server reverts to using the rules-based WHERE clause planner.

WHERE Clause Definitions and Terminology

- **WHERE clauses** are selection criteria for a query that specify one or more Boolean predicates. Implementing the criteria, SPD Server selects only records that satisfy the WHERE clause.
- **Predicates** are the building blocks of WHERE clauses. Use them stand-alone or combine them with the operators AND and OR to form complex WHERE clauses. An example of a WHERE clause is

  ```
  "where x > 1 and y in (1 2 3)"
  ```

  In this example, there are two predicates, **x > 1** and **y in (1 2 3)**. You specify the negative of a predicate by using not. For example, **where x > 1 and not (y in (1 2 3))**.
- **Boolean logic** determines whether two predicates, joined with an AND or OR, are true (satisfies) or false (does not satisfy) the specification. The AND operator requires that all predicates be true for the entire expression to be true. For example, the expression p1 AND p2 AND p3, is true only if all three predicates (p1, p2, andp3) are true. In contrast, the OR operator requires only one predicate to be true for the entire expression to be true.

  For the WHERE clause (x < 5 or y in (1 2 3)) and z = 10, the following truth table describes the overall result (truth):

  ```
  "x < 5 ?"    "y in (1 2 3) ?"    "z = 10 ?"    Result
  =========    ================    ==========    ======
    False          False            False        False
    False          False            True         False
    False          True             False        False
    False          True             True         True
    True           False            False        False
    True           False            True         True
    True           True             False        False
    True           True             True         True
  ```

- **Indexes** are structures associated with tables that permit SPD Server to quickly access records that satisfy an indexed predicate. In an example WHERE clause, where x = 10 and y > 11, SPD Server selects the best index on column x to directly retrieve records that have a value of 10 in the x column. If no index exists for x, SPD Server must sequentially read each record in the table searching for x equal to 10.
- **Simple and composite indexes:** Simple indexes index a single column; composite indexes index two or more columns. The list of column(s) in an index is sometimes called the index key.
- **Parallelism** is the SPD Server capability that enables multiple threads to execute in parallel. Using multiple processors in parallel mode is sometimes called 'divide and conquer' processing. SPD Server uses parallelism to evaluate the multiple indexes that are involved in more complicated WHERE clauses.

SPD Server Indexing

Overview of Server Indexing

SPD Server tables can have one or more indexes. There is a combination of four different indexing strategies a table can use, and the choice depends on the data populating the table, the size of the table, and the types of queries that will be executed against the table.

SPD Server indexing evaluates the processor cost of a WHERE clause. The section "WHERE-Costing Using Duplicity and Distribution Values" on page 187 shows how factors of duplicity and distribution are used to choose the evaluation strategy that will perform the WHERE clause at the smallest processor cost. The five evaluation strategies that the WHERE clause planner uses are EVAL 1, EVAL 2, EVAL 3, EVAL 4, and EVAL 5. The different EVAL strategies calculate the number of rows that will be required to execute a given query.

True rows are rows that contain the variable values specified in a WHERE clause. False rows do not contain the variable value specified in the clause. EVAL 1, EVAL 3, EVAL 4, and EVAL 5 evaluate true rows in the table using indices. EVAL 2 evaluates true rows of a table without using indices. EVAL strategies are explored in more detail in the section below on "WHERE Clause EVAL Strategies" on page 187.

SPD Indexes

SPD Server uses segmented indices. A segmented index is created by dividing the index of a table into equally sized ranges of rows. Each range of rows is called a segment, or slot. You use the SEGSIZE= setting to define the size of the segment. A series of sub-indices each point to blocks of rows in the table. By default, SPD Server creates an index segment for every 8192 rows in a table.

The SPD segmented index facilitates SPD Server's parallel evaluation of WHERE clauses with an indexed predicate. First, the SPD index supports a pre-evaluation phase to determine which segments contain values that satisfy the predicate. Pre-evaluation speeds queries by eliminating segments that do not contain any possible values. Then, a number of threads up to the value of the SPDSTCNT= variable are launched to query the remaining index segments. The threads query the segments of the SPD index in parallel to retrieve the segment rows that satisfy the predicate. When all segments have been queried, the per-segment results are accumulated to determine the rows that satisfy the predicate. If the query contains multiple indexed predicates, then those predicates are also evaluated in parallel. When all predicates have been completed, their results are accumulated to determine the rows that satisfy the query.

MINMAX Variable List

SPD Server contains a table option called MINMAXVARLIST=. The primary purpose of MIINMAXVARLIST= is for use with SPD Server dynamic cluster tables, where specific members in the dynamic cluster contain a set or range of values, such as sales data for a given month. When an SPD Server SQL subsetting WHERE clause specifies specific months from a range of sales data, the WHERE planner checks the MIN and MAX variable list. Based on the MIN and MAX list information, the SPD Server WHERE planner includes or eliminates member tables in the dynamic cluster for evaluation.

Use the MINMAXVARLIST= table option with either numeric or character-based columns. MINMAXVARLIST= uses the list of columns you submit to build a variable list. The MINMAXVARLIST= list contains only the minimum and maximum values for each column. The WHERE clause planner uses the index to filter SQL predicates quickly, and to include or eliminate member tables belonging to the cluster table from the evaluation.

Although the MINMAXVARLIST= table option is primarily intended for use with dynamic clusters, it also works on standard SPD Server tables. MINMAXVARLIST= can help reduce the need to create many indexes on a table, which can save valuable resources and space.

The MINMAXVARLIST= table option is available only when a table is being created or defined. If a table has a MINMAXVARLIST= variable list, moving or copying the table will destroy the variable list unless MINMAXVARLIST= is specified in the table output.

```
%let domain=path3 ;
%let host=kaboom ;
%let port=5201 ;

LIBNAME &domain sasspds "&domain"
   server=&host..&port
   user='anonymous' ;

/* Create three tables called */
/* xy1, xy2, and xy3.         */

data &domain..xy1(minmaxvarlist=(x y));
  do x = 1 to 10;
  do y = 1 to 3;
  output;
  end;
end;
run;

data &domain..xy2(minmaxvarlist=(x y));
  do x = 11 to 20;
  do y = 4 to 6 ;
  output;
  end;
end;
run;

data &domain..xy3(minmaxvarlist=(x y));
  do x = 21 to 30;
  do y = 7 to 9 ;
  output;
  end;
end;
run;


/* Create a dynamic cluster table */
/* called cluster_table out of    */
/* new tables xy1, xy2, and xy3   */

PROC SPDO library=&domain ;
   cluster create cluster_table
      mem=xy1
      mem=xy2
      mem=xy3
      maxslot=10;
quit;


/* Enable WHERE evaluation to see  */
/* how the SQL planner selects     */
/* members from the cluster. Each  */
/* member is evaluated using the   */
/* min-max variable list.          */

%let SPDSWDEB=YES;


/* The first member has true rows  */

PROC PRINT data=&domain..cluster_table ;
   where x eq 3 and y eq 3;
run;


/* Examine the other tables */

PROC PRINT data=&domain..cluster_table ;
   where x eq 19
   and y eq 4 ;
run;

PROC PRINT data=&domain..cluster_table ;
   where x eq 22
   and y eq 9;
run;

PROC PRINT data=&domain..cluster_table ;
   where x between 1 and 10
   and y eq 3;
run;

PROC PRINT data=&domain..cluster_table ;
   where x between 11 and 30
   and y eq 8 ;
```

```
run;


/* Delete the dynamic cluster table. */


PROC SPDO library=&domain ;
   cluster undo cluster_table ;
quit;

PROC DATASETS lib=&domain nolist;
   delete xy1 xy2 xy3 ;
quit ;
```

WHERE Clause Planner

The WHERE clause Planner implemented in SPD Server avoids computation-intensive operations and uses simple computations where possible. WHERE clauses in large database operations can be very resource-intensive operations. In SPD Server 3.x and earlier releases, query authors often needed to manually tune queries for performance. The tuning was accomplished using macro variables and index settings. The WHERE clause planner integrated into SPD Server does the tuning work for the user by automatically costing the different approaches to index evaluation.

WHERE-Costing Using Duplicity and Distribution Values

Two key factors are used to evaluate, or cost WHERE clause indices. The factors are duplicity and distribution.

Duplicity refers to the proportion expressed by the number of rows in a table divided by the number of distinct values in the index. When many observations in a table hold the same value for a given variable, the variable value is said to have a high duplicity. An example of a table with high duplicity might be a table of unleaded gasoline prices from service stations in the same area of a large city.

Conversely, when a table has only one or few observations that contain a given value for a variable, then that value can be described as low duplicity. An example of a table with low duplicity might be an office phone directory, where the variable for phone extension is always unique.

The duplicity value for an index ranges from 1 to the number of rows in the table. Indices with a duplicity value of 1 are unique. Indices with high duplicity generate a score that is close to the number of rows in the table.

Distribution refers to the sequential proximity between observations for values of a variable that are repeated throughout the variable's data set distribution. When a certain value for a variable exists in many observations that are scattered uniformly throughout the table, that value is said to have a wide distribution. If a variable value exists in many contiguous or nearly contiguous rows, the distribution is clustered.

WHERE Clause EVAL Strategies

SPD Server indexing keeps track of the duplicity and distribution of variable values in a table and uses them to calculate the cost of a WHERE clause. The WHERE clause planner uses four evaluation strategies to determine the number of rows that will be required to execute a given query. The four evaluation strategies are EVAL 1, EVAL 2, EVAL 3, and EVAL 4. True rows are rows that contain the variable values specified in a WHERE clause. False rows do not contain the variable value specified in the clause.

EVAL 1, EVAL 3, EVAL 4, and EVAL 5 evaluate true rows in the table using indices. EVAL 2 evaluates true rows of a table without using indices.

- **EVAL 1** evaluates true rows using an index to locate the true rows in each segment of the table. The index evaluation process generates a list of row IDs per segment. EVAL 1 accepts WHERE clause operators for equivalency expressions such as **EQ**, =, **LE**, <=, **LT**, <, **GE**, >=, **GT**, >, **IN**, and **BETWEEN**. EVAL 1 uses threaded parallel processing across the index segments to permit concurrent evaluation of multiple indices. EVAL 1 combines multiple segment bitmaps from queries that use multiple indices to generate the list of row IDs per segment.
- **EVAL 2** takes true rows as determined by EVAL 1, EVAL 3, or EVAL 4, and then uses brute force to eliminate any rows shown to be false, leaving a table that contains only true rows. EVAL 2 processes all rows of a table when no index evaluation is possible. For example, no index evaluation is possible when an index is not present or when some predecessor function performs an operation that invalidates the index.
- **EVAL 3** is a single index sequential process. Use EVAL 3 when the number of rows returned by an index is unique or nearly unique (when duplicity is low). EVAL 3 returns a list of true rows for the entire table. EVAL 3 only supports the equality operators **EQ** and =.
- **EVAL 4** is similar to EVAL 3 but supports a larger set of inequality and inclusion operators, such as **IN**, **GT**, **GE**, **LT**, **LE**, and **BETWEEN**.
- **EVAL 5** can operate when the SPD Server Index Scan facility is used. The EVAL 5 strategy uses index metadata and aggregate SQL functions to evaluate true rows. The EVAL 5 strategy does not require a table scan.

  For example, when x is indexed, and SPD Server uses EVAL 5 to evaluate the SQL expression

  ```
  count(*) where x=5 ,
  ```

  the index metadata is scanned for the condition, x = 5 instead of performing table scans. The EVAL 5 strategy supports the min(), max(), count(), count(distint), nmiss(), and range() functions. The EVAL 5 strategy cannot be used on SQL expressions, which use functions other than those listed above.

The WHERE clause planner in SPD Server 3.x relied heavily on EVAL 1 and EVAL 2 threaded strategies to evaluate most clauses. Sometimes the SPD Server 3.x EVAL 1 and EVAL 2 strategies would over-thread and over-manipulate indices during the evaluations during WHERE clause evaluation. This resulted in reduced performance or excessive resource consumption. With SPD Server 4.5's WHERE clause costing in place, EVAL 3 and EVAL 4 strategies are more suitable evaluation engines which conserve resources and boost processor performance.

Assigning EVAL Strategies

Overview of Assigning EVAL Strategies

The SPD Server WHERE clause planner uses the following logic when selecting an EVAL strategy to evaluate expressions:

When the planner encounters a WHERE clause, it builds a tree that represents all of the possible predicate expressions. The objective of the WHERE clause planner is to divide the set of predicate expressions into two trees. One tree collects predicate expressions that lack usable indices and are constrained to EVAL 2 evaluation. The remaining predicate expressions are put in the other tree. Each of the predicate expressions in the second tree is scanned and assigned an evaluation strategy of EVAL 1, EVAL 3, or EVAL 4, depending on the WHERE clause costing values and the syntax used in the predicate expression .

The second tree, which does not use the EVAL 2 method, is scanned for predicate expressions that return values with high duplicity . When high duplicity predicate expressions are identified, they are ranked. The predicate expression with the highest duplicity value is set aside for an index-based evaluation. All of the other remaining predicate expressions are evaluated using the EVAL 2 tree strategy. The lowest duplicity predicate expression is evaluated using either the EVAL 3 or the EVAL 4 strategy. The syntax used in the predicate expression determines which of the two strategies to use. Frequently, the single index EVAL 3 or EVAL 4 is chosen because single index evaluations require smaller processing loads and yield reliable results. With a low processor overhead and a high data yield, there is no reason to include other indices when a single index is sufficient.

When the WHERE clause planner determines that no predicate expressions meet the high duplicity criteria, it chooses the EVAL 1 strategy. Before the EVAL 1 operation is performed, the costing algorithm is run on the remaining predicates to prune any predicate expressions that represent large processor loads and large data yields. Predicate expressions that will require large processor loads and produce large data yields are moved to the EVAL 2 tree.

Index Scan Facility

When SPD Server invokes the Index Scan facility, and the SQL aggregate uses the specified supported functions for EVAL 5, the EVAL 5 strategy uses a fast index metadata scan to select SQL statements that meet the aggregate function criterion.

High Yield Predicate Expressions

A large, or high data yield expression has a high percentage of rows containing true segments. The default threshold for a high yield expression is one where less than 25% of the rows evaluated are returned by the predicate. At this point, processor costs related to index use begin increasing without proportional returns on the evaluation results.

High Processing Load Predicate Expressions

Predicate expressions that require high processing loads are predicates that usually require large amounts of index manipulation before they can complete. When the amount of index work that is required exceeds the work that is required to use an EVAL 2 strategy, the predicate expression will be best evaluated by the EVAL 2 tree. Open-ended predicate expressions that contain many syntax inequality operators such as GT and LT or many variations in syntax are good high work candidates for EVAL 2. High work predicate expressions are detected by comparing the number of unique values in the predicate expression to the number of unique values contained in the index.

High Yield and High Processing Load Predicate Expressions

When all predicate expressions in EVAL 1 are high yield or high processor load, SPD Server uses segmented costing. In segmented costing, true segments are passed to EVAL 2 for processing. EVAL 2 only processes table segments that can provide true rows for the WHERE clause.

Turning WHERE Clause Costing Off

You can use the SPD Server **spdsserv.parm** parameter file to configure the default WHERECOSTING parameter setting to ON. If you want to turn off WHERE clause costing within the scope of a job, you can use macros or a DATA step to turn WHERE clause costing off and on:

- The SPDSWCST=NO macro setting turns off WHERE clause costing.
- The SPDSWSEQ=YES macro overrides WHERE clause costing and allows you to force a global EVAL3 or EVAL4 strategy.
- The WHERECOSTING parameter can be removed or set to NOWHERECOSTING in the **spdsserv.parm** file if you want to turn off costing for the entire server.

If you turn off WHERE clause costing in the **spdsserv.parm** parameter file, or if you use the macro setting SPDSWCST=NO, the WHERE clause planner reverts to the rules-based WHERE clause planning of earlier versions of SPD Server.

WHINIT: Indexed and Non-Indexed Predicates

Overview of WHINIT

If SPD Server is not configured to use dynamic WHERE-costing, the WHERE clause planner reverts to the rule-based heuristics of WHINIT. WHINIT uses rules to select indexes for the predicates, and then select the most appropriate EVAL strategy for the query.

WHINIT splits the WHERE clause, represented as a tree, into non-indexed and indexed parts. Non-indexed predicates include

- non-indexed columns
- functions
- columns that have indexes that WHINIT cannot use

If the WHERE clause planner places indexed predicates in the non-indexed tree, it is usually because the predicates involve an OR expression. An example of a predicate with an OR expression is, where x = 1 or y = 2. Even if column x is indexed, WHINIT cannot use the index because the OR is disjunctive. As a result of the disjunctive OR, the planner cannot use the index, and places both the predicates x = 1 and y = 2 into the non-indexed part of the WHERE tree.

Sample WHINIT Output

SAS users can use an SPD Server macro variable to view WHERE clause planner output:

```
%let SPDSWDEB=YES;
```

The following is what the WHINIT plan might give for the following scenario:

- a WHERE clause of **`where a = 1 and b in (1 2 3) and d = 3 and (d + 3 = c)`**
- an SPD index IDX_ABC on columns (A B C)
- an SPD index D on column (D)

Note: The line numbers are for reference; they are NOT part of the actual output.

```
1:whinit: WHERE ((A=1) and B in (1, 2, 3) and (D=3) and (C=(D+3)))
2:whinit: wh-tree presented
3:
       /-NAME = [A]
 4:           /-CEQ----|
 5:          |
     \-LITN = [1]
 6: --LAND---|
 7:          |
     /-NAME = [B]
 8:          |--IN-----|
 9:          |
    |          /-LITN = [1]
10:          |
    \-SET----|
11:          |
             |--LITN = [2]
12:          |
              \-LITN = [3]
13:          |
    /-NAME = [D]
14:          |--CEQ----|
15:          |
    \-LITN = [3]
16:          |
    /-NAME = [C]
17:           \-CEQ----|
18:
     |          /-NAME = [D]
19:
      \-AADD---|
20:
                \-LITN = [3]
21:whinit: wh-tree after split
22:           /-NAME = [C]
23: --CEQ----|
24:          |
     /-NAME = [D]
25:           \-AADD---|
26:
      \-LITN = [3]
27:whinit: SBM-INDEX D uses 50% of segs (WITHIN maxsegratio 75%)
28:whinit: INDEX tree after split
29:
      /-NAME = [A] <1>SBM-INDEX IDX_ABC (A,B)
30:           /-CEQ----|
31:          |
     \-LITN = [1]
32: --LAND---|
33:          |
     /-NAME = [B]
34:          |--IN-----|
35:          |
    |          /-LITN = [1]
36:          |
     \-SET----|
37:          |
              |--LITN = [2]
38:          |
               \-LITN = [3]
39:           |
     /-NAME = [D] <2>SBM-INDEX D (D)
40:           \-CEQ----|
41:
       \-LITN = [3]
42:whinit returns: ALL EVAL1(w/SEGLIST) EVAL2
```

Line 1 shows what the WHINIT Planner received. Do not be surprised -- what the Planner receives can differ from your entries. Sometimes SAS optimizes or transforms a WHERE clause before passing it to SPD Server. For example, it can eliminate entities such as NOT operators, the union of set lists, and so on.

Lines 2 to 20 show the presented WHERE clause in a tree format. The tree format is a user-readable form of the actual WHERE clause that is processed by the SPD Server engine.

Lines 21 to 26 show the non-indexed WHERE tree, the result of splitting off the indexed part. The non-indexed WHERE tree can be empty or it can look the same as lines 2 to 20 if no indexes are selected. Consider that it is the non-indexed part of the WHERE clause that WHINIT uses to filter records obtained by the indexed strategies (EVAL1, 3 or 4).

Lines 27 to 41 shows that the percentage of segments containing values selected from column D is with the maximum allowed to proceed with pre-segment logic. Therefore, only those segments that contain values that satisfy the WHERE clause for column D will be included in further query processing for that column. Composite index IDX_ABC and simple index D are used to resolve the indexed WHERE clause predicates.

Line 42, the last line in our output, shows which strategies are used. The first keyword ALL indicates that SPD Server can identify correctly ALL resulting records, without help from the SAS System. First, SPD Server will call EVAL1, an indexed method, to quickly access a list of records that satisfy **`where a = 1 and b in (1 2 3) and d = 3`**, then it will use EVAL2 to determine whether **`c = d + 3`** is true on these records.

When output from EVAL1 displays the suffix w/ seglist, as it does in the above output, it means that SPD indexes were detected, and that the indexes were used to filter only the segments that satisfy the indexed predicates. When EVAL1 has no suffix, it means that ALL segments will be evaluated.

SPD Server stores the minimum and maximum values for a table index in a global structure. WHINIT can use the numeric range to 'prune' predicates when the table index values are out of the min / max range. WHINIT output keywords can indicate pruning activity. For example, if WHINIT had determined that the values for D (in our WHERE clause) are between 5 and 13, then as a consequence, the predicate **`where d = 3`** could never be true. In this case, WHINIT would have pruned this predicate since it is logically impossible, or FALSE. Pruning can also affect higher nodes. If the **d = 3** predicate were deemed FALSE, then the AND sub tree would also be FALSE and would also have been pruned.

WHINIT Output Return Keywords

In the last line of the output, ALL is one of the following keywords that the Planner can display:

- **ALL** - SPD Server can evaluate ALL of the WHERE clause when determining which records satisfy the clause.
- **SOME** - SPD Server can handle SOME or part of the WHERE clause; it will then need SAS to help identify resulting records.
- **NONE** - SPD Server cannot evaluate this WHERE clause; SAS will perform all evaluations.

- **TRUE** - SPD Server has determined that the entire WHERE clause is TRUE, and that all the records satisfy the given WHERE clause.
- **FALSE** - SPD Server determined that the WHERE clause is FALSE, that is, no records can satisfy the WHERE clause.
- **RC=number** - An internal error has occurred; the error number is displayed.
- **EVALx** - the EVAL strategies the planner will use; *x* can be 1, 2, 3, or 4.

Composite Index Permutations

A composite index can involve one or more in set equality predicates, such as an index on columns (a b c). When WHINIT is presented with a WHERE clause that has such a composite index, such as **`where a = 1 and b in (1 2 3) and c in (4 5)`**, it will generate all permutations of this compound key, probing the index for each value. In our example, six values are generated:

(a b c) = (1 1 4) (1 1 5) (1 2 4) (1 2 5) (1 3 4) (1 3 5)

The permutations start at the back end of the key to take advantage of locality: to locate keys with close values that access the same disk page. This means less input/output operations on the index.

How to Affect the WHERE Planner

Macro Variable SPDSWCST=

To turn off dynamic WHERE-costing, specify

```
%let SPDSWCST=NO;
```

Macro Variable SPDSWDEB=

To turn on WHINIT planning output, specify

```
%let SPDSWDEB=YES;
```

Macro Variable SPDSIRAT=

To affect the WHERE planner SPD index pre-evaluation, specify

```
%let SPDSIRAT=index-segment-ratio;
```

The SPDSIRAT= macro variable specifies a maximum percentage (ratio) for the number of segments in the hybrid bitmap that must contain the index value before the WHERE planner should pre-evaluate a segment list.

The segment list enables the planner to launch threads only for segments that contain the value. If the value number exceeds the ratio, the planner performs no pre-evaluation. Instead, the planner launches a thread for each segment in the table.

The SPDSIRAT= macro variable option can be used to ensure that time spent in pre-evaluation does not exceed the cost of launching a thread for each segment in the table. By default SPDSIRAT= is set to 75 percent. This means that if an index value is contained in 75 percent or less of the index segments, the hybrid bitmap logic will pre-evaluate the value and return a list of segments to the WHERE clause planner. If more than 75 percent of the index segments contain the target index value, the time spent on pre-evaluation might be more than the time saved by skipping a small number of segments.

For some 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. The allowable range to adjust the setting value is from 0 to 100, where 0 means **never** perform WHERE clause pre-evaluation, and 100 means **always** perform WHERE clause pre-evaluation.

Macro Variable SPDSNIDX= or Table Option NOINDEX=

To suppress WHINIT use of any index, specify the no index SPD Server macro variable or the corresponding SPD Server table option:

```
%let SPDSNIDX=YES;
```

```
data _null_;
set foo.a (noindex=yes);
```

Macro Variable SPDSWSEQ=

By default, when WHINIT detects equality predicates that have indexes, it chooses EVAL1. However, the user can decide that sequential EVAL3 or EVAL4 methods are better. For example, in an equality WHERE predicate such as where x = 3, WHINIT will default to EVAL1 to evaluate the clause. If a user knows that the table queried has only a few records that can satisfy this predicate, EVAL3 might be a better choice. To force WHINIT to choose EVAL3/4, specify:

```
%let SPDSWSEQ=YES;
```

Server Parameter Option [NO]WHERECOSTING

Controls whether the server uses dynamic WHERE-costing. When dynamic WHERE-costing is disable, the rules-based WHINIT heuristic is used to tune WHERE clauses for performance. The default setting is for NOWHERECOSTING.

WHERENOINDEX Option

A user might decide that one or more indexes selected by a WHINIT plan are not the best choice. This can occur because WHINIT is rule-based, not cost-based. Sometimes WHINIT selects a less-than-optimal plan. WHINIT's use of specific indexes can be affected by specifying the SPD Server option WHERENOINDEX= in your DATA step.

```
data _null_;
set foo.a (wherenoindex=(idx_abc d))
```

This example specifies that WHINIT not use index idx_abc and index d.

When and Why Should I Suppress Indexes?

Most rule-based planners, including WHINIT from SPD Server, assume that the index has a uniform distribution of values between the upper and lower value boundaries. This means if data values range between 2 and 10, that there are an equal number of 3s and 4s, and so on. When the assumption of a uniform distribution is false, an indexed predicate can return a large number of records. In turn, this causes WHINIT's indexed plan to run slower than a sequential read of the entire table. In this case the index should be suppressed.

Here is another, more subtle instance. When the WHERE clause uses only the front part of the key, WHINIT selects a composite index. Assume an index **abcd** on columns A, B, C, and D, and an index **e** on column E, and specify the WHERE clause

```
where a = 3 and e = 5;
```

Normally, WHINIT will select both indexes (**abcd** and **e**) and choose EVAL1. However, using the index **abcd** just to interrogate **a** might return a large number of records. In this case, suppressing the **abcd** index might be a good idea. If so, WHINIT will still choose EVAL1 for **e = 5**, or EVAL3 if SPDSWEV1=NO, and EVAL2, the post-filter, for **a = 3**.

Identical Parallel WHERE Clause Subsetting Results

Overview of Parallel WHERE Clause Subsetting

Under certain circumstances, it is possible to perform parallel WHERE clause subsetting on a table more than once and to receive slightly different results. This event can occur when submitting parallel WHERE clause code to SPD Server that uses the SAS **OBS=nnnn** data set option.

The SAS **OBS=nnnn** data set option causes processing to end with the specified (nth) observation in a table. Because parallel WHERE clause processing is threaded, subsetting a table and using **OBS=nnnn** might not produce identical results from run to run, or different batch jobs using the same WHERE clause code might produce slightly different results.

When a parallel WHERE-cause evaluation is split into multiple threads, SPD Server uses a multi-threading model that is designed to return rows as fast as possible. Some threads might be able to complete row scans incrementally faster than other threads, due to uneven loads across multiple processors or system contention issues. This inequity can create minute variances that can generate nonidentical results to the same subsetting request.

If you have code that performs parallel WHERE clause subsetting in conjunction with the **OBS=nnnn** data processing option, and if it is critical that successive WHERE clause subsets on the same data must be identical, you can eliminate thread contention error by setting the thread count value for that operation to 1.

To set the SPD Server thread count value, you can use the SPDSTCNT= macro:

```
%let SPDSTCNT=1;
```

The same potential for subsetting variation applies when a DATA step uses the **OBS=nnnn** data processing option with a parallel by-clause, such as:

```
    data test1;
      set spds45.testdata (obs=1000);
      where j in (1,5,25);
      by i;
    run;
```

Use the SPDSTCNT= macro solution to ensure identical results across multiple identical table subsetting requests.

WHERE Clause Subsetting Variation Example

Job 1 and Job 2 use the same tables and data requests but produce non-identical results as seen in the respective Job 1 and Job 2 outputs.

To eliminate variation in the output, simply add the thread count statement

```
%let SPDSTCNT=1;
```

to the beginning of each job.

Job 1

```
data test1;
   set spds45.testdata
     (obs=1000);
   where j in (1,5,25);
run;

PROC SORT data=test1;
   by i;
run;

PROC PRINT data=test1
   (obs=10);
run;
```

Job 1 Output

```
The SAS System     11:44 Monday, May 9, 2005   1

     Obs    a      i       j    k

       1         24601     1    1
       2         24605     5    5
       3         24625    25    0
       4         24701     1    1
       5         24705     5    5
       6         24725    25    0
       7         24801     1    1
       8         24805     5    5
       9         24825    25    0
      10         24901     1    1
```

Job 2

```
data test2;
   set spds45.testdata
     (obs=1000);
   where j in (1,5,25);
run;

PROC SORT data=test2;
```

```
   by i;
run;

PROC PRINT data=test2
   (obs=10);
run;
```

Job 2 Output

```
The SAS System
11:44 Monday, May 9, 2005   1

     Obs   a      i      j    k

       1          1      1    1
       2          5      5    5
       3         25     25    0
       4        101      1    1
       5        105      5    5
       6        125     25    0
       7        201      1    1
       8        205      5    5
       9        225     25    0
      10        301      1    1
```

WHERE Clause Examples

Data for WHERE Examples

The WHERE clause examples below assume that the user is connected to the SPD Server LIBNAME foo and has executed the following SAS code:

```
data foo.a;
do i=1 to 100;
  do j=1 to 100;
    do k=1 to 100;
      m=mod(i,3);
      output;
    end;
  end;
end;
run;

proc datasets lib=foo;
modify a;
index create ijk = (i j k);
index create j;
index create m;
quit;
```

Example 1 "where i = 1 and j = 2 and m = 4"

```
whinit: WHERE ((I=1) and (J=2) and (M=4))
whinit: wh-tree presented

      /-NAME = [I]
           /-CEQ----|
          |
    \-LITN = [1]
 --LAND---|
          |
    /-NAME = [J]
          |--CEQ----|
          |
    \-LITN = [2]
          |
    /-NAME = [M]
           \-CEQ----|

      \-LITN = [4]
whinit: wh-tree after split
 --[empty]
whinit: pruning INDEX node which is trivially FALSE
           /-NAME = [M] INDEX M (M)
 --CEQ----|
           \-LITN = [4]
whinit: INDEX tree evaluated to FALSE
whinit returns: FALSE
```

Here the only values that column M can contain are 0, 1, or 2. Thus, the predicate **m = 4** is identified as trivially FALSE. Because this predicate is part of an AND predicate, it too is FALSE. Consequently, the entire WHERE clause is pre-evaluated to FALSE, meaning that no records can satisfy this WHERE clause. Thus, as a result of the pre-evaluation, no records are actually read from disk. This is an example of optimization at its best.

WHERE_EXAMPLE 2: where i in (1, 2, 3) and j in (4, 5, 6, 7) and k > 8 and m = 2

```
  whinit: WHERE (I in (1, 2, 3) and J in (4, 5, 6, 7) and (K>8) and (M=2))
whinit: wh-tree presented

      /-NAME = [I]
           /-IN-----|
          |
    |          /-LITN = [1]
          |
    \-SET----|
          |
             |--LITN = [2]
          |
              \-LITN = [3]
 --LAND---|
          |
    /-NAME = [J]
```

```
          |--IN-----|
          |
   |          /-LITN = [4]
          |
    \-SET----|
          |
             |--LITN = [5]
          |
             |--LITN = [6]
          |
              \-LITN = [7]
          |
    /-NAME = [K]
          |--CGT----|
          |
    \-LITN = [8]
          |
    /-NAME = [M]
           \-CEQ----|

      \-LITN = [2]
whinit: SBM-INDEX M uses 60% of segs(WITHIN maxsegratio 100%)
whinit: wh-tree after split
           /-NAME = [K]
 --CGT----|
           \-LITN = [8]
whinit: INDEX tree after split

      /-NAME = [I] <1>SBM-INDEX IJK (I,J)
           /-IN-----|
          |
   |          /-LITN = [1]
          |
    \-SET----|
          |
             |--LITN = [2]
          |
              \-LITN = [3]
 --LAND---|
          |
    /-NAME = [J]
          |--IN-----|
          |
   |          /-LITN = [4]
          |
    \-SET----|
          |
             |--LITN = [5]
          |
             |--LITN = [6]
          |
              \-LITN = [7]
          |
    /-NAME = [M] <2>SBM-INDEX M (M)
           \-CEQ----|
      \-LITN = [2]
whinit returns: ALL EVAL1(w/SEGLIST) EVAL2
```

Here, a composite index **ijk** was defined on columns (**i j k**). This composite index is used for column's **i** and **j**, which is an equality index predicate. Column **k** is **not** included because it involves an inequality operator (greater than). Since there are no other indexes for column **k**, this predicate is assigned to EVAL2 . EVAL2 will post-filter the records obtained through the use of indexes.

WHERE_EXAMPLE 3: where i = 1 and j > 5 and mod(k, 3) = 2

```
  whinit: WHERE ((I=1) and (J>5) and (MOD(K, 3)=2))
whinit: wh-tree presented

      /-NAME = [I]
           /-CEQ----|
          |
    \-LITN = [1]
 --LAND---|
          |
    /-NAME = [J]
          |--CGT----|
          |
    \-LITN = [5]
          |
             /-FUNC = [MOD()]
          |
    /-FLST---|
          |
   |         |--NAME = [K]
          |
   |          \-LITN = [3]
           \-CEQ----|

      \-LITN = [2]
whinit: wh-tree after split

      /-FUNC = [MOD()]
           /-FLST---|
          |
   |--NAME = [K]
          |
    \-LITN = [3]
 --CEQ----|
           \-LITN = [2]
whinit: SBM-INDEX IJK uses 1% of sges(WITHIN maxsegratio 75%)
whinit: SBM-INDEX J uses at least 76% of segs(EXCEEDS maxsegratio 75%)
whinit: INDEX tree after split

      /-NAME = [I] <1>SBM-INDEX IJK (I)
           /-CEQ----|
          |
    \-LITN = [1]
 --LAND---|
          |
    /-NAME = [J] <2>SBM-INDEX J (J)
```

```
          \-CGT----|

      \-LITN = [5]
whinit returns: ALL EVAL1(w/SEGLIST) EVAL2
```

Here the indexes on column **i**, a composite index on the columns (**i j k**), and the column **j** are combined. In this example WHINIT uses both EVAL1 and EVAL2. The **j** predicate involves an inequality operator (greater than). Therefore, WHINIT cannot combine the predicate with **i** and the composite index involving **i** and **j** (and **k**).

Using the composite index **ijk** in this plan might be inefficient. If a smaller composite index (that is, one on **i j** or a simple index on **i**) were available, WHINIT would select it. In lieu of this, try benchmarking the plan. Suppress the composite index and compare the results to the existing plan to see which is more efficient (faster) on your machine.

The example that follows shows what WHINIT's plan would look like with the composite index suppressed.

WHERE_Example 4: where i = 1 and j > 5 and mod(k, 3) = 2

In this example, the index IJK is suppressed.

```
  whinit: WHERE ((I=1) and (J>5) and (MOD(K, 3)=2))
whinit: wh-tree presented

       /-NAME = [I]
           /-CEQ----|
          |
     \-LITN = [1]
 --LAND---|
          |
     /-NAME = [J]
          |--CGT----|
          |
     \-LITN = [5]
          |
              /-FUNC = [MOD()]
          |
     /-FLST---|
          |
    |         |--NAME = [K]
          |
    |          \-LITN = [3]
           \-CEQ----|

      \-LITN = [2]
whinit: wh-tree after split

      /-NAME = [I]
           /-CEQ----|
          |
    \-LITN = [1]
 --LAND---|
          |
             /-FUNC = [MOD()]
          |
    /-FLST---|
          |
   |        |--NAME = [K]
          |
   |         \-LITN = [3]
          \-CEQ----|

      \-LITN = [2]
whinit: SBM_INDEX J uses at least 76% of segs (EXCEEDS maxsegratio 75%)
whinit: checking all hybrid segments
whinit: INDEX tree after split
          /-NAME = [J] <1>SBM-INDEX J (J)
 --CGT----|
          \-LITN = [5]
whinit returns: ALL EVAL1 EVAL2
```

Notice that the predicate involving column **i** is non-indexed. WHINIT evaluates it using EVAL2. Because the predicate **j > 5** still uses an inequality comparison, WHINIT continues to use EVAL1. Finally, because the percentage of segments that contain values for column **J** exceeds the maximum segment ratio, pre-segment logic is not done on column **J**. As a result, all segments of the table are queried for values that satisfy the WHERE clause for column **J**.

Server-Side Sorting

Overview of Server-Side Sorting

In most instances, using a BY clause in SAS code submitted to an SPD Server table triggers a BY clause evaluation by SPD Server. This BY clause assertion to the SPD Server might or might not require sorting to produce the ordered row set that the BY clause requires. In some cases, a table index can be used to sort the rows to satisfy a BY clause.

For example, the input table to a PROC SORT step is sorted in server context (by the associated LIBNAME proxy). The rows are returned to PROC SORT in BY clause order. In this case, PROC SORT knows that the data is already ordered, and writes the data to the output table without sorting it again. Unfortunately, this approach still must send the data from the LIBNAME proxy to the SAS client and then back to the LIBNAME proxy. However, there are other ways to use an SPD Server SQL Pass-Through COPY statement to avoid the overhead of the data round-trip.

SPD Server attempts to use an index when performing a BY clause. The software looks specifically for an index that has variables in the order specified in the BY clause. On the surface this seems like a good idea: table row order is already determined because the keys in the index are ordered. SPD Server reads the keys in order from the index, and then returns the rows from the table, based on the row IDs that are stored with the index key values.

Use caution when using BY clauses on tables that have indexes on their BY columns. Using the index is not always a good idea. When no suitable index exists to determine BY clause order, SPD Server uses a parallel table scan sort that keeps the table row intact with the sort key. The time required to access a highly random distribution of row IDs (obtained by using the index) can greatly exceed the time required to sort the rows from scratch.

When you use a WHERE clause to filter the rows from an SPD Server table with a BY clause to order them in a desired way, SPD Server handles both the subsetting and the ordering for this request. In this case, the filtered rows that were qualified by the WHERE clause are fed directly into a sort step. Feeding the filtered rows into the sort step is part of the parallel WHERE clause evaluation. The final ordered row set is the result. In this case, the previous discussion of index use does not apply. Index use for WHERE clause filtering is very desirable and greatly improves the filtering performance that feeds into the sort step. Arbitrarily suppressing index use with a WHERE and BY combination should be avoided.

Suppressing the Use of Indexes

Suppress the use of indexes on the BY clause by using the SPDSNIDX=YES macro variable or by asserting the NOINDEX=YES table option. Suppressing the use of the index can significantly improve time required to process a BY clause in SPD Server.

Advantages of Implicit Server Sorts

An exceptional feature is the software's ability to execute ad hoc order-BY queries without pre-sorting the table on the BY variables. Many SAS job streams are structured with code that alternates PROC SORT followed by PROC xxxx invocations, where the PROC SORT step is needed only for the execution of the PROC xxxx step.

When sort order is relevant only to the following step, eliminate the PROC SORT step and just use the BY clause on the PROC xxxx step. This eliminates the extra data transfer (to PROC SORT from SPD Server and then back from PROC SORT to SPD Server) to store the sorted result. Even if SPD Server performs the sort associated with the PROC SORT, there is extra data transfer. The data's round trip from the server to the SAS client and back can impose a substantial time penalty.

Chapter 12
SPD Server Macro Variables

Introduction

Macro variables, known as symbolic variables, operate similarly to LIBNAME and table options. But, they have an advantage because they apply globally. That is, their value remains constant until explicitly changed.

This chapter presents reference information for SPD Server macro variables, including their purpose, default values, and when and how to use them. The variables are grouped by function or purpose of the default value. Changing the value can also change the purpose, making the variable fall into another group.

For example, the default setting for the macro variable SPDSSADD= is NO. The SPDSSADD= macro enhances performance during data appends. Setting SPDSSADD= to YES changes the way the variable functions. The macro setting SPDSADD=YES ensures compatibility with the Base SAS engine. The default setting improves performance. Changing the setting from the default improves Base SAS software compatibility.

To set a macro variable to YES submit the following statement:

```
%let MACROVAR=YES;
```

Note: Assignments for macro variables with YES|NO arguments must be entered in uppercase (capitalized).

When you specify table option settings, precedence matters. If you specify a table option after you set the option in a macro variable statement, the table option setting takes precedence over the macro variable option setting. If you specify an option using a LIBNAME statement, and then later specify an option setting through a macro variable statement, the table option setting made in the macro variable takes precedence over the LIBNAME statement setting.

To view the default values for the SPD Server macro variables, use the SPDSMAC command associated with PROC SPDO. SAS displays the macro variables and their current settings. Understanding proper use of macro variables in SPD Server allows you to unleash the power of the software.

Variable for Compatibility with the Base SAS Engine

SPDSBNEQ=

Use the SPDSBNEQ= setting to specify the output order of table rows that have identical values in the BY column.

Syntax

SPDSBNEQ=YES|NO

Default: NO

Corresponding Table Option: BYNOEQUALS=

Arguments

YES
: outputs rows with identical values in a BY clause in random order.

NO
: outputs rows with identical values in a BY clause using the relative table position of the rows from the input table.

Description

SPDSBNEQ=NO configures the SPD Server to imitate the Base SAS engine behavior. If strict compatibility is not required, assign SPDSBNEQ=YES. Random output allows the SPD Server to create indexes and append to tables faster.

Example

Configure the SPD Server so that it output table rows as quickly as possible when processing rows that have identical values in the BY column.

```
%let SPDSBNEQ=YES;
```

Variables for Miscellaneous Functions

SPDSEOBS=

Use the SPDSEOBS= macro variable to specify the number of the last row (end observation) of a user-defined range that you want to process in a table.

Syntax

SPDSEOBS=*n*

Default: The default setting of 0 processes the entire table.

Corresponding Table Option: ENDOBS=

Arguments

n
: is the number of the end row.

Description

The SPD Server processes the entire table by default unless you specify a range of rows. You can specify a range using the macro variables SPDSSOBS= and SPDSEOBS=, or you can use the table options, STARTOBS= and ENDOBS=.

If you use the range start macro variable SPDSSOBS= without specifying an end range value using the SPDSEOBS= macro variable, SPD Server processes to the last row in the table. If you specify values for both SPDSSOBS= and SPDSEOBS= macro variables, the value of SPDSEOBS= must be greater than SPDSSOBS=. The SPDSSOBS= and SPDSEOBS= macro variables specify ranges for table input processing as well as WHERE clause processing.

Example

In order to create test tables, you configure the SPD Server to subset the first 100 rows of each table in your job. Submit the macro variable statement for SPDSEOBS= at the beginning of your job.

```
%let SPDSEOBS=100;
```

SPDSSOBS=

Use the SPDSSOBS= macro variable to specify the number of the starting row (observation) in a user-defined range of a table.

Syntax

SPDSSOBS=*n*

Default: The default setting of 0 processes the entire table.

Corresponding Table Option: STARTOBS=

Arguments

n
is the number of the start row.

Description

By default, SPD Server processes entire tables unless you specify a range of rows. You can specify a range using the macro variables SPDSSOBS= and SPDSEOBS=, or you can use the table options, STARTOBS= and ENDOBS=.

If you specify the end of a user-defined range using the SPDSEOBS= macro variable, but do not implicitly specify the beginning of the range using SPDSSOBS=, SPD Server sets SPDSSOBS= to 1, or the first row in the table. If you specify values for both SPDSSOBS= and SPDSEOBS= macro variables, the value of SPDSEOBS= must be greater than SPDSSOBS=. The SPDSSOBS= and SPDSEOBS= macro variables specify ranges for table input processing as well as WHERE clause processing.

Example

Print the INVENTORY.OLDAUTOS table, skipping rows 1-999, and beginning with row 1000. You should submit the SPDSSOBS= macro variable statement before the PROC PRINT statement in your job.

```
%let SPDSSOBS=1000;
```

The statement above specifies the starting row with SPDSSOBS=, but does not declare an ending row for the range using SPDSEOBS=. When the program executes, SAS will begin printing at row 1000 and continues until the final row of the table is reached.

```
PROC PRINT data=inventory.oldautos;
run;
```

SPDSUSAV=

Use the SPDSUSAV= macro variable to specify whether to save rows with nonunique (rejected) keys to a separate SAS table.

Syntax

SPDSUSAV=YES|NO|REP

Default: NO

Affected by Table Option : SYNCADD=

Use in Conjunction with Variable : SPDSUSDS=

Corresponding Table Option : UNIQUESAVE=

Arguments

YES
: writes rows with nonunique key values to a SAS table. Use the macro variable SPDSUSDS= to reference the name of the SAS table for the rejected keys.

NO
: nonunique key values are ignored and rejected rows are not written to a separate table.

REP
: when updating a master table from a transaction table, where the two tables share identical variable structures, the SPDSUSAV=REP option replaces the updated row in the master table instead of appending a row to the master table. The REP option only functions in the presence of a /UNIQUE index on the MASTER table. Otherwise, the REP setting is ignored..

Description

When performing an append operation, SPD Server does not save the rows that contain duplicate key values unless the SPDSUSAV= macro variable is set to YES.

When SPDSUSAV= is set to YES, SPD Server creates a hidden SAS table and writes rejected rows to the table. Use the SPDSUSDS= macro variable command to view the contents of the table. Each append operation creates a different table.

Example

Append several tables to the EMPLOYEE table, using employee number as a unique key. The appended tables should not have records with duplicate employee numbers.

At the beginning of the job, configure SPD Server to write any rejected (identical) employee number records to a SAS table. The macro variable SPDSUSDS= holds the name of the SAS table for the rejected keys.

```
%let SPDSUSAV=YES
```

Use a %PUT statement to display the name of the table, and then print the table.

```

%put Set the macro variable spdsusds to &spdsusds;

title 'Duplicate (nonunique) employee numbers found in
       EMPS';
PROC PRINT data=&spdsusds run;
```

SPDSUSDS=

Use the SPDSUSDS= macro variable to reference the name of the SAS table that SPD Server creates for duplicate or rejected keys when the SPDSUSAV= macro variable is set to YES.

Syntax

SPDSUSDS=

Default: SPD Server automatically generates identifying strings for the duplicate or rejected key tables.

Use in Conjunction with Table Option: SYNCADD=

Use in Conjunction with Variable: SPDSUSAV=

Corresponding Table Option: UNIQUESAVE=

Description

When SPDSUSAV= or UNIQUESAVE= is set to YES, SPD Server creates a table to store any rows with duplicate key values encountered during an append operation. Submitting the SPDSUSDS= macro variable references the generated name for the hidden SAS table.

To obtain the name and print the table's contents, reference the variable SPDSUSDS=.

Example

```
%let SPDSUSAV=YES
```

Use a %PUT statement to display the name of the table created by SPDSUSDS= and to print out the duplicate rows.

```
%put Set the macro variable spdsusds to &spdsusds;

title 'Duplicate Rows Found in MYTABLE
       During the Last Data Append';
PROC PRINT data=&spdsusds run;
```

SPDSVERB=

Use the SPDSVERB= macro variable to provide verbose details on all indexes, ACL information, and other information that is associated with SPD Server tables.

Syntax

SPDSVERB=YES|NO

Default:NO

Corresponding Table Option:VERBOSE=

Arguments

YES
: requests detail information for indexes, ACLs, and other SPD Server table values.

NO
: suppresses detail information for indexes, ACLs, and other SPD Server table values.

Example

You need information about associated indexes for the SPD Server table SUPPLY. Configure SPD Server for verbose details at the start of your session so you can see index details. Submit the SPDSVERB= macro variable as a line in your autoexec.sas file:

```
%let SPDSVERB=YES;
```

Submit a PROC CONTENTS request for the SUPPLY table:

```
PROC CONTENTS data=supply;
run;
```

SPDSFSAV=

Use the SPDSFSAV= macro variable to specify whether you want to retain table data if the SPD Server table creation process terminates abnormally.

Syntax

SPDSFSAV=YES|NO

Default: NO. Normally SAS closes and deletes tables that are not properly created.

Arguments

YES
: enables FORCESAVE mode and saves the table.

NO
: default SPD Server actions delete partially completed tables.

Description

Large tables can require a long time to create. If problems such as network interruptions or disk space shortages occur during this time period, the table might not be properly created and signal an error condition. If SAS encounters such an error condition, it deletes the partially completed table.

In SPD Server you can set SPDSFSAV=YES. Saving the partially created table can protect the time and resources invested in a long-running job. When the SPDSFSAV= macro variable is set to YES, the SPD Server LIBNAME proxy saves partially completed tables in their last state and identifies them as damaged tables.

Marking the table damaged prohibits other SAS DATA or PROC steps from accessing the table until its state of completion can be verified. After you verify or repair a table, you can clear the 'damaged' status and enable further read/update/append operations on the table. Use the PROC DATASETS REPAIR operation to remove the damaged file indicator.

Example

Configure SPD Server before you run the table creation job for a large table called ANNUAL. If some error prevents the successful completion of the table ANNUAL, the partially completed table will be saved.

```
%let SPDSFSAV=YES;
DATA SPDSLIB.ANNUAL;
...
RUN;
```

SPDSEINT=

Use the SPDSEINT= macro to specify how SPD Server responds to network disconnects during SQL pass-through EXECUTE() statements.

Syntax

SPDSEINT=YES|NO

Default: YES

Description:

The SPD Server SQL server interrupts SQL processing by default when a network failure occurs . The interruption prematurely terminates the EXECUTE() statement. Setting SPDSEINT=NO configures the SPD Server's SQL server to continue processing until completion regardless of network disconnects.

Warning: Use the macro variable setting SPDSEINT=NO carefully! A runaway EXECUTE() statement requires a privileged system user on the server machine to kill the SPD Server SQL proxy process. This is the only way to stop the processing.

Variables for Sorts

SPDSBSRT=

Use the SPDSBSRT= macro variable to configure SPD Server's sorting behavior when it encounters a BY-clause and there is no index available.

Syntax

SPDSBSRT=YES|NO

Default:YES

Corresponding Table Option:BYSORT=

Arguments

YES
: SPD Server performs a server sort when it encounters a BY clause and there is no index available.

NO
: SPD Server does not perform a sort when it encounters a BY clause.

Description

Base SAS software requires an explicit PROC SORT statement to sort SAS data. In contrast, SPD Server sorts a table whenever it encounters a BY clause, if it determines that the table has no index.

Advantages for using SPD Server implicit sorts are discussed in detail in the Help section for “Additional LIBNAME Options ” on page 27.

Example 1

At the start of a session to run old SAS programs, you realize that you do not have time to remove the existing PROC SORT statements. These statements are present only to generate print output.

To avoid redundant Server sorts, configure SPD Server to turn off implicit sorts. Put the macro variable assignment in your autoexec.sas file so SPD Server retains the configuration for all job sessions.

%let SPDSBSRT=NO;

During the Example 1 session you decide to run a new program that has no PROC SORT statements. Instead, the new program takes advantage of SPD Server implicit sorts.

```
data inventory.old_autos;
   input
      year $4.
      @6 manufacturer $12.
      model $10.
      body_style $5.
      engine_liters
      @39 transmission_type $1.
      @41 exterior_color $10.
      options $10.
      mileage condition;
   datalines;

1971 Buick       Skylark   conv  5.8 A yellow  00000001 143000 2
1982 Ford        Fiesta    hatch 1.2 M silver  00000001  70000 3
1975 Lancia      Beta      2door 1.8 M dk blue 00000010  80000 4
1966 Oldsmobile  Toronado  2door 7.0 A black   11000010 110000 3
1969 Ford        Mustang   sptrf 7.1 M red     00000111 125000 3
;

PROC PRINT data=inventory.old_autos
; by model;

run;
```

When the code executes, the PRINT procedure returns an error message. What happened? SAS expected INVENTORY.OLDAUTOS to be sorted before it would generate print output. Since there is no PROC SORT statement -- and implicit sorts are still turned off -- the sort does not occur.

Example 2

Keep implicit sorts turned off for the session, but specify an implicit sort for the table INVENTORY.OLDAUTOS.

```
 PROC PRINT data=inventory.oldautos(bysort=yes);
 by model;
 run;
```

SPDSNBIX=

Use the SPDSNBIX= macro variable to configure whether to use an index during a BY-sort.

Syntax

SPDSNBIX=YES|NO

Default: NO

Corresponding Server Parameter Option: [NO]BYINDEX

Arguments

YES
: Set SPDSNBIX=YES to suppress index use during a BY-sort. If the distribution of the values in the table are not relatively sorted or clustered, using the index for the BY sort can result in poor performance.

NO
: Set SPDSNBIX=NO or use the default value to allow the [NO]BYINDEX server parameter option to determine whether to use an index for a BY sort.

Example

```
%let SPDSNBIX=YES;
```

SPDSSTAG=

Use the SPDSSTAG= macro variable to specify whether to use non-tagged or tagged sorting for PROC SORT or BY processing.

Syntax

SPDSSTAG=YES|NO

Default:NO

Arguments

YES
performs tagged sorting.

NO
performs non-tagged sorting.

Description

During a non-tagged sort, SPD Server attaches the entire table column to the key field(s) to be sorted. Non-tagged sorting allows the software to deliver better performance than a tagged sort. Non-tagged sorting also requires more temporary disk space than a tagged sort.

Example

You are running low on disk space and you do not know whether you have enough disk overhead to accommodate the extra sort space required to support a non-tagged sort operation.

Configure SPD Server to perform a tagged sort.

```
%let SPDSSTAG=YES;
```

Variables for WHERE Clause Evaluations

SPDSTCNT=

Use the SPDSTCNT= macro variable to specify the number of threads that you want to use during WHERE clause evaluations.

Syntax

SPDSTCNT=*n*

Default: The value of MAXWHTHREADS is configured by SPD Server parameters.

Used in Conjunction with the SPD Server Parameter: MAXWHTHREADS

Corresponding Table Option: THREADNUM=

Arguments

n
is the number of threads.

Description

See “THREADNUM=” on page 262 for a description and an explanation of how SPDSTCNT= interacts with the SPD Server parameter MAXWHTHREADS.

SPDSEV1T=

Use the SPDSEV1T= macro variable to indicate whether data returned from an SPD Server WHERE clause evaluations should be in strict row (observation) order.

The macro variables SPDSEV1T= and SPDSEV2T= work in conjunction with the SPD Server WHERE clause planner WHINIT.

The variables SPDSEV1T= and SPDSEV2T= are identical in purpose. You use them to specify the row order of data returned in WHERE-processing. Which variable the server exercises depends on the evaluation strategy selected by WHINIT. The SPDSEV1T= evaluation strategy is indexed. The SPDSEV2T= evaluation strategy is non-indexed. Avoid using these options unless you absolutely understand the SPD Server performance tradeoffs that depend on maintaining the order of data.

If compatibility with Base SAS software is important, set both SPDSEV1T= and SPDSEV2T= to 0. When both evaluation strategies are set to 0, SPD Server returns data in row order whether the SPDSEV1T= or the SPDSEV2T= strategy is selected.

When you use a SAS PROC to retrieve rows from a sorted table, some SAS PROCs can use the sort order information to optimize how to receive and process the rows. For example, if you use PROC SQL to perform table joins on a sorted table that uses WHERE predicates to filter table rows, then PROC SQL will use the sort order information to optimize the join strategy. If you use the default values of SPDSEV1T= and SPDSEV2T= in these instances, the SAS PROC receives the table rows in sorted order.

If the SAS PROC you submit does *not* utilize the sorted order, the default values of SPDSEV1T= and SPDSEV2T= will restrict the use of parallel WHERE clauses, which can negatively impact performance. For example, PROC PRINT and most SAS data step code does not take advantage of sorted tables. If you know that the SAS PROC you are submitting does not take advantage of a sorted table, you can change the setting for SPDSEV1T= or SPDSEV2T= to 2, in order to allow parallel WHERE evaluations that can improve performance. However, this should be done with care: a parallel WHERE evaluation does not guarantee that rows are returned to SAS in sorted order, and this can cause incorrect results for a SAS PROC that uses that information.

Note: The SPDSEV1T= and SPDSEV2T= usage that is discussed here does not apply to SQL statements that are executed via the SPD Server pass-through SQL facility.

Syntax

SPDSEV1T=0|1|2

Default: 1

Used in Conjunction with Indexed WHERE clause Evaluation Strategy

Arguments

0

returns data in row order.

1

might not return the data in row order. SPD Server can override as needed to force a 0 setting if the table is sorted using PROC SORT.

2

always forces parallel evaluation regardless of sorted order. May not return data in row order.

Description

If SPD Server must return many rows during WHERE clause processing, setting the variable to **0** will greatly slow performance. Use **0** only when row order is required. Use **2** only when you know row order is not important to the result.

Example

Configure SPD Server to send back data in row order whenever WHINIT performs an EVAL1 evaluation.

```
%let SPDSEV1T=0;
```

SPDSEV2T=

Use the SPDSEV2T= macro variable to specify whether the data returned from WHERE clause evaluations should be in strict row (observation) order.

The macro variables SPDSEV1T= and SPDSEV2T= work in conjunction with the SPD Server WHERE clause planner WHINIT.

The variables SPDSEV1T= and SPDSEV2T= are identical in purpose. You use them to specify the row order of data returned in WHERE-processing. Which variable the server exercises depends on the evaluation strategy selected by WHINIT. The SPDSEV1T= evaluation strategy is indexed. The SPDSEV2T= evaluation strategy is non-indexed. Avoid using these options unless you absolutely understand the SPD Server performance tradeoffs that depend on maintaining the order of data.

If compatibility with Base SAS software is important, set both SPDSEV1T= and SPDSEV2T= to 0. When both evaluation strategies are set to 0, SPD Server returns data in row order whether the SPDSEV1T= or the SPDSEV2T= strategy is selected.

When you use a SAS PROC to retrieve rows from a sorted table, some SAS PROCs can use the sort order information to optimize how to receive and process the rows. For example, if you use PROC SQL to perform table joins on a sorted table that uses WHERE predicates to filter table rows, then PROC SQL will use the sort order information to optimize the join strategy. If you use the default values of SPDSEV1T= and SPDSEV2T= in these instances, the SAS PROC receives the table rows in sorted order.

If the SAS PROC you submit does *not* utilize the sorted order, the default values of SPDSEV1T= and SPDSEV2T= will restrict the use of parallel WHERE clauses, which can negatively impact performance. For example, PROC PRINT and most SAS data step code does not take advantage of sorted tables. If you know that the SAS PROC you are submitting does not take advantage of a sorted table, you can change the setting for SPDSEV1T= or SPDSEV2T= to 2, in order to allow parallel WHERE evaluations that can improve performance. However, this should be done with care: a parallel WHERE evaluation does not guarantee that rows are returned to SAS in sorted order, and this can cause incorrect results for a SAS PROC that uses that information.

Note: The SPDSEV1T= and SPDSEV2T= usage that is discussed here does not apply to SQL statements that are executed via the SPD Server pass-through SQL facility.

Syntax

SPDSEV2T=0|1|2

Default:1

Used in Conjunction with Non-Indexed WHERE clause Evaluation Strategy

Arguments

0
: returns data in row order.

1
: might not return the data in row order. SPD Server can override as needed to force 0 setting if the table is sorted using PROC SORT.

2
: always forces parallel evaluation regardless of sorted order. May not return the data in row order.

Description

If SPD Server must return many rows during WHERE clause processing, setting the variable to **0** will greatly slow performance. Use **0** only when row order is required. Use **2** only when you know row order is not important to the result.

Example

Configure SPD Server to send back data in row order whenever WHINIT performs an EVAL2 evaluation.

```
%let SPDSEV2T=0;
```

SPDSWDEB=

Use the SPDSWDEB= macro variable to specify whether the WHERE clause planner WHINIT, when evaluating a WHERE expression, should display a summary of the execution plan.

Syntax

SPDSWDEB=YES|NO

Default: NO

Arguments

YES
: displays WHINIT's planning output.

NO
: suppresses WHINIT's planning output.

SPDSIRAT=

Use the SPDSIRAT= macro variables to specify whether to perform segment candidate pre-evaluation when performing WHERE clause processing with hybrid indexes.

Syntax

SPDSIRAT=0..100

Default: MAXSEGRATIO server parameter

Description:

When using hybrid indexes, WHERE-based queries pre-evaluate segments. The segments are scanned for candidates that match one or more predicates in the WHERE clause. The candidate segments that were identified during the pre-evaluation are queried in subsequent logic to evaluate the WHERE clause. Eliminating the non-candidate segments from the WHERE clause evaluation generally results in substantial performance gains.

Some queries can benefit by limiting the pre-evaluation phase. SPD Server imposes the limit based on a ratio: the number of segments that contain candidates compared to the total number of segments in the table. The reason for this is simple. If the predicate has candidates in a high percentage of the segments, the pre-evaluation work is largely wasted.

The ratio formed by dividing the number of segments that containing candidates by the number of total segments is compared to a cutoff point. If the segment ratio is greater than the value assigned to the cutoff point, the extra processing required to perform pre-evaluation outweighs any potential process savings that might be gained through the predicate pre-evaluation. SPD Server calculates the ratio for a given predicate and compares the ratio to the SPDSIRAT= value, which acts as the cutoff point. If the calculated ratio is less than or equal to the SPDSIRAT= value, pre-evaluation is performed. If the calculated ratio is greater than the SPDSIRAT= value, pre-evaluation is skipped and every segment is a candidate for the WHERE clause.

Use the global SPD Server parameter, MAXSEGRATIO to set the default cutoff value. The default MAXSEGRATIO should provide good performance. Certain specific query situations might be justification for modifying your SPDSIRAT= value. When you modify your SPDSIRAT= value, it overrides the default value established by MAXSEGRATIO.

Example:

Configure SPD Server to perform a pre-evaluation phase for WHERE clause processing with hybrid indexes if the candidates are in 65% or less of the segments.

```
%let SPDSIRAT=65;
```

SPDSNIDX=

Use the SPDSNIDX= macro variable to specify whether to use the table's indexes when processing WHERE clauses. SPDSNIDX= can also be used to disable index use for BY-order determination.

Syntax

SPDSNIDX=YES|NO

Default: NO

Corresponding Table Option: NOINDEX=

Arguments

YES
: ignores indexes when processing WHERE clauses.

NO
: uses indexes when processing WHERE clauses.

Description:

Set SPDSNIDX=YES to test the effect of indexes on performance or for specific processing. Do not use YES routinely for normal processing.

Example:

Assume you are processing data from SPORT.MAILLIST. There is an index for the SEX column, and you should test it to determine whether the index will improve performance when you use PROC PRINT processing on SPORT.MAILLIST.

You should configure SPD Server not to use the index:

```
data sport.maillist;
  input
```

```
    name $ 1-20
    address $ 21-57
    phoneno $ 58-69
    sex $71;

datalines;

Douglas, Mike       3256 Main St., Cary, NC 27511        919-444-5555 M
Walters, Ann Marie  256 Evans Dr., Durham, NC 27707      919-324-6786 F
Turner, Julia       709 Cedar Rd., Cary, NC 27513        919-555-9045 F
Cashwell, Jack      567 Scott Ln., Chapel Hill, NC 27514 919-533-3845 M
Clark, John         9 Church St.,  Durham, NC 27705      919-324-0390 M
;

PROC DATASETS lib=sport nolist;
modify maillist;
index create sex;
quit;

/*Turn on the macro variable SPDSWDEB */
/* to show that the index is not used */
/* during the table processing.       */

%let spdswdeb=YES;

%let spdsnidx=YES;

title "All Females from Current Mailing List";
PROC PRINT data=sport.maillist;
where sex="F";
run;

%let spdsnidx=NO;
```

SPDSWCST=

Use the SPDSWCST= macro variable to specify whether to use dynamic WHERE clause costing.

Syntax

SPDSWCST=YES|NO

Default:NO

Corresponding Server Parameter Option: [NO]WHERECOSTING

Turns WHERE-costing on or off for an entire server.

Description:

Set SPDSWCST=YES to use dynamic WHERE clause costing. Disabling SPDSWCST= defaults SPD Server to using WHERE-costing with WHINIT.

Example:

%let SPDSWCST=YES;

SPDSWSEQ=

Syntax

SPDSWSEQ=YES|NO

Default:NO

Description:

Set the SPDSWSEQ= macro variable to YES. When set to YES, the SPDSWSEQ= macro variable overrides WHERE clause costing and forces a global EVAL3 or EVAL4 strategy.

Example:

%let SPDSWSEQ=YES;

Variables That Affect Disk Space

SPDSCMPF=

Use the SPDSCMPF= macro variable to specify the amount of growth space, sized in bytes, to be added to a compressed data block.

Syntax

SPDSCMPF=*n*

Default:0 bytes

Arguments

n
: is the number of bytes to add.

Description

Updating rows in compressed tables can increase the size of a given table block. Additional space is required for the block to be written back to disk. When contiguous space is not available on the hard drive, a new block fragment stores the excess, updated quantity. Over time, the table will experience block fragmentation.

When opening compressed tables for OUTPUT or UPDATE, you can use the SPDSCMPF= macro variable to anticipate growth space for the table blocks. If you estimate correctly, you can greatly reduce block fragmentation in the table.

Note: SPD Server table metadata does not retain compression buffer or growth space settings.

SPDSDCMP=

Use the SPDSDCMP= macro variable to compress SPD Server tables that are stored on disk.

Syntax

SPDSDCMP=YES|NO

Default:NO

Use in Conjunction with Table Option: IOBLOCKSIZE=

Corresponding Table Option: COMPRESS=

Arguments

YES
performs the run-length compression algorithm SPDSRLLC.

NO
performs no table compression.

Description

When you set the SPDSDCMP= macro variable to YES, SPD Server compresses newly created tables by 'blocks' according to the algorithm specified. To control the amount of compression, use the table option IOBLOCKSIZE= to specify the number of rows that you want to store in the block. For a complete discussion, refer to "IOBLOCKSIZE=" on page 256.

Note: Once a compressed table is created, you cannot change its block size. To resize the block, you must PROC COPY the table to a new table, setting IOBLOCKSIZE= to the new block size for the output table.

Example

Before creating huge tables, you want to conserve disk space. Specify compression, and the default algorithm SPDSRLLC, at the beginning of your job.

```
%let SPDSDCMP=YES;
```

SPDSIASY=

Use the SPDSIASY= macro variable to specify whether to create indexes in parallel when creating multiple indexes on an SPD Server table.

Syntax

SPDSIASY=YES|NO

Default:NO

Corresponding Table Option : ASYNCINDEX=

Arguments

YES
creates the indexes in parallel.

NO
creates one index at a time.

Description

You use the macro variable SPDSIASY= to choose between parallel and sequential index creation on SPD Server tables with more than one index. One advantage of creating multiple indexes in parallel is speed. The speed enhancements that can be achieved with parallel indexes are not free. Parallel indexes require significantly more disk space for working storage. The default SPD Server setting for the SPDSIASY= macro variable is set to NO, in order to avoid exhausting the available work storage space.

However, if you have adequate disk space to support parallel sorts, **it is strongly recommended** that you override the default SPDSIASY=NO setting and assign

SPDSIASY=YES. You can substantially increase performance -- indexes that take hours to build complete much faster.

How many indexes should you create in parallel? The answer depends on several factors, such as the number of CPUs in the SMP configuration and available storage space needed for index key sorting.

When managing disk space on your SPD Server, remember that grouping **`index create`** statements can minimize the number of table scans that SPD Server performs, but it also affects disk space consumption. There is an inverse relationship between the table scan frequency and disk space requirements. A minimal number of table scans requires more auxiliary disk space; a maximum number of table scans requires less auxiliary disk space.

Example

Your perform batch processing from midnight to 6:00 a.m. All of your processing must be completed before start of the next work day. One frequently repeated batch job creates large indexes on a table, and usually takes several hours to complete. Configure SPD Server to create indexes in parallel to reduce the processing time.

```
%let SPDSIASY=YES;
proc datasets lib=spds;
   modify a;
   index create x;
   index create y;
   modify a;
   index create comp=(x y) comp2=(y x);
   quit;
```

In the example above, the X and Y indexes will be created in parallel. After creating X and Y indexes, SPD Server creates the COMP and COMP2 indexes in parallel. In this example, two table scans are required: one table scan for the X and Y indexes, and a second table scan for the COMP and COMP2 indexes.

SPDSSIZE=

Use the SPDSSIZE= macro variable to specify the size of an SPD Server table partition.

Syntax

SPDSSIZE=*n*

Default:16 Megabytes

Corresponding Table Option: PARTSIZE=

Affected by LIBNAME option: DATAPATH=

Arguments

n

is the size of the partition in Megabytes.

Description

Use this SPDSSIZE= macro variable option to improve performance of WHERE clause evaluation on non-indexed table columns.

Splitting the data portion of a server table at fixed-sized intervals allows SPD Server to introduce a high degree of scalability for non-indexed WHERE clause evaluation. This is because SPD Server launches threads in parallel and can evaluate different partitions of the table without file access or thread contention. The speed enhancement comes at the cost of disk usage. The more data table splits you create, the more you increase the number of files, which are required to store the rows of the table.

Scalability limits on the SPDSSIZE= macro variable ultimately depend on how you structure the DATAPATH= option in your LIBNAME statement. The configuration of the DATAPATH= file systems across striped volumes is important. You should spread each individual volume's striping configuration across multiple disk controllers and SCSI channels in the disk storage array. Your configuration goal, at the hardware level, should be to maximize parallelism when performing data retrieval.

The SPDSSIZE= specification is also limited by MINPARTSIZE=, an SPD Server parameter maintained by the SPD Server administrator. MINPARTSIZE= ensures that an over-zealous SAS user cannot arbitrarily create small partitions, thereby generating an excessive number of physical files. The default for MINPARTSIZE= is 16 Mbytes.

Note: The SPDSSIZE= value for a table cannot be changed after the table is created. To change the SPDSSIZE=, you must PROC COPY the table and use a different SPDSSIZE= (or PARTSIZE=) option setting on the new (output) table.

For an example using the table option, see “PARTSIZE=” on page 254.

```
%let SPDSSIZE=32;
```

Variables to Enhance Performance

SPDSNETP=

Use the SPDSNETP= macro variable to size buffers in server memory for the network data packet.

Syntax

SPDSNETP=*size-of-packet*

Default:32K

Corresponding Table Option:“NETPACKSIZE= ” on page 257

Arguments

size-of-packet
is the size (integer) in bytes of the network packet.

Description

When sizing the buffer for data packet transfer between SPD Server and your SAS client machine, the packet must be greater than or equal in size to one table row. See “NETPACKSIZE= ” on page 257 for more information.

Example

Despite recent upgrades to your network connections, you are experiencing significant pauses when the SPD Server transfers data. You want to resize the data packet to send three rows at a time for a more continuous data flow.

Specify a buffer size in server memory that is three times the row size (6144 bytes.) Submit your SPDSNETP= macro variable statement at the top of your job.

```
%let SPDSNETP=18432;
```

SPDSSADD=

Use the SPDSSADD= macro variable to specify whether SPD Server appends tables by transferring a single row at a time synchronously, or by transferring multiple rows asynchronously (block row appends).

Syntax

SPDSSADD=YES|NO

Default:NO

Related Table Option: SYNCADD=

Arguments

YES
: applies a single row at a time during an append operation. This behavior imitates the Base SAS engine.

NO
: appends multiple rows at a time

Description

SPDSSADD=YES slows performance. Use this argument only if you require strict compatibility with Base SAS software when processing a table. For a complete discussion, refer to “SYNCADD=” on page 252.

SPDSSYRD=

Use the SPDSSYRD= macro variable to specify whether SPD Server should perform asynchronous data streaming when reading a table.

Syntax

SPDSSYRD=YES|NO

Default:NO

Related Table Option: SYNCREAD=

Arguments

YES
: enables asynchronous data streaming.

NO
: disables asynchronous data streaming.

Description

Use SPDSSYRD=YES only with a MODIFY statement. If you use it with any other processing operation, you slow performance.

SPDSAUNQ=

Use the SPDSAUNQ= macro variable setting to specify whether to cancel an append to a table if the table has a unique index and the append would violate the index uniqueness.

Syntax

SPDSAUNQ=YES|NO

Default:NO

Description:

Use SPDSAUNQ=YES macro variable to improve append performance to a table with unique indexes. If uniqueness is not maintained, the append is canceled and the table is returned to its state before the append. In such an instance, you can scrub the table to remove nonunique values and redo the append with the macro variable SPDSAUNQ= set to YES. The other alternative is to simply redo the append with the macro variable SPDSAUNQ= set to NO.

If SPDSAUNQ=NO, the SPD Server will enforce uniqueness at the expense of appending unique indexes in observation order one row at a time. If uniqueness is not maintained for any given row, that row is discarded from the append.

Variable for a Client and a Server Running on the Same UNIX Machine

SPDSCOMP=

specifies to compress the data when sending a data packet through the network.

Syntax

SPDSCOMP=YES|NO

Default: NO

Chapter 13 SPD Server LIBNAME Options

Introduction

All SAS users who want to use LIBNAME access to an SPD Server should read this chapter.

This chapter contains reference information for the SPD Server LIBNAME options. The options are grouped by the function or purpose of their default value. You can change the default, thereby controlling how they function in different data situations. The examples for using the options assume that a LIBNAME statement to access the SPD Server engine SASSPDS has previously been issued.

When using the options, remember that if a table option is used subsequent to a LIBNAME option of the same name, the value of the table option or macro variable takes precedence.

Options to Locate an SPD Server Host

HOST=

Summary

Specifies an SPD Server machine by node name or IP address, and locates the Name Server using the SERVICE value.

Syntax

`HOST=`*`hostname`* `<SERVICE=`*`service`*`>`

Arguments

hostname
: is the node name of the SPD Server machine or an IP address.

service
: is the name of a service or the port number for the SPD Server's Name Server.

Description

This option provides the node name of an SPD Server host machine and locates the port number of the SPD Server's Name Server. When there is no SERVICE= specification, SPD Server checks the client's **`/etc/services`** file (or its equivalent file) for SPDSNAME – a reserved name for the SPD Server's Name Server.

Examples

Specify the server machine SAMSON and use the default named service SPDSNAME to obtain the port number of the SPD Server Name Server.

```
LIBNAME mylib sasspds 'spdsdata'
   host='samson';
```

Specify the server machine SAMSON and provide the port number of the SPD Server Name Server.

```
LIBNAME mylib sasspds 'spdsdata'
   host='samson'
   service='5002';
```

Using a Macro Variable to Specify the SPD Server Host

Assign the macro variable SPDSHOST to the SPD Server host SAMSON so that the LIBNAME statement is not required to SAMSON.

```
%let spdshost=samson;
LIBNAME mylib sasspds 'spdsdata'
   user='yourid'
   password='swami';
```

SERVER=

Summary

Specifies an SPD Server host machine by node name, and locates the network address (port number) of the SPD Server Name Server.

Syntax

`SERVER=`*`hostname.servname`*

Arguments

hostname
: is the node name of the SPD Server host machine.

servname
: is the name of a service or the port number of the SPD Server Name Server.

Examples

Specify the SPD Server host machine SAMSON and use the default named service SPDSNAME to obtain the port number of the SPD Server Name Server.

```
LIBNAME mylib sasspds 'spdsdata'
   server=samson.spdsname;
```

Specify the SPD Server host machine SAMSON and give the port address of the SPD Server Name Server.

```
LIBNAME mylib sasspds 'spdsdata'
   server=samson.5002;
```

Options to Identify the SPD Server Client

ACLGRP=

Summary

Names an ACL group that has been previously assigned to the SPD Server user ID. The SPD Server system administrator sets up ACL groups and can assign a single user to up to five ACL groups.

Syntax

`ACLGRP=`*`aclgroup`*

Arguments

aclgroup
: Names the ACL group that the SPD Server Administrator assigned to your SPD Server user ID. (You can be assigned up to five ACL groups.)

Example

Specify the ACL group PROD.

```
LIBNAME mylib sasspds 'spdsdata'
   user='receiver'
   aclgrp='PROD'
   prompt=yes;
```

Note: Password values are case sensitive. If the SPD Server administrator assigns a lowercase password value, you must enter the password value in lowercase.

CHNGPASS=

Summary

Specifies whether to prompt an SPD Server user for a change of password. If ACL file security is enabled, SPD Server validates the old and new password against its user ID table.

Syntax

`CHNGPASS= YES | NO`

Arguments

YES
: prompts for a change of the SPD Server user password.

NO
: suppresses a prompt for a change of the SPD Server user password. This is the default.

Example

Specify a prompt to change the password of SPD Server user TEMPHIRE.

```
LIBNAME mylib sasspds 'spdsdata'
   user='temphire'
   password='whizbang'
   chngpass=yes;
```

Note: If you are using LDAP user authentication, and you create a user connection that uses the CHNGPASS= LIBNAME option, the user password will not be changed. If you are using LDAP authentication and want to change a user password, follow the operating system procedures to change a user password, and check with your LDAP server administrator to ensure that the LDAP database also records password changes.

NEWPASSWORD= or NEWPASSWD=

Summary

Specifies a new password for an SPD Server client user. If ACL file security is enabled, SPD Server validates the old or new password against its user ID table.

Syntax

```
NEWPASSWORD= newpassword
NEWPASSWD= newpassword
```

Arguments

newpassword
is the new password of an SPD Server client user. The password, visible in a SAS program, is encrypted in the SAS log file.

Example

Specify a new password **rambo** for SPD Server client user RECEIVER.

```
LIBNAME mylib sasspds 'spdsdata'
   user='receiver'
   password='whizbang'
   newpassword='rambo';
```

Note: If you are using LDAP user authentication, and you create a user connection that uses the NEWPASSWORD= LIBNAME option, the user password will not be changed. If you are using LDAP authentication and want to change a user password, follow the operating system procedures to change a user password, and check with your LDAP server administrator to ensure that the LDAP database also records password changes.

PASSWORD= or PASSWD=

Summary

Specifies the SPD Server password of an SPD Server client user. If ACL file security is enabled, SPD Server validates the password against its user ID table.

Syntax

```
PASSWORD='password'
PASSWD='password'
```

Arguments

'password'
: is the case-sensitive password of an SPD Server client user. The password, visible in a SAS program, is encrypted in the SAS log file.

Example

Specify the password **whizbang** for SPD Server client user SPDSUSER.

```
LIBNAME mylib sasspds 'spdsdata'
   server=kaboom.5200
   user='spdsuser'
   password='whizbang';
```

Options

SPD Server 4.5 supports the integration of the SAS 9.2 PROC PWENCODE. This permits scripts to be generated that do not explicitly contain secure passwords that could easily be used without authorization. You must run PROC PWENCODE in Base SAS to enable the usage of script password encoding within SPD Server 4.5. See the Base SAS documentation for detailed instruction on running PROC PWENCODE for use with SPD Server 4.5.

The example below shows an SPD Server LIBNAME statement that uses the password encoding option:

```
LIBNAME mylib sasspds 'spdsdata'
   server=kaboom.5200
   user='spdsuser'
   password='{sas001}c3BkczEyMw==';
```

PROMPT=

Summary

Specifies whether to prompt an SPD Server user for a password. If ACL file security is enabled, SPD Server validates the password against its user ID table.

Syntax

```
PROMPT= YES | NO
```

Arguments

YES
: prompts an SPD Server user for a password.

NO
: suppresses a prompt for a password.

Example

Configure SPD Server to prompt SPD Server user BIGWHIG for a password.

```
LIBNAME mylib sasspds 'spdsdata'
   user='bigwhig'
   prompt=yes;
```

USER=

Summary

Specifies the ID of an SPD Server client user. If ACL file security is enabled, SPD Server validates the ID against its user ID table. (The SPD Server user ID defaults to the SAS process user ID if it is available, that is, when the client is not a Windows client.)

Syntax

`USER='`*`username`*`'`

Arguments

'*username*'
: is the ID of an SPD Server client user.

Example

Specify the identifier SPDSUSER for an SPD Server client user.

```
LIBNAME mylib sasspds 'spdsdata'
  user='spdsuser'
  prompt=yes;
```

Options to Specify Implicit SQL Pass-Through

IP=YES

Summary

This is an abbreviated specification which replaces the more verbose PASSTHRU= option. The IP=YES option draws on information specified in the LIBNAME declaration. The IP=YES option specifies an implicit SQL Pass-Through connection for a single user to a specified domain and server during a given SPD Server session.

Syntax

```
LIBNAME BOAF sasspds 'BOAF'
   server=kaboom.5200
   user='rcnye'
   password='*******'
   IP=YES ;
```

PASSTHRU=

Summary

This older and more verbose specification for IP=YES is still supported. It specifies an implicit SQL Pass-Through connection for a single user to a specified domain and server during a given SPD Server session.

Syntax

```
PASSTHRU=<'dbq=<SAS-data-library>
  <SPD Server-options>
  user=<'UserID'>
  password=<'password'> ;
```

Arguments

DBQ=*libname-domain* (required)
Specifies the primary SPD Server LIBNAME domain for the SQL Pass-Through connection. The name that you specify is identical to the LIBNAME domain name that you used when making a SAS LIBNAME assignment to **sasspds**. Use single or double quotes around the specified value.

SPD Server-options
one or more SPD Server options.

USER=*SPD Server user ID* (required on Windows, but not UNIX)
Specifies an SPD Server user ID in order to access an SPD Server SQL Server. Use single or double quotes around the specified value.

PASSWORD=*password* (required, or use PROMPT=YES, unless USER='anonymou')
Specifies an SPD Server user ID password to access an SPD Server. (This value is case sensitive.)

Example:

The following is a LIBNAME statement that specifies the implicit SQL Pass-Through option for user rcnye, using a libref to connect to the domain named 'BOAF' on the server named 'Kaboom' on port 5200:

```
LIBNAME BOAF sasspds 'BOAF'
   server=kaboom.5200
   user='rcnye'
   password='*******'

   PASSTHRU='
   dbq="BOAF"
   server=kaboom.5200
   user="rcnye"
   password="*******"' ;
```

Options

SPD Server 4.5 supports the integration of the SAS 9.2 PROC PWENCODE. This permits scripts to be generated that do not explicitly contain secure passwords that could easily be used without authorization. You must run PROC PWENCODE in Base SAS to enable the usage of script password encoding within SPD Server 4.5. See the Base SAS documentation for detailed instruction on running PROC PWENCODE with SPD Server 4.5.

The example below shows an SPD Server LIBNAME statement that uses the password encoding option:

```
LIBNAME mylib sasspds 'spdsdata'
   server=kaboom.5200
   user='spdsuser'
   password='{sas001}c3BkczEyMw=='

   PASSTHRU='
   dbq="spdsdata"
   server=kaboom.5200
   user="spdsuser"
   password="{sas001}c3BkczEyMw=="';
```

Options to Specify File Paths for Table Storage

SPD Server strongly recommends that your site administrator defines SPD Server domain options in the SPD Server **libnames.parm** configuration file. However, in unusual cases, such as the SPD Server administrator being temporarily unavailable, the following four LIBNAME options can be issued by an SPD Server user to define domains and table file storage paths.

CREATE=

Summary

Creates the primary directory for an SPD Server domain, if it does not already exist.

Syntax

`CREATE=YES | NO`

Arguments

YES
: creates the primary directory if it does not already exist.

NO
: fails the LIBNAME assignment if the primary directory does not already exist. This is the default setting.

Description

An SPD Server administrator defines the primary directory for the SPD Server domain in the LIBNAME parameter file. If CREATE= is set to YES, the software creates the directory (primary file system) in the event that an SPD Server administrator forgets to create it.

DATAPATH=

Summary

The SPD Server administrator for your site should use options in the SPD Server **libnames.parm** configuration file to define SPD Server domain options. However, if the SPD Server is temporarily unavailable, the following LIBNAME option can be issued by an SPD Server user to specify a list of initial or overflow paths to store data (.dpf) file partitions for an SPD Server table.

Syntax

DATAPATH=('*filesystem*' '*filesystem*'...)

Arguments

'*filesystem*'
: is a directory path for UNIX or Windows.

Example

Create partitions as needed by cycling through the directories specified, DATAFLOW1 directory on DISK1 and DATAFLOW2 directory on DISK2.

```
LIBNAME mylib sasspds 'spdsdata'
   datapath=('/disk1/dataflow1'
             '/disk2/dataflow2');
```

INDEXPATH=

Summary

The SPD Server administrator for your site should use options in the SPD Server **libnames.parm** configuration file to define SPD Server domain options. However, if the SPD Server is temporarily unavailable, the following LIBNAME option can be issued by an SPD Server user to specify a list of initial or overflow paths to store index (.hbx), (.idx), and (.aux) file partitions associated with an SPD Server table.

Syntax

INDEXPATH=('*filesystem*' '*filesystem*'...)

Arguments

'*filesystem*'
: is a directory path for UNIX or Windows.

Example

Create index file partitions as needed using the directories specified, IDXFLOW1 directory on DISK1 and IDXFLOW2 directory on DISK2.

```
LIBNAME mylib sasspds 'spdsdata'
   indexpath=('/disk1/idxflow1'
              '/disk2/idxflow2');
```

METAPATH=

Summary
The SPD Server administrator for your site should use options in the SPD Server **libnames.parm** configuration file to define SPD Server domain options. However, if the SPD Server is temporarily unavailable, the following LIBNAME option can be issued by an SPD Server user to specify a list of overflow paths to store metadata (.mdf) file partitions for an SPD Server table.

Syntax
METAPATH=('*filesystem*' '*filesystem*'...)

Arguments

'*filesystem*'
: is a directory path for UNIX or Windows.

Example
Create overflow metadata file partitions as needed using the directories specified, METAFLOW1 directory on DISK1 and METAFLOW2 directory on DISK2.

```
LIBNAME mylib sasspds 'spdsdata'
   metapath=('/disk1/metaflow1'
             '/disk2/metaflow2');
```

Options for Access Control Lists (ACLs)

ACLSPECIAL=

Summary
Grants special access to SPD Server resources in the LIBNAME domain to an SPD Server user. The SPD Server user must also be defined as 'special' by the SPD Server administrator.

Syntax
ACLSPECIAL=YES | NO

Arguments

YES
: grants special access (Read, Write, Alter, and Control permission) to all SPD Server resources in the domain.

NO
: denies special access (Read, Write, Alter, and Control permission) to all SPD Server resources in the domain.

Description
Grants special privileges to all SPD Server tables and associated indexes in the LIBNAME domain. The special privileges, (Read, Write, Alter, and Control permission), override

normal ACL restrictions only if the SPD Server administrator defines the user as 'special' in the user ID table.

Example

Grant special privileges to THEBOSS allowing him to Read, Write, Alter, and Control all tables in the CONVERSION_AREA domain. (The SPD Server administrator has defined THEBOSS as 'special'.)

```
LIBNAME mydatalib sasspds 'conversion_area'
   server=husky.5105
   user='theboss'
   prompt=yes
   aclspecial=yes ;
```

Options for a Client and Server Running on the Same UNIX Machine

NETCOMP=

Summary

Compresses the data stream for an SPD Server network packet.

Syntax

`NETCOMP=YES | NO`

Arguments

YES
: sends compressed data in an SPD Server network packet.

NO
: sends uncompressed data in an SPD Server network packet.

Description

Normally, data compression for inter-process transfers is recommended. However, for a client and server process on the same machine -- with UNIXDOMAIN=YES -- turning off compression can improve performance. You should examine NETCOMP together with UNIXDOMAIN and NETPACKSIZE for both client and server on the same machine.

Example

Specify to turn off compression of the data stream.

```
LIBNAME mylib sasspds 'test_area'
   netcomp=no;
```

UNIXDOMAIN=

Summary
Specifies the use of UNIX domain sockets for data communication between an SPD Server and client process **running on the same machine**. (Not available in Windows.)

Syntax
`UNIXDOMAIN=YES | NO`

Arguments

YES
: uses AF_UNIX domain sockets for client/server data communication.

NO
: uses the default AF_INET domain sockets for client/server data communication.

Description
When UNIXDOMAIN=YES, SPD Server uses AF_UNIX domain sockets rather than the customary AF_INET domain sockets for data communication. AF_UNIX sockets typically are much faster and greatly enhance performance but are possible only for cases where client and server are running on the same machine. You should also examine NETCOMP and NETPACKSIZE parameters for possible use to enhance performance in conjunction with UNIXDOMAIN.

Example
You find that using the AF_UNIX sockets for your session that is running on the same machine as the SPD Server is not faster. Configure SPD Server to use the default AF_INET sockets instead.

```
LIBNAME mylib sasspds 'test_area'
   unixdomain=no;
```

Note: If you are running SPD Server 4.5 or later, and the client and server are both running UNIX, SPD Server automatically detects UNIX domain sockets. In such cases, it is not necessary to specify the UNIXDOMAIN parameter for optimum performance.

Options for Other Functions

BYSORT=

Summary
Specifies whether to use implicit automatic SPD Server sorts on BY clauses.

Syntax
`BYSORT=YES | NO`

Arguments

YES
: performs an implicit sort for a BY clause. This is the default.

NO
: does not perform an implicit sort for a BY clause.

Description

Where Base SAS software requires an explicit sort statement (PROC SORT) to sort SAS data, by default, SPD Server performs a sort whenever it encounters a BY clause. If the value of the BYSORT= option is NO, the SPD Server software performs the same as the Base SAS engine.

Example 1

Specify to turn off implicit SPD Server sorts for the session.

```
LIBNAME mydatalib sasspds 'conversion_area'
   server=husky.5105
   user='siteusr1'
   prompt=yes
   bysort=no ;

data mydatalib.old_autos;
   input
     year $4.
     @6 manufacturer $12.
     model $10.
     body_style $5.
     engine_liters
     @39 transmission_type $1.
     @41 exterior_color $10.
     options $10.
     mileage condition ;
   datalines ;

1971 Buick       Skylark   conv  5.8  A  yellow    00000001 143000 2
1982 Ford        Fiesta    hatch 1.2  M  silver    00000001  70000 3
1975 Lancia      Beta      2door 1.8  M  dk blue   00000010  80000 4
1966 Oldsmobile  Toronado  2door 7.0  A  black     11000010 110000 3
1969 Ford        Mustang   sptrf 7.1  M  red       00000111 125000 3
;

PROC PRINT data=mydatalib.old_autos;
   by model;
run;
```

In this program, the PRINT procedure will return an error message because the table MYDATALIB.OLD_AUTOS is not sorted.

Example 2

Turn off implicit SPD Server sorts with the LIBNAME option, but specify a server sort for the table MYDATALIB.OLD_AUTOS using the BYSORT table option.

```
PROC PRINT data=mydatalib.old_autos
 (bysort=yes);
```

```
  by model;
run;
```

DISCONNECT=

Summary

The DISCONNECT= option is used to control how user proxy resources are assigned for an SPD Server user. Each SPD Server user in a SAS session requires an SPD Server user proxy process to handles client requests.

Syntax

```
DISCONNECT=YES | NO
```

Arguments

YES
: closes network connections between the SAS client and SPD Server when all SPD Server librefs are cleared.

NO
: closes network connections between the SAS client and SPD Server only when the SAS session ends. This is the default setting.

Description

The DISCONNECT= option is used to control how user proxy resources are created and terminated for an SPD Server user. Each SPD Server user in a SAS session requires an SPD Server user proxy process to handles client requests.

The DISCONNECT= state of the user proxy is determined by the first LIBNAME statement a user issues in the SAS session.

When the DISCONNECT= option is set to NO, the network connections between the SAS client and the SPD Server user proxy are closed when the SAS session ends. Closing the network connection ends all SPD Server user proxy processes for that session.

When the DISCONNECT= option is set to YES, the network connections between the SAS client and the SPD Server user proxy are closed after the user's last SPD Server libref in the SAS session is cleared. Closing the network connection ends all SPD Server user proxy processes, but not necessarily the SAS session. If the user issues a subsequent SPD Server libref in that SAS session, a new SPD Server user proxy process must be started up.

The advantage of using DISCONNECT=NO is that the processor overhead that is required to create an SPD Server user proxy is only required when an SPD Server user issues his first LIBNAME of his session. The disadvantage of using DISCONNECT=NO is that the SPD Server user proxy does not terminate until the user's SAS session ends. For example, if a user does not log out at the end of the day and leaves an SPD Server session running overnight, the user proxy remains in force, occupying system resources that might be utilized by other jobs.

The advantage of using DISCONNECT=YES is that user resources are freed as soon as the user's last LIBNAME of the session is cleared. The disadvantage of using DISCONNECT=YES is if the user needs to issue a subsequent LIBNAME in that session, the LIBNAME assignment will require a new SPD Server user proxy to be launched.

The DISCONNECT=YES LIBNAME option must be used with the LIBNAME CLEAR statement to be effective.

The default setting for the DISCONNECT= option is NO.

Example 1

Use the default setting of DISCONNECT=NO to retain the user proxy process. Libref SPUD is assigned using user proxy process 8292, and then libref SPUD is cleared. Then libref CAKE is assigned, still using user proxy process 8292. The user proxy process is not terminated when libref SPUD is cleared, and no new user proxy process is required to assign libref CAKE.

```
LIBNAME spud sasspds 'potatoes'
   server=husky.6100
   user='bob'
   passwd='bob123';

NOTE: Libref SPUD was successfully assigned as follows:
      Engine:        SASSPDS
      Physical Name: :8292/spds/test/potatoes/

LIBNAME spud clear;

LIBNAME cake sasspds 'carrots'
   server=husky.6100
   user='bob'
   passwd='bob123';

NOTE: Libref CAKE was successfully assigned as follows:
      Engine:        SASSPDS
      Physical Name: :8292/spds/test/carrots/
```

Example 2

Use the DISCONNECT=YES setting to terminate the user proxy process when the last user LIBNAME is cleared. Libref SPUD is user Bob's last open LIBNAME. SPUD is assigned using user proxy process 8234, and then cleared. Next, libref CAKE is assigned using user proxy process 8240. When libref SPUD is cleared, user proxy process 8234 is terminated, and the resources that were allocated to proxy process 8324 are freed. When Bob submits a subsequent libref statement for CAKE, a new user proxy process 8240 is created.

```
LIBNAME spud sasspds 'potatoes'
   server=husky.6100
   user='bob'
   passwd='bob123'
DISCONNECT=YES;

NOTE: Libref SPUD was successfully assigned as follows:
      Engine:        SASSPDS
      Physical Name: :8234/spds/test/potatoes/

LIBNAME spud clear;

LIBNAME cake sasspds 'carrots'
   server=husky.6100
   user='bob'
   passwd='bob123'
DISCONNECT=YES;
```

```
NOTE: Libref CAKE was successfully assigned as follows:
      Engine:        SASSPDS
      Physical Name: :8240/spds/test/carrots/
```

Now Bob has libref CAKE assigned using user proxy process 8240. Suppose Bob makes another libref FRUIT without first clearing the CAKE libref. The libref FRUIT will re-use the active proxy process 8240. In this case, both the CAKE and FRUIT librefs must be cleared before the user proxy process can terminate.

```
LIBNAME fruit sasspds 'apples'
   server=husky.6100
   user='bob'
   passwd='bob123'
DISCONNECT=YES;

NOTE: Libref FRUIT was successfully assigned as follows:
      Engine:        SASSPDS
      Physical Name: :8240/spds/test/apples/
```

ENDOBS=

Summary

Specifies the end row (observation) number in a user-defined range for processing.

Syntax

`ENDOBS=`*n*

Arguments

n
: is the number of the end row.

Description

By default SPD Server processes the entire table unless the user specifies a range of rows with the STARTOBS= and ENDOBS= options. If the STARTOBS= option is used without the ENDOBS= option, the implied value of ENDOBS= is the end of the table. When both options are used together, the value of ENDOBS= must be greater than STARTOBS=.

In contrast to the Base SAS software options FIRSTOBS= and OBS=, the STARTOBS= and ENDOBS= SPD Server options can be used for WHERE clause processing in addition to table input operations.

Example 1

Specify for SPD Server to process only row numbers (observations) 200 - 500 while the LIBNAME is active.

```
LIBNAME mydatalib sasspds 'conversion_area'
   server=husky.5105
   user='siteusr1'
   prompt=yes
   startobs=200
   endobs=500;
```

LIBGEN=

Summary

The LIBGEN=YES option is used in explicit SQL connection statements. When you set LIBGEN= yes, SPD Server is configured to generate additional domain connections that enable you to perform SQL joins across different SPD Server domains.

Syntax

```
LIBGEN=YES
```

Description

You should specify the LIBGEN=YES option in the explicit SQL LIBNAME connection. You cannot specify the LIBGEN=YES option setting without first creating a LIBNAME connections to the domain.

Examples

The two code examples that follow both perform the same task. Both examples use explicit SQL to join two tables from different domains. The first example uses execute connection statements to facilitate joining the tables from separate domains. The second example uses the LIBGEN=YES option to perform the same join without having to issue the extra execute connection statements.

SQL without LIBGEN=YES

```
/* The example code without LIBGEN=YES */
/* must issue execute connection       */
/* statements to access tables that    */
/* reside in two different domains.    */

LIBNAME path1 sasspds 'path1'
  server=boxer.5140
  ip=YES
  user='anonymous' ;

LIBNAME path2 sasspds 'path2'
  server=boxer.5140
  ip=YES
  user='anonymous' ;

DATA path1.table1
  (keep=i table1)
path2.table2
  (keep=i table2) ;

table1 = 'table1' ;
table2 = 'table2' ;

do i = 1 to 10 ;
  output ;
  end ;
run ;
```

```
PROC SQL ;
CONNECT to sasspds (
  dbq='Path1'
  server=boxer.5140
  user='anonymous') ;

/* Without LIBGEN=YES, you must make  */
/* two execute connect statements.    */

execute (LIBREF path1 engopt="dbq='path1'")
  by sasspds;
execute (LIBREF path2 engopt="dbq='path2'")
  by sasspds;

execute
  (create table table4 as
   select *
   from
     path1.table1 a,
     path2.table2 b
   where a.i = b.i)
 by sasspds ;

disconnect from sasspds ;

quit ;
```

SQL with LIBGEN=YES

```
/* The example code that uses LIBGEN=YES   */
/* can join the tables from two different  */
/* domains in a more simple manner.        */

LIBNAME path1 sasspds 'path1'
  server=boxer.5140
  LIBGEN=YES
  ip=YES
  user='anonymous' ;

LIBNAME path2 sasspds 'path2'
  server=boxer.5140
  LIBGEN=YES
  ip=YES
  user='anonymous' ;

DATA path1.table1
  (keep=i table1)
path2.table2
  (keep=i table2) ;

table1 = 'table1' ;
table2 = 'table2' ;

do i = 1 to 10 ;
```

```
  output ;
  end ;
run ;

PROC SQL ;
CONNECT to sasspds (
  dbq='Path1'
  server=boxer.5140
  user='anonymous') ;

/* Syntax used with LIBGEN=YES option */

execute
 (create table table4 as
  select *
  from
   path1.table1 a,
   path2.table2 b
  where a.i = b.i)
by sasspds ;

disconnect from sasspds ;

quit ;
```

LOCKING=

Overview of Record-Level Locking

Record-level locking is an SPD Server feature that allows multiple users concurrent Read and Write access to SPD Server tables while maintaining the integrity of the table contents. When record-level locking is enabled, users can insert, append, delete, and update the contents of an SPD Server table while performing concurrent reads on the table. When a client enables record-level locking, the client connects to the single SPD Server record-level locking proxy process. When record-level locking is not enabled, clients connect to separate SPD Server user proxy processes for each LIBNAME connection to a domain.

Record-Level Locking Details

Record-level locking is enabled when an SPD Server client specifies the LOCKING=YES LIBNAME option to the client's LIBNAME connection statement. All subsequent operations on the given LIBNAME domain will use record-level locking. The primary use of record-level locking is to allow multiple clients or parallel operations from the same client to have both Read and Write access to the same SPD Server table resource. If record-level locking is not enabled, then any Write operation (update, append, insert, or delete) on an SPD Server table requires exclusive access to the resource, or a member lock failure error occurs. Operations that affect metadata, such as creating or deleting indexes, renaming variables, and renaming tables require exclusive access to the resource, whether record-level locking is enabled or not. These types of operations will report a member lock failure error when record-level locking is enabled, but exclusive access is not available.

Record-level locking must be enabled in SPD Server before a SAS client can use the CNTLEV=REC table option in their SAS program to access SPD Server tables. Record-level locking enforces SAS style record-level integrity across multiple clients so clients are guaranteed that an observation will not change during a multiphased Read or Write operation on the specified observation. Record-level locking will allow multiple concurrent update access to a single SPD Server table, but it will deny concurrent access to the specified observation within the table.

When an SPD Server client establishes a LIBNAME connection to a domain with record-level locking enabled, it connects using the single record-level locking proxy process. There is only one record-level locking proxy process per SPD Server. All SPD Server clients that use record-level locking connections are processed through the record-level locking proxy process. If there are a large number of record-level locking connections, there might be some contention for process resources between the clients. The record-level locking proxy process is a single point of failure for all these connections, so care should be taken when you use record-level locking to update critical data.

When you append or insert new rows into a table with defined indexes, the table updates are processed more sequentially through the record-level locking proxy process than they would be through the SPD Server user proxy processes. The performance of record-level locking will probably be less than the performance that can be obtained without record-level locking enabled for these types of operations. The standard member-level locking that is used in SPD Server user proxy processes allows for more parallel processing when doing table append or insert operations.

Record-level locking is not supported for operations on tables that use dynamic clusters.

Syntax

`LOCKING=YES|NO`

Default: NO

Arguments

YES
: enables record sharing mode.

NO
: disables record sharing mode.

Example

```
LIBNAME testrl sasspds 'tmp'
      server=serverNode.port
      user='anonymous'
      locking=YES ;
```

STARTOBS=

Summary

Specifies the start row (observation) number in a user-defined range for processing.

Syntax

STARTOBS=*n*

Arguments

n
: is the number of the start row.

Description
By default SPD Server processes the entire table unless the user specifies a range of rows with the options STARTOBS= and ENDOBS=. 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 software options FIRSTOBS= and OBS=, the STARTOBS= and ENDOBS= SPD Server options can be used for WHERE clause processing in addition to table input operations.

Example
Specify for SPD Server to process only row numbers (observations) 200–500 while the LIBNAME is active.

```
LIBNAME mydatalib sasspds 'conversion_area'
   server=husky.5105
   user='siteusr1'
   prompt=yes
   startobs=200
   endobs=500;
```

TEMP=

Summary
Controls the creation of a temporary LIBNAME domain for this LIBNAME assignment.

Syntax
`TEMP=YES|NO`

Default: NO

Arguments

YES
  creates a temporary LIBNAME domain for the LIBNAME assignment.

NO
  does not create a temporary LIBNAME domain.

Description
Use this option to create temporary LIBNAME domains that exist for the duration of the LIBNAME assignment. The TEMP (temporary) domains are analogous to SAS WORK libraries.

To create a temporary LIBNAME domain, use TEMP=YES. Any data objects, tables, catalogs, or utility files that are created in the TEMP=YES temporary domain are automatically deleted when you end the SAS session. This functions similarly to a SAS WORK library. (Note: The temporary domain is created as a subdirectory of the directory specified as the library domain.)

Example 1
Create a LIBNAME domain to use for temporary storage during your SAS session.

```
LIBNAME mydatalib sasspds 'conversion_area'
```

```
server=kaboom.5191
user='siteusr1'
prompt=yes
temp=yes ;
```

TRUNCWARN=

Summary

Suppresses hard failure on NLS transcoding overflow and character mapping errors.

Syntax

```
TRUNCWARN=YES|NO
```

Default: NO

Description

When using the TRUNCWARN=YES LIBNAME option, data integrity might be compromised because significant characters can be lost in this configuration. The default setting is NO, which causes hard Read and Write stops when transcode overflow or mapping errors are encountered. When TRUNCWARN=YES, and an overflow or character mapping error occurs, a warning is posted to the SAS log at data set close time if overflow occurs, but the data overflow is lost.

WORKPATH=

Summary

I/O contention can occur when many SPD Server users or SPD Server jobs perform heavy processing that uses the same workpath. The WORKPATH= option permits users to specify an alternate workpath that utility files (such as index builds and sorting files) can use. Specifying an alternate workpath can relieve I/O contention issues when other users are running heavy processing jobs at the same time.

A properly configured workpath directs I/O from utility operations to a separate disk. Mapping the utility file work to a separate disk using the WORKPATH= option avoids conflicts with other jobs that use a default work path that is specified in the spdsserv.parm configuration file.

Using the optional WORKPATH= specification to direct utility file operations to a separate disk increases the overall I/O throughput for the utility files and speeds up the server performance as well.

Syntax

```
WORKPATH=('path-specification') ;
```

Example

Two SPD Server power users perform heavy index creation and are creating heavy I/O contention on the default workpath that is defined in the **spdsserv.parm** configuration file:

```
workpath=('workspace1')
```

Both users override the default workpath by using the alternate WORKPATH= specification when issuing the LIBNAME statements in their jobs:

User 1 LIBNAME statement:

```
LIBNAME domain-name sasspds "domain-name"
   server=host-name.port-number
   user='user1'

workpath=('/bigdisk/spdsmgr/workpath1') ;
```

User 2 LIBNAME statement:

```
LIBNAME domain-name sasspds "domain-name"
   server=host-name.port-number
   user='user2'

workpath=('/bigdisk/spdsmgr/workpath2') ;
```

All SPD Server jobs by other users continue to use the default workpath specification that is declared in **spdsserv.parm**

The **libnames.parm** configuration file also accepts alternate WORKPATH= specifications for each domain.

Chapter 14
SPD Server Table Options

Introduction

All SAS users who use LIBNAME access to SAS Scalable Performance Data (SPD) Server should read this chapter. Most table options also work in SQL Pass-Through statements.

This chapter presents reference information for the SPD Server table options. To specify a table option with LIBNAME access, place the option value in parentheses after the table name. The option value then specifies processing that applies only to that table. To specify a table option with Pass-Through access, place the option value in brackets after the table name. The option value then specifies processing that applies only to that table. The SPD Server table options that follow are grouped by the function of their default value.

When using the options in this chapter, remember that if a table option is used subsequent to a LIBNAME option or macro variable, the value of the table option takes precedence.

Option for Compatibility with Base SAS Software

SYNCADD=

Specifies when appending to a table whether to apply a single or multiple rows at a time.

Syntax

SYNCADD=YES|NO

Default: NO

Corresponding Macro Variable

SPDSSADD

Related Table Option

UNIQUESAVE=

Arguments

YES
: imitates the behavior of the Base SAS engine, applying a single row at a time (synchronously).

NO
: appends multiple rows at a time (asynchronously).

Description

When SYNCADD= is set to YES, processing performance becomes slower. Use this setting only in order to force the server's append processing to be compatible with Base SAS software processing. That is, when the server encounters a row with a nonunique value, to cancel the append operation, back out the transactions just added, and leave the original table on disk.

Example

In this example, when executing the first INSERT statement, PROC SQL permits insertion of the values 'rollback1' and 'rollback2' because the row additions to table A are performed asynchronously. PROC SQL does not get the true completion status at the time it adds a row.

When executing the second INSERT statement, PROC SQL performs a rollback on the INSERT, upon encountering the Add error on 'nonunique', and deletes the rows 'rollback3' and 'rollback4'.

```
data a;
  input z $ 1-20 x y;
  list;

  datalines;
one                 1 10
```

```
two                 2 20
three               3 30
four                4 40
five                5 50
;

PROC SQL sortseq=ascii exec noerrorstop;
create unique index comp on a (x, y);
insert into a
  values('rollback1', -80, -80)
  values('rollback2',-90, -90)
  values('nonunique', 2, 20);

insert into a(syncadd=yes)
  set z='rollback3', x=-60, y=-60
  set z='rollback4', x=-70, y=-70
  set z='nonunique', x=2, y=20;
 quit;
```

Options That Affect Disk Space

ASYNCINDEX=

Specifies when creating multiple indexes on an SPD Server table whether to create the indexes in parallel.

Syntax

ASYNCINDEX=YES|NO

Default: NO

Corresponding Macro Variable

SPDSIASY

Arguments

YES
: creates the indexes in parallel.

NO
: creates a single index at a time.

Description

SPD Server can create multiple indexes for a table at the same time. To do this, it launches a single thread for each index created, and then processes the threads simultaneously. Although creating indexes in parallel is much faster, the default for this option is NO. The reason is because parallel creation requires additional sort work space that might not be available.

For a complete description of the benefits and tradeoffs of creating multiple indexes in parallel, see“SPDSIASY=” on page 221 in the *SAS Scalable Performance Data (SPD) Server 4.5: User's Guide*.

Example

Since the disk workspace required for parallel index creation is available, specify for SPD Server to create, in parallel, the X, Y, and COMP indexes for table A.

```
PROC DATASETS lib=mydatalib;
   modify a(asyncindex=yes);
   index create x;
   index create y;
   index create comp=(x y);
   quit;
```

COMPRESS=

Compresses SPD Server tables on disk.

Syntax

COMPRESS=YES|NO

Default: NO

Use in Conjunction with Table Option

IOBLOCKSIZE=

Corresponding Macro Variable

SPDSDCMP

Arguments

YES
: performs the run-length compression algorithm SPDSRLLC.

NO
: performs no table compression.

Description

When COMPRESS= is assigned YES, SPD Server compresses newly created tables by 'blocks' based on the algorithm specified. To control the amount of compression, use the table option IOBLOCKSIZE=. This option specifies the number of rows that you want to store in the block.

Note: Once a compressed table is created, you cannot change its block size. To resize the block, you must PROC COPY the table to a new table, setting IOBLOCKSIZE= to the block size desired for the output table.

PARTSIZE=

Specifies the size of an SPD Server table partition.

Syntax

PARTSIZE=*n*

Default: 16 Megabytes

Corresponding Macro Variable

SPDSSIZE=

Affected by LIBNAME option

DATAPATH=

Arguments

n
: is the size of the partition in megabytes.

Description

Specifying PARTSIZE= forces the software to partition (split) SPD Server tables at the given size. The actual size is computed to accommodate the largest number of rows that will fit in the specified size of n Mbytes.

Use this option to improve performance of WHERE clause evaluation on non-indexed table columns and on SQL GROUP_BY processing. By splitting the data portion of a Scalable Platform Data Server table at fixed-sized intervals, the software can introduce a high degree of scalability for these operations. The reason: it can launch threads in parallel to perform the evaluation on different partitions of the table, without the threat of file access contention between the threads. There is, however, a price for the table splits: an increased number of files, which are required to store the rows of the table.

Ultimately, scalability limits using PARTSIZE= depend on how you structure DATAPATH=, a LIBNAME option discussed in the documentation on Chapter 13, “SPD Server LIBNAME Options ,” on page 227. Specifically, the limits depend on how you configure and spread the DATAPATH= file systems across striped volumes. You should spread each individual volume's striping configuration across multiple disk controllers or SCSI channels in the disk storage array. The goal for the configuration is, at the hardware level, to maximize parallelism during data retrieval.

The PARTSIZE= specification is limited by MINPARTSIZE=, an SPD Server parameter maintained by the SPD Server administrator. MINPARTSIZE= ensures that an over-zealous SAS user does not create arbitrarily small partitions, thereby generating a large number of files. The default for MINPARTSIZE= is 16 Mbytes and probably should not be lowered much beyond this value.

Note: The PARTSIZE value for a table cannot be changed after a table is created. To change the PARTSIZE, you must PROC COPY the table and use a different PARTSIZE option setting on the new (output) table.

Example

Using PROC SQL, extract a set of rows from an existing table to create a non-indexed table with a partition size of 32 Mbytes in a SAS job:

```
PROC SQL;
create table SPDSCEN.HR80SPDS(partsize=32)
  as select
    state,
    age,
    sex,
    hour89,
    industry,
    occup
   from SPDSCEN.PRECS
   where hour89 > 40;
quit;
```

Options to Enhance Performance

BYNOEQUALS=

Specifies the output order of table rows with identical values for the BY column.

Syntax

BYNOEQUALS=YES | NO

Arguments

YES
: does not guarantee the output order of table rows with identical values in a BY clause.

NO
: guarantees the output order of table rows with identical values in a BY clause will be the relative table position of the rows from the input table. This is the default.

Example

Specify for SPD Server in the ensuing BY-column operation to output rows with identical values in the key column randomly.

```
data sport.racquets(index=(string));
   input
     raqname $20.
     @22 weight
     @28 balance $2.
     @32 flex
     @36 gripsize
     @42 string $3.
     @47 price
     @55 instock;
   datalines;
Solo Junior          10.1   N   2  3.75  syn   50.00  6
Solo Lobber          11.3   N  10  5.5   syn  160.00  1
Solo Queensize       10.9  HH   6  5.0   syn  130.00  3
Solo Kingsize        13.1  HH   5  5.6   syn  140.00  3
;

data sport.racqbal(bynoequal=yes);
  set sport.racquets;
  by balance;
run;
```

IOBLOCKSIZE=

Specifies the number of rows in a block to be stored in or read from an SPD Server table.

Syntax

IOBLOCKSIZE=*n*

Default: 4096

Use in Conjunction with Macro Variable "SPDSDCMP=" on page 220 or Table Options COMPRESS= or ENCRYPT= .

Arguments

n
  is the size of the block.

Description

The software reads and stores a server table in blocks. IOBLOCKSIZE= is useful on compressed or encrypted tables. SPD Server software does not use IOBLOCKSIZE= on noncompressed or nonencrypted tables.

For tables that you compress or encrypt, using either the option COMPRESS= or the macro variable SPDSDCMP=, the IOBLOCKSIZE= specification determines the number of rows to include in the block. The specification applies to block compression as well as data I/O to and from disk. The IOBLOCKSIZE= value affects the table's organization on disk.

When using SPD Server table compression or encryption, specify an IOBLOCKSIZE= value that complements how the data is to be accessed, sequentially or randomly. Sequential access or operations requiring full table scans favor a large block size, for example 64K. In contrast, random access favors a smaller block size, for example 8K.

Example

A huge company mailing list is processed sequentially. Specify a block size for compression that is optimal for sequential access.

```
/* IOblocksize set to 64K */
data sport.maillist(ioblocksize=65536 compress=yes);
   input name $ 1-20
     address $ 21-57
     phoneno $ 58-69
     sex $71;

  datalines;

Douglas, Mike       3256 Main St., Cary, NC 27511        919-444-5555 M
Walters, Ann Marie  256 Evans Dr., Durham, NC 27707      919-324-6786 F
Turner, Julia       709 Cedar Rd., Cary, NC 27513        919-555-9045 F
Cashwell, Jack      567 Scott Ln., Chapel Hill, NC 27514 919-533-3845 M
Clark, John         9 Church St.,  Durham, NC 27705      919-324-0390 M
;
run;
```

NETPACKSIZE=

Specifies the size of the SPD Server network data packet.

Syntax

NETPACKSIZE=*size-of-packet*

Arguments

size-of-packet
  is the size of the network packet in bytes.

Description

This option controls the size of the buffer used for data transfer between SPD Server and a SAS client. The default is 32K bytes. The buffer size is relative to the size of a table row. It cannot be less than the size of a single row. Packet size must be equal to some multiple of the table rows. If it is not, SPD Server rounds up the size specified. For example, if the packet buffer size is 4096 bytes and the row size is 3072, the software rounds up the buffer size to 6144.

Select a packet size to complement the bandwidth of the network it must travel through. An optimum size will flow the data continuously without significant pauses between packets.

Example

Create a 12K buffer in the memory of the server to send three rows from MYTABLE in each network packet. (The row size in MYTABLE is 4K.)

```
data mylib.mytable (netpacksize=12288);
```

SEGSIZE=

Specifies the size of the segment for an index file associated with an SPD Server table.

Syntax

SEGSIZE=*number*

Arguments

number
is the number of table rows to include in the index segment.

Description

The minimum SEGSIZE= value is 1024 table rows. The default value is 8192 table rows. The size of the index segment corresponds to the structure of the table and cannot be changed after the table is created.

Example

Specify a segment size of 64K for MYLIB.MYTABLE.

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

Note: Tests show that increasing the size of the segment does not significantly increase performance.

Option to Test Performance

NOINDEX=

Specifies whether to use the table's indexes when processing WHERE clauses.

Syntax

NOINDEX=YES|NO

Default: NO

Arguments

YES
ignores indexes when processing WHERE clauses.

NO
uses indexes when processing WHERE clauses.

Description

Set NOINDEX= to YES to test the effect of indexes on performance or for specific processing. Do not use YES routinely for normal processing.

Example

We created an index for the SEX column but decide to test whether it is necessary for our PROC PRINT processing. Specify for the server not to use the index.

```
data sport.maillist;
   input
     name $ 1-20
     address $ 21-57
     phoneno $ 58-69
     sex $71;

  datalines;

Douglas, Mike       3256 Main St., Cary, NC 27511        919-444-5555 M
Walters, Ann Marie  256 Evans Dr., Durham, NC 27707      919-324-6786 F
Turner, Julia       709 Cedar Rd., Cary, NC 27513        919-555-9045 F
Cashwell, Jack      567 Scott Ln., Chapel Hill, NC 27514 919-533-3845 M
Clark, John         9 Church St.,  Durham, NC 27705      919-324-0390 M
;

PROC DATASETS lib=sport nolist;
  modify maillist;
  index create sex;
  quit;

/*Turn on the macro variable SPDSWDEB */
/* to show that the index is not used */
/* used during the table processing. */

%let spdswdeb=YES;

title All Females from Current Mailing List;
PROC PRINT data=sport.maillist(noindex=yes);
where sex=F;
run;
```

Options for WHERE Clause Evaluations

MINMAXVARLIST=

Creates a list that documents the minimum and maximum values of specified variables. SPD Server WHERE clause evaluations use MINMAXVARLIST= lists to include or eliminate member tables in an SPD Server dynamic cluster table from SQL evaluation scans..

Syntax

MINMAXVARLIST=(varname1 varname2 ... varnameN)

Arguments

varname1 varname2 ... varname N
: are SPD Server table variable names.

Description

The primary purpose of the MIINMAXVARLIST= table option is for use with SPD Server Chapter 7, "SAS Scalable Performance Data (SPD) Server Dynamic Cluster Tables," on page 67 where specific members in the dynamic cluster contain a set or range of values, such as sales data for a given month. When an SPD Server SQL subsetting WHERE clause specifies specific months from a range of sales data, the WHERE planner checks the MIN and MAX list. Based on the MIN and MAX list information, the SPD Server WHERE planner includes or eliminates member tables in the dynamic cluster for evaluation.

MINMAXVARLIST= uses the list of columns you submit to build the list. The MINMAXVARLIST= list contains only the minimum and maximum values for each column. The WHERE clause planner uses the index to filter SQL predicates quickly, and to include or eliminate member tables belonging to the cluster table from the evaluation.

Although the MINMAXVARLIST= table option is primarily intended for use with dynamic clusters, it also works on standard SPD Server tables. MINMAXVARLIST= can help reduce the need to create many indexes on a table, which can save valuable resources and space.

Example

```
%let domain=path3 ;
%let host=kaboom ;
%let port=5201 ;

LIBNAME &domain sasspds "&domain"
   server=&host..&port
   user='anonymous' ;

/* Create three tables called */
/* xy1, xy2, and xy3.         */

data &domain..xy1(minmaxvarlist=(x y));
  do x = 1 to 10;
  do y = 1 to 3;
  output;
  end;
end;
run;

data &domain..xy2(minmaxvarlist=(x y));
  do x = 11 to 20;
  do y = 4 to 6 ;
  output;
  end;
end;
run;
```

```
data &domain..xy3(minmaxvarlist=(x y));
  do x = 21 to 30;
  do y = 7 to 9 ;
  output;
  end;
end;
run;


/* Create a dynamic cluster table */
/* called cluster_table out of    */
/* new tables xy1, xy2, and xy3   */

PROC SPDO library=&domain ;
   cluster create cluster_table
      mem=xy1
      mem=xy2
      mem=xy3
      maxslot=10;
quit;


/* Enable WHERE evaluation to see  */
/* how the SQL planner selects     */
/* members from the cluster. Each  */
/* member is evaluated using the   */
/* min-max list.                   */

%let SPDSWDEB=YES;


/* The first member has true rows  */

PROC PRINT data=&domain..cluster_table ;
   where x eq 3
   and y eq 3;
run;


/* Examine the other tables */

PROC PRINT data=&domain..cluster_table ;
   where x eq 3
   and y eq 3 ;
run;

PROC PRINT data=&domain..cluster_table ;
   where x eq 3
   and y eq 3;
run;

PROC PRINT data=&domain..cluster_table ;
   where x between 1 and 10
   and y eq 3;
run;
```

```
PROC PRINT data=&domain..cluster_table ;
   where x between 11 and 30
   and y eq 8 ;
run;


/* Delete the dynamic cluster table. */

PROC DATASETS lib=&domain nolist;
   delete cluster_table ;
quit ;
```

THREADNUM=

Specifies the number of threads to be used for WHERE clause evaluations.

Syntax

THREADNUM=n

Default: THREADNUM= is set equal to the value of the MAXWHTHREADS server parameter.

Used in Conjunction with SPD Server Parameter

MAXWHTHREADS

Corresponding Macro Variable

SPDSTCNT=

Arguments

n
: is the number of threads.

Description

THREADNUM= allows you to specify the thread count the SPD Server should use when performing a parallel WHERE clause evaluation.

Use this option to explore scalability for WHERE clause and GROUP_BY evaluations in non-production jobs. If you use this option for production jobs, you are likely to lower the level of parallelism that is applied to those clause evaluations.

THREADNUM= works in conjunction with MAXWHTHREADS, a configurable system parameter. MAXWHTHREADS imposes an upper limit on the consumption of system resources. The default value of MAXWHTHREADS is dependent on your operating system. Your SPD Server administrator can change the default value for MAXWHTHREADS.

If you do not use THREADNUM=, the software provides a default thread number, up to the value of MAXWHTHREADS as required. If you use THREADNUM=, the value that you specify is also constrained by the MAXWHTHREADS value.

The THREADNUM= value applies both to parallel table scans (EVAL2 strategy), parallel indexed evaluations (EVAL1 strategy), parallel BY-clause processing, and parallel GROUP_BY evaluations. See the *SAS Scalable Performance Data (SPD) Server 4.5: User's Guide* for more information about "Optimizing WHERE clauses" on page 183.

Example

The SPD Server administrator set MAXWHTHREADS=128 in the SAS Scalable Performance Data (SPD) Server's parameter file. Explore the effects of parallelism on a given query by using the following SAS macro:

```
%macro dotest(maxthr);
%do nthr=1 %to &maxthr
   data _null_;
     set SPDSCEN.PRECS(threadnum=&nthr);
     WHERE
       occup='022'
       and state in('37','03','06','36');
   run;
%mend dotest;
```

WHERENOINDEX=

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

Syntax

WHERENOINDEX=(name1 name2...)

Arguments

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

Example

We have a table PRECS with indexes defined as follows:

```
PROC DATASETS lib=spdscen;
modify precs(bitindex=(hour89));
index create
  stser=(state serialno)
  occind=(occup industry)
  hour89;
quit;
```

When evaluating the next query, we want the SPD Server to exclude from consideration indexes for both the STATE and HOUR89 columns.

In this case, we know that the AND combination of the predicates for the OCCUP and INDUSTRY columns will produce a very small yield. Few rows satisfy the respective predicates. 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 table hr80spds
  as select
    state,
    age,
    sex,
    hour89,
    industry,
    occup
  from spdscen.precs(wherenoindex=(stser hour89))
```

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

Note: Specify index names in the WHERENOINDEX list, not the column names. The example excludes both the composite index for the STATE column STSER and the simple index HOUR89 from consideration by the WHINIT WHERE planner.

Options for Other Functions

BYSORT=

Perform an implicit automatic sort when SPD Server encounters a BY clause for a given table.

Syntax

```
BYSORT=YES | NO
```

Arguments

YES
: sorts the data based on the BY columns and returns the sorted data to the SAS client. This powerful capability means that the user does not have to sort data using a PROC SORT statement before using a BY clause.

NO
: does not sort the data based on the BY columns. This might be desirable if a DATA step BY clause has a GROUPFORMAT option or if a PROC step reports grouped and formatted data.

Description

The default is YES. The NO argument means the table must have been previously sorted by the requested BY columns. The NO argument allows grouped data to maintain their precise order in the table. A YES argument groups the data correctly but possibly in a different order from the order in the table.

Example 1 - Group Formatting with BYSORT=

```
LIBNAME sport sasspds 'mylib'
   host='samson'
   user='user19'
   passwd='dummy2';

PROC FORMAT;
   value dollars
     0-99.99="low"
     100-199.99="medium"
     200-1000="high";
run;

data sport.racquets;
   input
```

```
     raqname $20.
     @22 weight
     @28 balance $2.
     @32 flex
     @36 gripsize
     @42 string $3.
     @47 price
     @55 instock;

   datalines;
Solo Junior          10.1   N   2  3.75  syn   50.00   6
Solo Lobber          11.3   N  10  5.5   syn  160.00   1
Solo Queensize       10.9  HH   6  5.0   syn  130.00   3
Solo Kingsize        13.1  HH   5  5.6   syn  140.00   3
;

PROC PRINT data=sport.racquets (bysort=yes);
   var raqname instock;
   by price;
   format price dollars.;
title 'Solo Brand Racquets by Price Level';
run;
```

Output 14.1 *Report Output with BYSORT=*

```
                Solo Brand Racquets by Price Level

--------------------------- Price=low --------------------------

OBS                          RAQNAME                       INSTOCK

 1                         Solo Junior                        6

------------------------- Price=medium -------------------------

OBS                          RAQNAME                       INSTOCK

 3                        Solo Queensize                      3

 4                        Solo Kingsize                       3

 2                         Solo Lobber                        1
```

Example 2 - Group Formatting without BYSORT=

```
PROC PRINT data=sport.racquets (bysort=no);
   var raqname instock;
   by price;
   format price dollars.;
title 'Solo Brand Racquets by Price Level';
run;
```

Output 14.2 *Report Output without BYSORT=*

```
                Solo Brand Racquets by Price Level

--------------------------- Price=low --------------------------

OBS                          RAQNAME                       INSTOCK

 1                         Solo Junior                        6

------------------------- Price=medium -------------------------

OBS                          RAQNAME                       INSTOCK

 2                         Solo Lobber                        1

 3                        Solo Queensize                      3

 4                        Solo Kingsize                       3
```

ENDOBS=

Specifies the end row (observation) number in a user-defined range for the processing of a given table.

Syntax

ENDOBS=n

Arguments

n
   is the number of the end row.

Description

By default, SPD Server processes the entire table unless the user specifies a range of rows with the STARTOBS= and ENDOBS= options. If the STARTOBS= option is used without the ENDOBS= option, the implied value of ENDOBS= is the end of the table. When both options are used together, the value of ENDOBS= must be greater than STARTOBS=.

In contrast to the Base SAS software options FIRSTOBS= and OBS=, the STARTOBS= and ENDOBS= SPD Server options can be used for WHERE clause processing in addition to table input operations.

Example

Print only rows 2-4 of the table INVENTORY.OLD_AUTOS.

```
LIBNAME inventory sasspds 'conversion_area'
   server=husky.5105
   user='siteusr1'
   prompt=yes;

data inventory.old_autos;
   input
     year $4.
     @6 manufacturer $12.
     model $10.
     body_style $5.
     engine_liters
     @39 transmission_type $1.
     @41 exterior_color $10.
     options $10.
     mileage conditon;

   datalines;
1971 Buick       Skylark   conv  5.8  A  yellow    00000001 143000 2
1982 Ford        Fiesta    hatch 1.2  M  silver    00000001  70000 3
1975 Lancia      Beta      2door 1.8  M  dk blue   00000010  80000 4
1966 Oldsmobile  Toronado  2door 7.0  A  black     11000010 110000 3
1969 Ford        Mustang   sptrf 7.1  M  red       00000111 125000 3
;

PROC PRINT data=inventory.old_autos (startobs=2 endobs=4);
run;
```

Output 14.3 *Data in the Printed Output*

```
1982   Ford         Fiesta        hatch   1.2  M   silver    00000001   70000
3

1975   Lancia       Beta          2door   1.3  M   dk blue   00000010   80000
4

1966   Oldsmobile   Toronado      2door   7.0  A   black     11000010  110000
3
```

STARTOBS=

Specifies the start row (observation) number in a user-defined range for the processing of a given table.

Syntax

STARTOBS=*n*

Arguments

n
   is the number of the start row.

Description

By default, SPD Server processes the entire table unless the user specifies a range of rows 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 ENDOBS=.

In contrast to the Base SAS software options FIRSTOBS= and OBS=, the STARTOBS= and ENDOBS= SPD Server options can be used for WHERE clause processing in addition to table input operations.

Example

Print only rows 2-4 of the table INVENTORY.OLD_AUTOS.

```
LIBNAME inventory sasspds 'conversion_area'
   server=husky.5105
   user='siteusr1'
   prompt=yes;

data inventory.old_autos;
   input
     year $4.
     @6 manufacturer $12.
     model $10.
     body_style $5.
     engine_liters
     @39 transmission_type $1.
     @41 exterior_color $10.
     options $10.
     mileage conditon;

   datalines;
1971 Buick       Skylark   conv  5.8  A  yellow    00000001 143000 2
1982 Ford        Fiesta    hatch 1.2  M  silver    00000001  70000 3
1975 Lancia      Beta      2door 1.8  M  dk blue   00000010  80000 4
1966 Oldsmobile  Toronado  2door 7.0  A  black     11000010 110000 3
1969 Ford        Mustang   sptrf 7.1  M  red       00000111 125000 3
;

proc print data=inventory.old_autos (startobs=2 endobs=4);
run;
```

UNIQUESAVE=

Specifies to save rows with nonunique key values (the rejected rows) to a separate table when appending data to tables with unique indexes.

Syntax

UNIQUESAVE=YES|NO|REP

Default: NO

Complements the Table Option

SYNCADD=

Used in Conjunction with Macro Variable

SPDSUSDS=

Corresponding Macro Variable:

SPDSUSAV=

Arguments

YES
: writes rejected rows to a separate, system-created table file which can be accessed by a reference to the macro variable SPDSUSDS=.

NO
: does not write rejected rows to a separate table, that is, ignores nonunique key values.

REP
: when updating a master table from a transaction table, where the two tables share identical variable structures, the UNIQUESAVE=REP option replaces the row updated

row in the master table instead of appending a row to the master table. The REP option only functions in the presence of a /UNIQUE index on the MASTER table. Otherwise the REP setting is ignored..

Description

SYNCADD= is defaulted to NO. When NO, table appends are 'pipelined', meaning that the server data is sent in a stream a block at a time. (See table option NETPACKSIZE=.) While pipelining is faster than a synchronous append, SAS reports the results of the append operation differently for these two modes.

When applying only a single row (SYNCADD=NO), SAS returns a status code for each ADD operation. The application can determine the next action based on the status value. If a row is rejected due to containing a nonunique value for a unique index, the user receives a status message. In contrast, when data is pipelined (SYNCADD=YES), SAS returns a status code only after **all** the rows are applied to a table. As a consequence, the user does not know which rows have been rejected.

To enjoy the performance of data pipelining but still retain the rejected rows, use the UNIQUESAVE= option. When set to YES, SPD Server will save any rows that are rejected to a hidden SAS table.

When using this option, SAS returns the name of the hidden table containing the rejected rows in the macro variable SPDSUSDS. If you want to report the contents of the table, reference “SPDSUSDS=” on page 209 .

Note: If SYNCADD= YES is set, data pipelining is overridden and the data is processed synchronously. In this situation, the UNIQUESAVE= option is not relevant and, if set, is ignored.

Example 1

We want to append two tables, NAMES2 and NAMES3, which contain employees' names, to the NAMES1 table. Before performing our append, we create an index on the NAME column in NAMES1, declaring the index unique.

Specify for SPD Server, during the append operation, to store rows found with duplicate employee names to a separate table file generated by the macro variable SPDSUSDS=.

Use a %PUT statement to display the table name for SPDSUSDS=. Then request a printout of the duplicate rows to review later.

```
data employee.names1;
input name $ exten;
datalines;
Jill 4344
Jack 5589
Jim  8888
Sam  3334
;
run;

data employee.names2;
input name $ exten;
datalines;
Jack  4443
Ann   8438
Sam   3334
Susan 5321
Donna 3332
;
```

```
run;

data employee.names3;
input name $ exten;
datalines;
Donna 3332
Jerry 3268
Mike  2213
;
run;

PROC DATASETS lib=employee nolist;
  modify names1;
  index create name/unique;
quit;

PROC APPEND data=employee.names2
            out=employee.names1(uniquesave=yes); run;

title 'The NAMES1 table with unique names
       from NAMES2';

PROC PRINT data=employee.names1;
run;

%put Set the macro variable spdsusds to &spdsusds;

title 'Duplicate (nonunique) name rows found in
       NAMES2';

PROC PRINT data=&spdsusds;
run;

PROC APPEND data=employee.names3
   out=employee.names1(uniquesave=yes);
run;
```

The SAS log provides the messages:

```
WARNING: Duplicate values not allowed on index NAME for
         file EMPLOYEE.NAMES1. (Occurred 2 times.)
NOTE: Duplicate records have been stored in file
      EMPLOYEE._30E3FD5.
```

And, an extract from our PROC PRINT shows:

```
The NAMES1 table with unique names from NAMES2

         OBS    NAME     EXTENs

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

```
Duplicate (nonunique) name rows found in NAMES2

        OBS    NAME     EXTEN    XXX00000

          1     Jack      4443      NAME
          2     Sam       3334      NAME
```

Example 2

Use the UNIQUESAVE=REP option to perform an update / append case using PROC APPEND instead of a DATA step:

```
* A MASTER table to update. ID */
/* will get a UNIQUE index      */

  DATA SPDS.MASTER;
    INPUT ID VALUE $;
    CARDS;
     1 one
     2 two
     3 three
  ;

  PROC DATASETS LIB=SPDS;
    MODIFY MASTER;
    INDEX CREATE ID/UNIQUE;
  QUIT;

  /* A transaction table TRANS to use to   */
  /* drive update/appends to MASTER        */

  DATA SPDS.TRANS;
    INPUT ID VALUE $;
     1 ONE
     3 THREE
     4 FOUR
     4 FOUR*
  ;

  /* Use of UNIQUESAVE=REP to update/append  */
  /* TRANS rows to MASTER based on whether   */
  /* TRANS records have an ID column that    */
  /* matches an existing row from the MASTER */
  /* table. Update MASTER rows with a match, */
  /* otherwise append TRANS row to MASTER    */

  PROC APPEND DATA=SPDS.TRANS
    OUT=SPDS.MASTER(UNIQUESAVE=REP);
  run;
```

Output of the resulting MASTER table would look like:

```
  Obs       ID      VALUE
```

```
    1         1      ONE
    2         2      two
    3         2      THREE
    4         4      FOUR*
```

VERBOSE=

Provides details of all indexes and ACL information associated with an SPD Server table.

Syntax

VERBOSE= YES | NO

Arguments

YES
: requests detail information for the indexes, ACLs, and other SPD Server table values. This argument must be used with the CONTENTS procedure.

NO
: suppresses detail information for the indexes, ACLs, and other SPD Server table values. This is the default.

Example

Request details of all the indexes for the table TEMP1 in the domain SPDS45.

```
PROC CONTENTS data=SPDS45 (verbose=yes);
run;
```

Output 14.4 *Output 14.4: Details of Table TEMP1 Indexes in Domain SPDS45*

```
                          The CONTENTS Procedure

Data Set Name           SPDS45.TEMP1                          Observations           1000
Member Type             DATA                                  Variables              2
Engine                  SASSPDS                               Indexes                2
Created                 Tuesday, May 10, 2005 10:00:02 AM     Observation Length     16
Last Modified           Tuesday, May 10, 2005 11:01:36 AM     Deleted Observations   0
Protection                                                    Compressed             NO
Data Set Type                                                 Sorted                 NO
Label
Data Representation     Default
Encoding                Default


                        Engine / Host Dependent Information

                        Blocking Factor (obs/block)     2047
                        ACL Entry                       NO
                        ACL User Access(R,W,A,C)        (Y,Y,Y,Y)
                        ACL User Name                   ANONYMOU
                        ACL Owner Name                  ANONYMOU
                        Data Set is Ranged              NO
                        Alphabetic List of Index Info   .
                        Bitmap index (No Global Index ) i
                        Keyvalue (Min)                  1
                        Keyvalue (Max)                  100
                        # of Discrete values            100
                        Bitmap index (No Global Index ) j
                        Keyvalue (Min)                  1
                        Keyvalue (Max)                  10
                        # of Discrete values            10
                        Data Partsize                   16777216


                        Alphabetic List of Variables and Attributes

                        *       Variable        Type    Len

                        1       i               Num     8
                        2       j               Num     9


                        Alphabetic List of Indexes and Attributes

                                                # of
                                                Unique
                        *       Index           Values

                        1       i               100
                        2       j               10
```

Options for Security

ENCRYPT=

Encrypts SPD Server tables on disk. Encryption is a security mechanism that protects table contents from users who have system access to raw SPD Server tables. Access to tables is normally controlled by SPD Server ACLs. The S*AS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide* contains detailed information about using SPD Server ACLs to control access to tables.

When the ENCRYPT= option setting is set to YES, SPD Server encrypts newly created tables by blocks. To control the amount of encryption per block, use the table option IOBLOCKSIZE=. The IOBLOCKSIZE= option specifies the number of rows to be encrypted in each block.

Syntax

ENCRYPT= YES | NO

Arguments

YES
: encrypts the data set. The encryption method uses passwords. At a minimum, you must specify the READ= or the PW= data set option at the same time that you specify an ENCRYPT=YES option setting.

NO
: no table encryption is performed. NO is the default setting for the ENCRYPT= option.

Usage Notes

1. Depending on your query patterns, increasing or decreasing the block size can affect performance.
2. SPD Server does not encrypt table indexes or metadata. Only table row data are encrypted.
3. To encrypt SPD tables with Pass-Through SQL, use only the READ= or PW= table option. With Pass-Through SQL, ENCRYPT=YES is implied with these options.
4. To access an encrypted table, the user must have appropriate ACL permissions to the table and must provide the encryption key via the READ= or PW= table option.
5. Encrypting an SPD Server table provides security from users that have system access to dump raw SPD Server tables. The section on Security in the *SAS Scalable Performance Data (SPD) Server 4.5: Administrator's Guide* contains more information about how to control system access to SPD Server tables.

Chapter 15
SPD Server Formats and Informats

Introduction

SAS Scalable Performance Data (SPD) Server supports some of the more commonly used SAS format and informats. Use these in your SQL Pass-Through code when you want SAS Scalable Performance Data (SPD) Server to associate a data set variable with a specific format.

A general reminder about formats: A format is applied to data set variables as it is written out. Informats are applied as the data set variable is being read.

Formats

List of Formats

- **$** — Writes standard character data
- **$BINARY** — Converts character values to binary representation
- **$CHAR** — Writes standard character data
- **$HEX** — Converts character values to hexadecimal representation
- **$OCTAL** — Converts character values to octal representation
- **$QUOTE** — Converts character values to quoted strings
- **$VARYING** — Writes varying length values
- **BEST** — SAS Scalable Performance Data (SPD) Server system chooses best notation
- **BINARY** — Converts numeric values to binary representation

- **COMMA** — Writes numeric values with commas and decimal points
- **COMMAX** — Writes numeric values with commas and decimal points (European style)
- **DATE** — Writes date values (ddmmmyy)
- **DATETIME** — Writes date time values (ddmmmyy:hh:mm:ss.ss)
- **DAY** — Writes day of month
- **DDMMYY** — Writes date values (ddmmyy)
- **DOLLAR** — Writes numeric values with dollar signs, commas, and decimal points
- **DOLLARX** — Writes numeric values with dollar signs, commas, and decimal points (European style)
- **DOWNAME** — Writes name of day of the week
- **E** — Writes scientific notation
- **F** — Writes scientific notation
- **FRACT** — Converts values to fractions
- **HEX** -- Converts real binary (floating-point) numbers to hexadecimal representation
- **HHMM** — Writes hours and minutes
- **HOUR** — Writes hours and decimal fractions of hours
- **IB** — Writes integer binary values
- **MMDDYY** — Writes date values (mmddyy)
- **MMSS** — Writes minutes and seconds
- **MMYY** — Writes month and year, separated by a 'M'
- **MONNAME** — Writes name of month
- **MONTH** — Writes month of year
- **MONYY** — Writes month and year
- **NEGPAREN** — Displays negative values in parentheses
- **OCTAL** — Converts numeric values to octal representation
- **PD** — Writes packed decimal data
- **PERCENT** — Prints numbers as percentages
- **PIB** — Writes positive integer binary values
- **QTR** — Writes quarter of year
- **RB** — Writes real binary (floating-point) data
- **SSN** — Writes Social Security numbers
- **TIME** — Writes hours, minutes, and seconds
- **TOD** — Writes the time portion of datetime values
- **w.d** — Writes standard numeric data
- **WEEKDATE** — Writes day of week and date (day-of-week, month-name dd, yy)
- **WEEKDATX** — Writes day of week and date (day-of-week, dd month-name yy)
- **WEEKDAY** — Writes day of week

- **WORDDATE** — Writes date with name of month, day, and year (month-name dd, yyyy)
- **WORDDATX** — Writes date with day, name of month, and year (dd month-name yyyy)
- **WORDF** — Converts numeric values to words
- **WORDS** — Converts numeric values to words (fractions as words)
- **YEAR** — Writes year part of date value
- **YYMM** — Write year and month, separated by a 'M'
- **YYMMDD** — Writes day values (yymmdd)
- **YYMON** — Writes year and month abbreviation
- **YYQ** — Writes year and quarter, separated by a 'Q'
- **Z** — Writes leading 0s
- **ZD** — Writes data in zoned decimal format

Note: Formats which begin with a '$' sign are character formats. Otherwise the format accepts numeric values.

Formats Example

Use the dollar. format to convert numeric sales figures into dollar values. Suppose you have an SPD Server data set **Sales** with a single numeric variable **salesite** representing the total sales for a given site. Using SQL Pass-Through, create a new data set containing the sales in dollar format.

```
PROC SQL;
connect to sasspds
  (dbq='tmp'
   user='anonymous'
   host='localhost'
   serv='5127');

execute(create table money
   as select salesite
   format=dollar.
   from sales)

by sasspds;

quit;
```

User-Defined Formats Example

This example is a sample test job that validates its own configuration to use user-defined formats. When properly configured, user-defined formats will allow columns to be formatted using parallel GROUP BY statements and a WHERE clause that uses a format to subset data to the server.

The example provides sample **spdsserv.parm** and **libnames.parm** file examples, as well as code examples that follow the two sample SPD Server configuration files.

This example is a sample test job that checks the usage of user-defined formats. When correctly set up, user-defined formats will allow formatting of columns in parallel GROUP BY and permits usage of a WHERE clause that uses a format to subset data.

SPD Server **spdsserv.parm** file used in the example:

```
SORTSIZE=8M;
INDEX_SORTSIZE=8M;
BINBUFSIZE=32K;
INDEX_MAXMEMORY=8M;
NOCOREFILE;
SEQIOBUFMIN=64K;
RANIOBUFMIN=4K;
NOALLOWMMAP;
MAXWHTHREADS=16;
WHERECOSTING;
RANDOMPLACEDPF;
FMTDOMAIN=FORMATS;
FMTNAMENODE=d8488 ;
FMTNAMEPORT=5200;
```

SPD Server **libnames.parm** file used in the example:

```
LIBNAME=tmp pathname=c:\temp;
LIBNAME=formats pathname=c:\data\formats;
```

SPD Server example code:

```
%let domain=tmp;
%let host=d8488;
%let serv=5200;

/* locking=YES must be specified when using */
/* options fmtsearch=(formats); */

LIBNAME formats sasspds 'formats'
 host=&host; serv=&serv;
 user='anonymous' locking=YES;
 LIBNAME &domain;
 sasspds &domain;
 host=&host;
 serv=&serv;
 user='anonymous'
 IP=YES;

 options fmtsearch=(formats);

PROC DATASETS nolist
 lib=formats
 memtype=catalog;
 delete formats;
quit ;
```

```
/* To create user defined formats, they must be */
/* loaded from the same platform where they are */
/* going to be stored. You cannot use Windows */
/* path specifications to load formats on UNIX */
/* platforms. */

/* Add formats to format domain */

PROC FORMAT lib=formats;
value AGEGRP
 0-13='Child'
 14-17='Adolescent'
 18-64='Adult'
 65-HIGH='Pensioner';
value $GENDER
 'F' = 'Female'
 'M' = 'Male';
run ;

/* Create a test table with a column that uses */
/* AGEGRP format */

data &domain..fmttest;
format age AGEGRP. GENDER $GENDER. id z5.;
 length GENDER $1;
do id=1 to 100;
 if mod (id,2) = 0
   then GENDER = 'F';
   else GENDER = 'M';
 age=int(ranuni(0)*100);
 income=age*int(ranuni(0)*1000);
output;

end;

run;

/* Use the parallel GROUP BY feature with the */
/* fmtgrpsel option. This groups the data based */
/* on the output format specified in the table. */
/* This will be executed in parallel. */

PROC SQL;
connect to sasspds
(dbq="&domain";
 serv="&serv";
 host="&host";
 user="anonymous");

/* Explicitly set the fmtgrpsel option */

execute(reset fmtgrpsel)
by sasspds;

title 'Simple Fmtgrpsel Example';
```

```
select *
from connection to sasspds
 (select age, count(*) as count
 from fmttest group by age);

disconnect from sasspds;

quit;

PROC SQL;
connect to sasspds
(dbq="&domain";
 serv="&serv";
 host="&host";
 user="anonymous");

/* Explicitly set the fmtgrpsel option */


execute(reset fmtgrpsel)
by sasspds;

title 'Format Both Columns Group Select Example';

select *
from connection to sasspds
 (select GENDER format=$GENDER.,
            AGE format=AGEGRP.,
  count(*) as count
  from fmttest
  formatted group by GENDER, AGE);

disconnect from sasspds;

quit;

PROC SQL;
connect to sasspds
(dbq="&domain";
 serv="&serv";
 host="&host";
 user="anonymous");

/* Explicitly set the fmtgrpsel option */

execute(reset fmtgrpsel)
by sasspds;

title1 'To use Format on Only One Column With Group Select';
title2 'Override Column Format With a Starndard Format';

select *
from connection to sasspds
 (select GENDER format=$1.,
         AGE format=AGEGRP.,
  count(*) as count
  from fmttest
  formatted group by GENDER, AGE);

disconnect from sasspds;

quit;

/* A WHERE clause that uses a format to subset */
/* data is pushed to the server. If it is not */
/* pushed to the server, the following warning */
/* message will be written to the SAS log: */
/* WARNING: Server is unable to execute the */
/* where clause. */

data temp;
set &domain..fmttest;
 where put
 (AGE,AGEGRP.) = 'Child';
 run;
```

Informats

- **$** — Reads standard character data
- **$BINARY** — Converts binary values to character values
- **$CB** — Reads standard character data from column-binary files
- **$CHAR** — Reads character data with blanks
- **$HEX** — Converts hexadecimal data to character data
- **$OCTAL** — Converts octal data to character data
- **$PHEX** — Converts packed hexadecimal data to character data
- **$QUOTE** — Converts quoted strings to character data
- **$SASNAME** —
- **$VARYING** — Reads varying length values
- **BEST** — SPD Server system chooses best notation
- **BINARY** — Converts positive binary values to integers
- **BITS** — Extract bits
- **COMMA** — Removes embedded characters (for example $,.)
- **COMMAX** — Removes embedded characters (for example $,.) European style
- **D** — Reads scientific notation
- **DATE** — Reads date values (ddmmmyy)
- **DATETIME** — Reads datetime values (ddmmmyy hh:mm:ss.ss)
- **DDMMYY** — Reads date values (ddmmyy)
- **DOLLAR** — Reads numeric values with dollar signs, commas, and decimal points

- **DOLLARX** — Reads numeric values with dollar signs, commas, and decimal points (European style)
- **E** — Reads scientific notation
- **F** — Reads scientific notation
- **HEX** — Converts hexadecimal positive binary values to fixed- or floating-point values
- **IB** — Reads integer binary (fixed-point) values
- **JULIAN** — Reads Julian dates (yyddd or yyyyddd)
- **MMDDYY** — Reads date values (mmddyy)
- **MONYY** — Reads month and year date values (mmmyy)
- **MSEC** — Reads TIME MIC values
- **OCTAL** — Converts octal values to integers
- **PD** — Reads packed decimal data
- **PDTIME** — Reads packed decimal time of SMF and RMF records
- **PERCENT** — Converts percentages into numeric values
- **PIB** — Reads positive integer binary (fixed-point) values
- **PK** — Reads unsigned packed decimal data
- **PUNCH** — Reads whether a row of column-binary data is punched
- **RMFSTAMP** — Reads time and date fields of RMF records
- **ROW** — Reads a column-binary field down a card column
- **SMFSTAMP** — Reads time-date values of SMF records
- **TIME** — Reads hours, minutes, and seconds (hh:mm:ss.ss)
- **TODSTAMP** — Reads 8-byte time-of-day stamp
- **TU** — Reads timer units
- **YYMMDD** — Reads date values (yymmdd)
- **YYQ** — Reads quarters of the year

Note: Informats that begin with a $ sign are character informats. Otherwise the informat accepts numeric values.

The SQL procedure itself does not use the INFORMAT= modifier: it stores informats in its table definitions so that other procedures and the DATA step can use the information. SPD Server informats are provided now to allow for forward compatibility with future development.

Chapter 16
SPD Server NLS Support

Overview of NLS

NLS, or National Language Support, deals both with Internationalization and Localization of SAS software. Internationalization is the process of designing an application so that it can be adapted to different languages and regions, without requiring engineering changes. Often the term internationalization is abbreviated as **i18n**, because there are 18 letters between the first i and the last n. Localization is the process of adapting software for a particular region or language by adding locale-specific components and translating text. The term localization is often abbreviated as **L10n**, because there are 10 letters between the L and the n. Translation of user interface, messages, and documentation is a large part (but not all) of localization. Localizers also verify that the formatting of dates, numbers, currencies, and so on, conforms to local requirements.

SAS 9 contains built-in support for NLS character set encoding and locale choices. Users access the NLS encoding and locale choices through various SAS, LIBNAME, and data set options. SAS Scalable Performance Data (SPD) Server and SAS together offer basic levels of NLS support. This document describes the basic entities of NLS support and how they are implemented in SPD Server

Character Encoding

Overview of Character Encoding

All input to a computer is represented internally as numbers. The computer assigns a number to each character – technically, the number is a binary number (base 2 numbering system, consisting of 0s and 1s).

Because most of us do not think in binary numbers, computers provide hexadecimal (base 16 numbering system) representation as a shorthand for binary representation. For example, for the decimal number 167, it's easier to understand the hexadecimal number A7 than the equivalent binary number 10100111. Therefore, you can think of the computer's internal numeric representation of all data as a hexadecimal number.

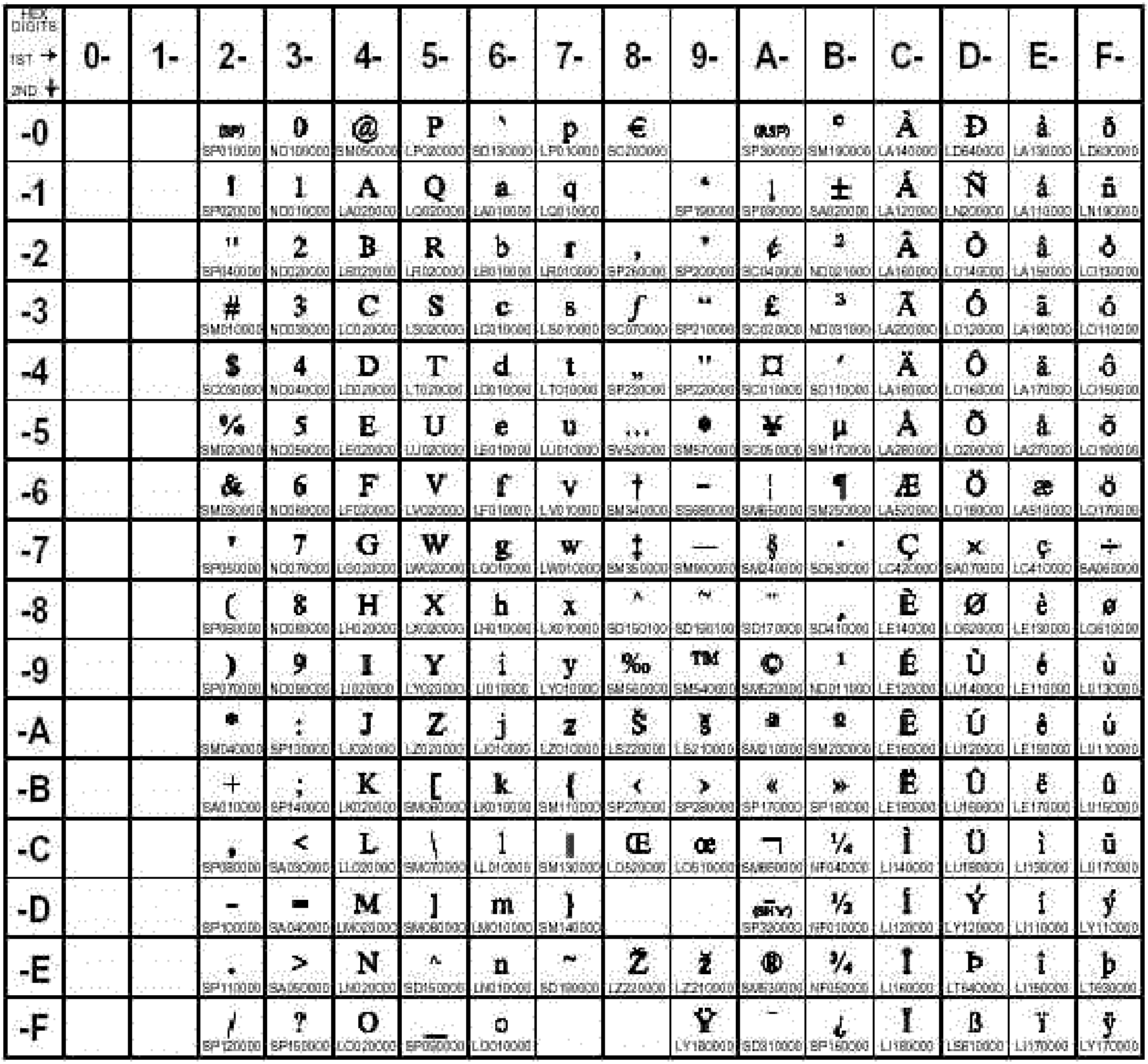
What is Character Encoding?

All data that is stored, transmitted, or processed by a computer is in an encoding. An encoding maps each character to a unique numeric representation. For example:

1. You press a key on a keyboard, like the uppercase letter A.
2. The computer assigns the internal numeric representation, that is, a unique hexadecimal number.
3. To display or print the character, the computer uses the font (graphical representation) that matches the numeric representation, that is, the uppercase letter A.

To assign the numeric representation to a character, an encoding uses a code page, which is an ordered set of characters in which a numeric index (code point value) is associated with each character. The position of a character on the code page determines its two-digit hexadecimal number. The first digit of the hexadecimal number is determined by the column, and the second digit by the row. For example, the following is the code page for the Windows Latin1 encoding. The numeric representation for the uppercase A is the hexadecimal number 41, and the numeric representation for the equal sign (=) is the hexadecimal number 3D.

Figure 16.1 *Figure 16.1: Latin1 Encoding Chart*



HEX DIGITS 1ST → 2ND ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0			(SP) SP010000	0 ND100000	@ SM050000	P LP020000	` SD130000	p LP010000	€ SC200000		(RSP) SP300000	° SM190000	À LA140000	Đ LD640000	à LA130000	ð LD630000
-1			! SP020000	1 ND010000	A LA020000	Q LQ020000	a LA010000	q LQ010000		' SP190000	¡ SP030000	± SA020000	Á LA120000	Ñ LN200000	á LA110000	ñ LN190000
-2			" SP040000	2 ND020000	B LB020000	R LR020000	b LB010000	r LR010000	, SP260000	' SP200000	¢ SC040000	² ND021000	Â LA160000	Ò LO140000	â LA150000	ò LO130000
-3			# SM010000	3 ND030000	C LC020000	S LS020000	c LC010000	s LS010000	ƒ SC070000	" SP210000	£ SC020000	³ ND031000	Ã LA200000	Ó LO120000	ã LA190000	ó LO110000
-4			$ SC030000	4 ND040000	D LD020000	T LT020000	d LD010000	t LT010000	„ SP230000	" SP220000	¤ SC010000	´ SD110000	Ä LA180000	Ô LO160000	ä LA170000	ô LO150000
-5			% SM020000	5 ND050000	E LE020000	U LU020000	e LE010000	u LU010000	… SV520000	• SM570000	¥ SC050000	µ SM170000	Å LA280000	Õ LO200000	å LA270000	õ LO190000
-6			& SM030000	6 ND060000	F LF020000	V LV020000	f LF010000	v LV010000	† SM340000	– SS680000	¦ SM650000	¶ SM250000	Æ LA520000	Ö LO180000	æ LA510000	ö LO170000
-7			' SP050000	7 ND070000	G LG020000	W LW020000	g LG010000	w LW010000	‡ SM350000	— SM900000	§ SM240000	· SD630000	Ç LC420000	× SA070000	ç LC410000	÷ SA060000
-8			(SP060000	8 ND080000	H LH020000	X LX020000	h LH010000	x LX010000	ˆ SD150100	˜ SD190100	¨ SD170000	¸ SD410000	È LE140000	Ø LO620000	è LE130000	ø LO610000
-9			) SP070000	9 ND090000	I LI020000	Y LY020000	i LI010000	y LY010000	‰ SM560000	™ SM540000	© SM520000	¹ ND011000	É LE120000	Ù LU140000	é LE110000	ù LU130000
-A			* SM040000	: SP130000	J LJ020000	Z LZ020000	j LJ010000	z LZ010000	Š LS220000	š LS210000	ª SM210000	º SM200000	Ê LE160000	Ú LU120000	ê LE150000	ú LU110000
-B			+ SA010000	; SP140000	K LK020000	[SM060000	k LK010000	{ SM110000	‹ SP270000	› SP280000	« SP170000	» SP180000	Ë LE180000	Û LU160000	ë LE170000	û LU150000
-C			, SP080000	< SA030000	L LL020000	\ SM070000	l LL010000	| SM130000	Œ LO520000	œ LO510000	¬ SM660000	¼ NF040000	Ì LI140000	Ü LU180000	ì LI130000	ü LU170000
-D			- SP100000	= SA040000	M LM020000	] SM080000	m LM010000	} SM140000			(SHY) SP320000	½ NF010000	Í LI120000	Ý LY120000	í LI110000	ý LY110000
-E			. SP110000	> SA050000	N LN020000	^ SD150000	n LN010000	~ SD190000	Ž LZ220000	ž LZ210000	® SM530000	¾ NF050000	Î LI160000	Þ LT640000	î LI150000	þ LT630000
-F			/ SP120000	? SP150000	O LO020000	_ SP090000	o LO010000			Ÿ LY180000	¯ SD310000	¿ SP160000	Ï LI180000	ß LS610000	ï LI170000	ÿ LY170000

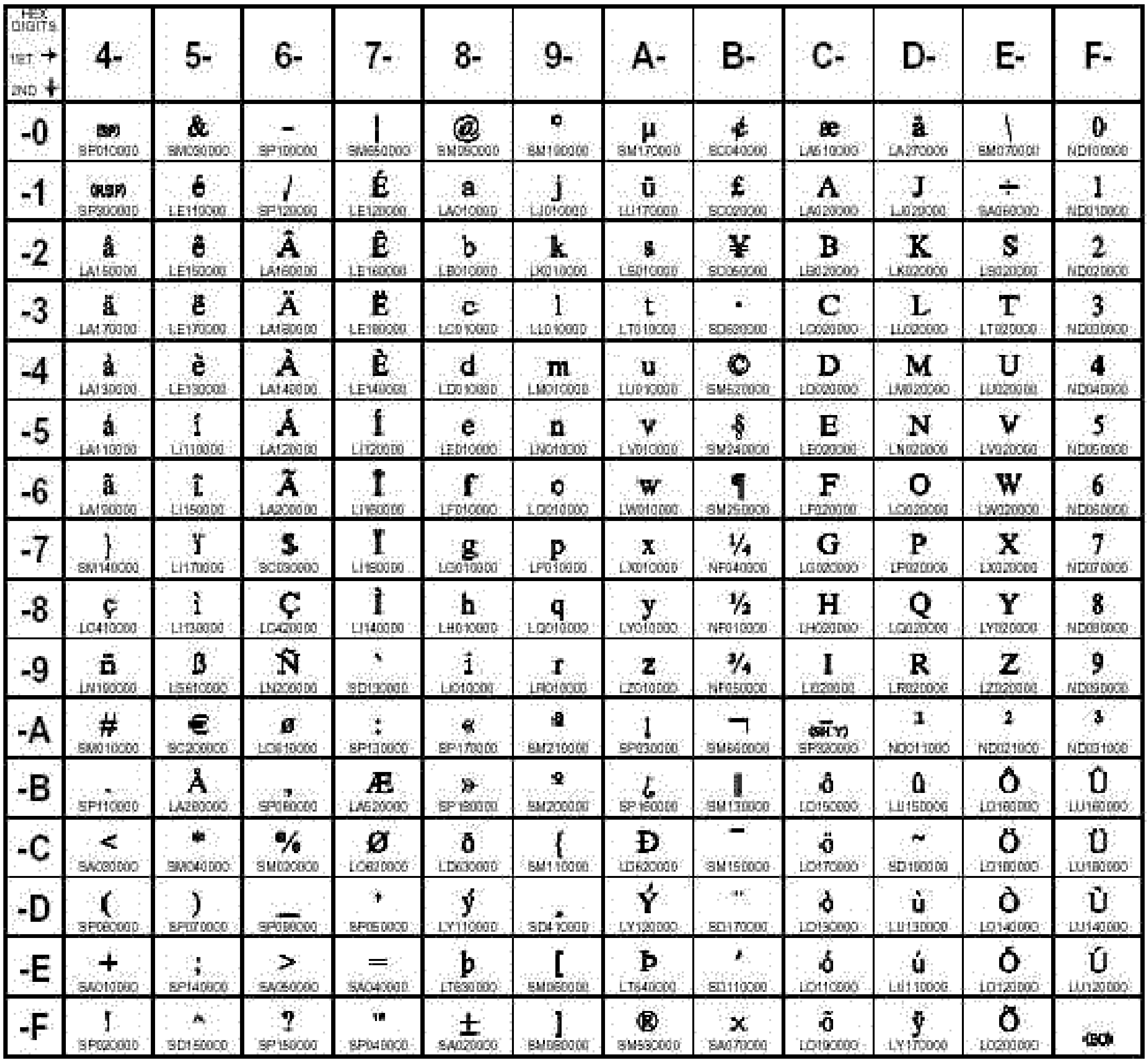
Encoding is the combination of a character set with an encoding method:

- A character set is the repertoire of characters and symbols that are used by a language or group of languages. A character set includes national characters (which are characters specific to a particular nation or group of nations), special characters (such as punctuation marks), the unaccented Latin characters A through Z, the digits 0 through 9, and control characters that are needed by the computer.
- An encoding method is the set of rules that are used to assign the numbers to the set of characters that are in an encoding. These rules govern such things as the size of the encoding (number of bits used to store the numeric representation of the character) and the ranges in the code page where characters are allowed to appear.

When the rules of the encoding method are followed, and numbers are assigned to the characters, the result is called an encoding.

An individual character can have different positions in code pages for different encodings, which result in different hexadecimal numbers. For example, the position of the uppercase letter A in the Wlatin1 code page (shown above) results in the hexadecimal number 41, while in the following Danish EBCDIC code page, the position of the uppercase letter A results in the hexadecimal number C1.

Figure 16.2 *Figure 16.2: Danish EBCDIC Code Page*



HEX DIGITS 1ST → 2ND ↓	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0	(SP) SP010000	& SM030000	- SP100000	| SM650000	@ SM050000	° SM190000	µ SM170000	¢ SC040000	æ LA510000	å LA270000	\ SM070000	0 ND100000
-1	(RSP) SP300000	é LE110000	/ SP120000	É LE120000	a LA010000	j LJ010000	ü LU170000	£ SC020000	A LA020000	J LJ020000	÷ SA060000	1 ND010000
-2	â LA150000	ê LE150000	Â LA160000	Ê LE160000	b LB010000	k LK010000	s LS010000	¥ SC050000	B LB020000	K LK020000	S LS020000	2 ND020000
-3	ä LA170000	ë LE170000	Ä LA180000	Ë LE180000	c LC010000	l LL010000	t LT010000	· SD630000	C LC020000	L LL020000	T LT020000	3 ND030000
-4	à LA130000	è LE130000	À LA140000	È LE140000	d LD010000	m LM010000	u LU010000	© SM520000	D LD020000	M LM020000	U LU020000	4 ND040000
-5	á LA110000	í LI110000	Á LA120000	Í LI120000	e LE010000	n LN010000	v LV010000	§ SM240000	E LE020000	N LN020000	V LV020000	5 ND050000
-6	ã LA190000	î LI150000	Ã LA200000	Î LI160000	f LF010000	o LO010000	w LW010000	¶ SM250000	F LF020000	O LO020000	W LW020000	6 ND060000
-7	} SM140000	ï LI170000	$ SC030000	Ï LI180000	g LG010000	p LP010000	x LX010000	¼ NF040000	G LG020000	P LP020000	X LX020000	7 ND070000
-8	ç LC410000	ì LI130000	Ç LC420000	Ì LI140000	h LH010000	q LQ010000	y LY010000	½ NF010000	H LH020000	Q LQ020000	Y LY020000	8 ND080000
-9	ñ LN190000	ß LS610000	Ñ LN200000	` SD130000	i LI010000	r LR010000	z LZ010000	¾ NF050000	I LI020000	R LR020000	Z LZ020000	9 ND090000
-A	# SM010000	€ SC200000	ø LO610000	: SP130000	« SP170000	ª SM210000	¡ SP030000	¬ SM660000	(SHY) SP320000	¹ ND011000	² ND021000	³ ND031000
-B	. SP110000	Å LA280000	, SP080000	Æ LA520000	» SP180000	º SM200000	¿ SP160000	| SM130000	ô LO150000	û LU150000	Ô LO160000	Û LU160000
-C	< SA030000	* SM040000	% SM020000	Ø LO620000	ð LD630000	{ SM110000	Ð LD620000	¯ SM150000	ö LO170000	~ SD190000	Ö LO180000	Ü LU180000
-D	(SP060000	) SP070000	_ SP090000	' SP050000	ý LY110000	¸ SD410000	Ý LY120000	¨ SD170000	ò LO130000	ù LU130000	Ò LO140000	Ù LU140000
-E	+ SA010000	; SP140000	> SA050000	= SA040000	þ LT630000	[SM060000	Þ LT640000	´ SD110000	ó LO110000	ú LU110000	Ó LO120000	Ú LU120000
-F	! SP020000	^ SD150000	? SP150000	" SP040000	± SA020000	] SM080000	® SM530000	× SA070000	õ LO190000	ÿ LY170000	Õ LO200000	(EO)

Common Encodings

There are many encodings that address the requirements of different languages. Very few languages use only the 26 characters A through Z of the Latin alphabet. In addition, there are different encodings to address different operating system standards.

An encoding that represents each character in one byte is a single-byte character set (SBCS). A single-byte character set can be either 7 bits (providing up to 128 characters) or 8 bits (providing up to 256 characters). An example of an 8-bit SBCS is the Latin1 encoding (represents the characters of Western Europe). (Note that the term octet, for the international community, is an 8-bit byte. Since a byte is not 8 bits in all computer systems, octet provides an unambiguous term.)

A multiple-byte character set (MBCS) is a mixed-width encoding in which some characters consist of more than one byte. For example, the Japanese, Korean, Simplified Chinese, and

Traditional Chinese are MBCS encodings. A double-byte character set (DBCS) is a specific type of an MBCS encoding that includes characters that consist of two bytes.

The following are common encodings:

ASCII (American Standard Code for Information Interchange)
: Is a 7-bit encoding for the United States that provides 128 character combinations. The encoding contains characters for uppercase and lowercase English, American English punctuation, base 10 numbers, and a few control characters. The set of 128 characters is the one common denominator that is contained in most encodings, excluding EBCDIC-based encodings. ASCII is used by personal computers.

ISO (International Organization for Standardization) 646 family
: Is a 7-bit encoding that is an international standard and provides 128 character combinations. The ISO 646 family of encodings is like ASCII except for 12 code points for national variants. The 12 national variants represent specific characters needed for a particular language.

EBCDIC (Extended Binary Coded Decimal Interchange Code) family
: Is an 8-bit encoding that provides 256 character combinations. There are multiple EBCDIC-based encodings. EBCDIC is used on IBM mainframes and most IBM midrange computers. EBCDIC follows ISO 646 conventions to facilitate translations between itself and 7-bit ASCII-based encodings. Characters A through Z and 0 through 9 are mapped to the same code points on all EBCDIC code pages, while the rest of the code points can be used for special characters and national characters, depending on the encoding.

ISO 8859 family and Windows family
: Is an 8-bit extension of ASCII that supports all of the ASCII code points and adds 12 more, providing 256 character combinations. Latin1, which is officially named ISO-8859-1, is the most frequently used member of the ISO 8859 family of encodings. In addition to the ASCII characters, Latin1 contains accented characters, other letters needed for languages of Western Europe, and some special characters.

Unicode
: Uses two bytes for each character rather than one and provides up to 65,536 character combinations. Unicode can handle the scripts of basically all of the world's languages. For example, the Japanese language, which has thousands of characters, uses a 16-bit, multiple-byte character set. There are various forms of Unicode, including UTF-8, UTF-16, and UTF-32.

Moving Data across Environments with Different Encodings

Transcoding

Although it's easy to move data across environments that use the same encoding, it can be more difficult to move data across environments that use different encodings. When the encoding of a file is incompatible with the computer environment's encoding, transcoding occurs.

Transcoding is the process of mapping data from one encoding to another, such as mapping data from an ASCII-based encoding to an EBCDIC-based encoding. Transcoding is not translating from one language to another; transcoding is remapping of characters.

For example, consider a file that was created on a UNIX platform that uses the Latin1 encoding, then moved to an IBM mainframe that uses the Danish EBCDIC encoding. When the file is processed on the IBM mainframe, the data is remapped from the Latin1 encoding to the Danish EBCDIC encoding. If the data contains a dollar sign ($), the hexadecimal number is converted from 24 to 67.

Transcoding can occur in the following situations:

- when you move a SAS file from one platform to another and the file's encoding is incompatible with the current session encoding. An example might be moving a SAS file from a z/OS operating environment with an EBCDIC-based encoding to a Windows operating environment with an ASCII-based encoding.
- when you share data between two SAS sessions (like in a client/server environment) that have incompatible session encodings.
- when you read and write an external file.

How Base SAS Transcodes Data

Base SAS provides transcoding when you move data and applications from one environment to another. To transcode one encoding to another, SAS uses translation tables, like the one that maps Wlatin2 (Windows) to ISO Latin2 (UNIX).

For example, when you

- use the CPORT and CIMPORT procedures to create a transport file, SAS automatically uses translation tables to transcode one encoding to another and back again. First, the data is converted from the source encoding to transport format, then the data is converted from the transport format to the target encoding.
- process a SAS data set that has an encoding that is different from the current session encoding, SAS automatically uses CEDA (cross environment data access) software to transcode data. (CEDA is the same software in SAS that converts a SAS file to the correct data representation when you move a file from one platform to another.)

Base SAS Encoding Behavior

Overview of Base SAS Encoding

For Base SAS files (not SPD Server), the encoding support depends on the version of SAS that created the file:

- Data sets created in SAS 9 automatically have an encoding attribute, which is stamped in the descriptor portion of the file.
- Data sets created in SAS 8 do not have an encoding value stamped on the file; they are assumed to be in the session encoding of the host environment.

The NLS features in SPD Server only support encoding from SAS 9.

SAS 9 Output Processing

For SAS 9 data sets (not SPD Server), encoding is determined as follows:

- For a new output file, the data is written to the file using the current session encoding.

- For a new output file that is created with the OUTREP= option, which specifies a data representation different from the current session, the data is written to the file using the default session encoding for the operating system that is based on the specified OUTREP= value.
- For output processing that replaces an existing file, the new file inherits the encoding of the existing file.
- For output processing that replaces an existing file that is from another platform or if the existing file has no encoding stamped on it, the current session encoding is used.

SAS 9 Input Processing

For input (read) processing in SAS 9 (not SPD Server), encoding behavior is as follows:

- If the session encoding and the encoding that is stamped on the file are incompatible, the data is transcoded to the session encoding. For example, if the current session encoding is ASCII and the encoding that is stamped on the file is EBCDIC, SAS transcodes the data from EBCDIC to ASCII.
- If a file does not have an encoding stamped on it, SAS transcodes the data only if the file's data representation is different from the current session.

Reading and Writing External Files

SAS reads and writes external files using the current session encoding. SAS assumes that the external file is in the same encoding as the session encoding. For example, if you are creating a new SAS data set by reading an external file, SAS assumes that the file's encoding is the same as the session encoding. If it is not, the data could be written to the new SAS data set incorrectly.

Setting the Encoding for Base SAS Sessions

When SAS 9 is installed, the Base SAS (not SPD Server) default encoding is host dependent and is determined by the default settings for several SAS system options. Here are three system options that you should be familiar with:

ENCODING=
: establishes the session encoding, which is the encoding that SAS uses to process syntax, process SAS data sets, and read and write external files. The default value is host dependent; all are SBCS encodings:

Table 16.1 *Default Session Encodings*

Host	Value	Description
OpenVMS	Latin1	Western (ISO)
z/OS	OPEN_ED_1047	OpenEdition EBCDIC cp1047-Latin1
UNIX	Latin1	Western (ISO)
Windows	WLatin1	Western (Windows)

LOCALE=
: specifies the locale of the SAS session. The locale reflects the local language, conventions, and culture for a particular geographical region. A locale's conventions can include the formatting of dates, times, and numbers, and printer preferences like paper size. Specifying a locale also automatically sets the default encoding that establishes the session encoding; a locale has a common encoding that is used most often for a particular operating environment. The default locale is English, and the common encodings for English are the defaults above for ENCODING=.

NONLSCOMPATMODE | NLSCOMPATMODE
: provides national language compatibility for non-English data processing using native characters. For SAS 9, the default is NONLSCOMPATMODE, which provides consistency for running SAS on multiple systems. NONLSCOMPATMODE specifies that data is to be processed in the encoding that is set by the ENCODING= or LOCALE= system option.

Changing the Encoding for Base SAS Sessions

You can change the session encoding by using the LOCALE= system option, the ENCODING= system option, or both. Note that valid values for both options are host dependent.

Here is how you can set the Base SAS (not SPD Server) session encoding when NONLSCOMPATMODE is specified:

- You can specify the LOCALE= system option in a configuration file, at SAS invocation, in an OPTIONS statement, or in the SAS System Options window. In SAS 9, several NLS-related system options are automatically set, based on the value of LOCALE=. Most customers will implicitly set encoding with the LOCALE= system option.
- You can specify the ENCODING= system option in a configuration file or at SAS invocation.
- Here is how LOCALE= and ENCODING= interact:
  - If a value is not specified for ENCODING= (that is, the installation default is set), then specifying a value for LOCALE= sets the encoding based on the LOCALE= value. In addition, values for the following system options are set based on the LOCALE= value: DFLANG=, TRANTAB=, DATESTYLE=, and PAPERSIZE=.
  - If a value is specified for ENCODING=, that value sets the session encoding and overrides LOCALE=.
  - If the value specified for LOCALE= is not compatible with the value specified for ENCODING=, then the value for LOCALE= is used. A warning message is provided if ENCODING= and LOCALE= conflict.
- If the DBCS system option is set, which specifies that SAS process DBCS encodings, the values for DBCSLANG= and DBCSTYPE= system options determine the session encoding and the locale. These options are used for Asian languages or for English with DBCS extensions.

Here is an example of implicitly setting the Base SAS (not SPD Server) session encoding based on the specified locale when you invoke SAS:

```
sas9 -explorer -locale spanish
```

Here is an example of explicitly setting the Base SAS (not SPD Server) session encoding with the OPTIONS statement:

```
options encoding=wlatin2;
```

TIP Changing encoding for a SAS session does not affect SAS keywords, which remain in English, or SAS log output, which also remains in English.

NLS Support in SPD Server

Overview of NLS Support

SPD Server contains support for a subset of the SAS 9 NLS functions documented above. SPD Server uses encoding and locale currently only on SAS software.

In future releases of SPD Server, the locale identifier information will be used for locale-sensitive case folding and linguistic collation. Case-folding is defined as a process applied to a sequence of characters, in which those identified as non-uppercase are replaced by their uppercase equivalents. Linguistic collation is performing linguistic sorts based on linguistic sort keys. However, those functions have yet to be implemented in SPD Server production code.

All tables that are produced by SPD Server and SAS inherit the SAS session's default encoding and locale settings. By default, SPD Server code expects new tables to follow the current SAS session's encoding and locale. Table updates that append rows or update existing rows will perform transcoding to ensure that appended and updated table rows match the existing table encoding.

Wire transfer is in the character set encoding of the SAS session for transfers to and from the SPD Server host, unless SPD Server transcoding has been disabled. SPD Server transcoding is enabled or disabled by inserting a [NO]NLSTRANSCODE statement in the SPD Server **spdsserv.parm** parameter file.

SPD Server NLS Limitations

Affected Data

SPD Server hosts are restricted in the way they handle NLS character strings. SPD Server hosts are restricted to data that is contained in character columns in data sets and some metadata structures. The NLS support for SPD Server is functional for only table labels and variable labels.

Column names, index names, table names, and catalog names are not supported in the SPD Server NLS support. Column names, index names, table names, and catalog names are still dependent on ASCII support. SPD Server SQL is subject to the NLS same restrictions.

Pass-Through SQL

SPD Server Pass-Through SQL does not support any NLS functions. Pass-through SQL operates in the encoding and locale of the SAS session that initiates the CONNECT to SASSPDS.

Case Folding and Sort Sequences

SPD Server NLS code supports very limited English Latin1 and Polish Latin2 case folding for SBCS encodings. UTF8 case folding is limited to the ASCII range of UTF8 encoding.

NLS Sort sequences for SPD Server 4.5 are restricted to lexical sorts for all combinations. Linguistic sorting is a subject for future SPD Server releases.

Indexes and Ordering

Indexes in SPD Server are created in the table's encoding, and only support lexical ordering. If the client's encoding and locale settings match the SPD Server host table's encoding and locale settings, index use is unrestricted. Otherwise, index usage is restricted to certain predicates in WHERE clauses that can be safely interpreted according to the table's encoding and locale settings. When the client and host table encoding and locale settings differ, the EVAL2 strategy is used to filter predicates that require use of order.

Date and Time Representations

SPD Server server-side functions and formats that produce or accept textual date, time, and date/time representations are not locale-sensitive.

Suppressing Transcoding

You can suppress transcoding in the SPD Server environment by entering the following into the **spdsserv.parm** options:

```
NONLSTRANSCODE;
```

If you add the NONLSTRANSCODE option to your spdsserv.parm file, character transcoding between the SPD Server host and connected clients is disabled. Disabling character transcoding restricts the types of operations that the SPD Server host performs to operations it can safely perform, where host and client tables share the same encoding. Disabling SPD Server host transcoding assumes that the client will perform any needed transcoding on the data streams that it sends and receives to match the encoding of referenced tables. The SPD Server host setting for NONLSTRANSCODE does not perform any actions to deny client access to a host table that has mismatched encoding.

LIBNAME Option Restrictions:

The following options are not implemented in the SPD Server NLS functions:

The LIBNAME option

```
OUTENCODING=<client-server encoding>
```

is not supported and will produce a WARNING message if submitted to **sasspds**.

In addition, the related data set option

```
ENCODING=<client-server encoding>
```

is supported by the SAS LIBNAME engine for OUTPUT data sets only. Character data is assumed to be in the encoding of the session that initiates the CONNECT to SASSPDS and is normally stored using that encoding. ENCODING= will cause SPD Server to transcode from the SAS session encoding to the specified encoding for storing data. If you specify ENCODING= for a data set that is not an OUTPUT data set, and if the encoding value that you specify does not match the data set's encoding, when you open the data set, SPD Server produces a warning:

```
ENCODING= specified on table open fails to match table
encoding. Option ignored.
```

The LIBNAME option

```
TRUNCWARN=YES
```

Suppresses hard failure on NLS transcoding overflow and character mapping errors. When using the TRUNCWARN=YES LIBNAME option, data integrity can be compromised because significant characters can be lost in this configuration. The default setting is NO, which causes hard Read and Write stops when transcode overflow or mapping errors are encountered. When TRUNCWARN=YES, and an overflow or character mapping error occurs, a warning is posted to the SAS log at data set close time if overflow occurs, but the data overflow is lost.

Part 6

Appendix

Chapter 17
SPD Server Frequently Asked Questions

SPD Server Frequently Asked Questions

Does SPD Server support files that are larger than 2 Gigabytes in size?

Yes. SPD Server does so by breaking up larger files into partitions that are smaller than 2 Gigabytes. The SPD Server host performs this function automatically and it requires no special syntax.

Can I create file systems that are larger than 2 Gigabytes in size?

Yes, if you use a volume manager that lets you create file systems greater than 2 Gigabytes. SAS recommends this practice.

How do SPD Server client and server processes communicate?

An SPD Server client communicates with three SPD Server processes.

When a client issues a LIBNAME assignment to the SPD Server host, the client communicates with the SPD Server Name Server process using the HOST= and SERV= options that were specified in the LIBNAME connection. The HOST= option specifies the host system where the SPD Server Name Server is running, and the SERV= option is the well-known port number of the SPD Server Name Server that was specified when the software was started. The SPD Server Name Server ensures that the domain of the LIBNAME assignment is valid, then returns the HOST= and SERV= option settings to the client. This ends the interaction of the client with the SPD Server Name Server for that LIBNAME assignment. The client communicates with the SPDSSERV process to complete the LIBNAME assignment.

The SPDSSERV process authenticates the USER and PASSWORD portion of the LIBNAME assignment, and validates whether the USER has access to the domain. If the LIBNAME is successfully authenticated, the SPDSSERV process forks and executes a user proxy, the SPDSBASE process, which continues to service all other client requests for that LIBNAME connection. Subsequent LIBNAME assignments from the same client that are resolved to the same SPD Server user and SPDSSERV context are passed directly to SPDSBASE for processing without any further SPDSSERV interaction. (No further interaction is required because the authentication is inherited by subsequent LIBNAME assignments.)

LIBNAME assignments from the same client for a different SPD Server user or LIBNAME assignments to a domain that is serviced by a different SPDSSERV results in a new SPDSBASE process to service that LIBNAME assignment.

Connections that use the record-level locking option LOCKING=YES to connect to a server in any domain are handled differently. All LIBNAME assignments share the same SPDSBASE record level locking process. When the LOCKING=YES option is in force, instead of forking and executing a new user proxy, the SPDSSERV process initiates communication with the shared LOCKING=YES SPDSBASE process and the client.

How do I know which ports must be surfaced through an Internet firewall?

There are two ports that the SPD Server Name Server uses that you can specify using command-line options. The **listenport** option defines the port that must be used by clients (such as SAS) in LIBNAME and SQL CONNECT statements. The listenport option can also define the port that an ODBC data source requires to communicate with the SPD Server Name Server. The **operport** option defines a second port that is used for various command communications from SPD Server utilities. Either of these ports can be specified using well-known port definitions in the operating system's services file, instead of specifying them on the command-line. In UNIX systems, this is typically the `/etc/services` file. In the services file, the **spdsname** specification corresponds to **listenport**, and the **spdsoper** setting corresponds to the **operport** setting. Both of these ports should be surfaced through the firewall.

The SPDSSERV process uses two types of ports. The first type of port is a port that SPDSSERV uses for local machine communications, internal to SPD Server. The second type of ports is ports that must be accessed by SPD Server clients.

Ports in the first category are not discussed here, because they do not need to be visible beyond the local machine, and therefore do not need firewall connectivity. There are two ports in the second category. The first port in the second category is the port that is defined by the SPDSSERV **listenport** command-line option that performs LIBNAME authentication of the SPD Server user and password, and validates access to the SPD Server domain. The second port in the second category is the port that is used for various communications from SPD Server utilities, and is defined by the SPDSSERV -operport command-line option.

The SPDSSERV **listenport** and **operport** specifications are registered in the SPD Server Name Server by the SPDSSERV process when it starts. Both specifications are returned to the SPD Server client from the SPD Server Name Server when it maps the LIBNAME domain to an SPDSSERV. If you do not specify a **listenport** or **operport** in the SPDSSERV command-line, any port that is available is used. Both of these ports should be specified in the SPDSSERV command-line and surfaced through the firewall.

Ports that the SPDSBASE process uses also fall into the two same categories. The first type of ports is used for local machine communications that are internal to SPD Server. The second type of ports is ports that must be accessed by SPD Server Clients. Like the SPDSSERV process, the SPDSBASE process only cares about the ports that outside clients need to access through an Internet firewall.

The way that the SPDSBASE processes use ports is complex and requires a range of port numbers that are declared using the SPD Server MINPORTNO=/MAXPORTNO= server parameter specifications. The MINPORTNO= and MAXPORTNO= parameters must both be specified to define the range of port numbers that are available to communicate with SPD Server clients, and, therefore, require access from outside of the firewall. If the SPD Server parameters for MINPORTNO= and MAXPORTNO= are not specified, the SPDSBASE processes uses any port that is available to communicate with the SPD Server client.

How many port numbers need to be set aside for SPDSBASE proxy processes? Each SPDSBASE process produces its own operator port that can be accessed using command-line specifications issued by an SPD Server client. In addition, each SPD Server table that is opened creates its own port. Each table's port becomes a dedicated data transfer connection that is used to stream data transfers to and from the SPD Server client. SPD Server table ports are normally dynamically assigned, unless the MINPORTNO= and MAXPORTNO= parameters have been specified. If the MINPORTNO= and

MAXPORTNO= parameters have been specified, SPD Server table ports are assigned from within the specified port range.

Therefore, it follows that the range of ports that is specified for the MINPORTNO= and MAXPORTNO= parameters must consider the peak number of concurrent LIBNAME connections that are made to the server, as well as the I/O streams that are channeled between the SPDSBASE processes and the SPD Server clients.

The following ports must be surfaced for access beyond the firewall:

- Two SPD Server Name Server ports, **listenport** and **operport** , must be surfaced for access beyond the firewall. This is also true for any other ports that are identified in SPDSNAME and SPDSOPER services.
- Two SPDSSERV ports: **listenport** and **operport** , as well as any other ports that are identified in SPDSSERV_SAS and SPDSSERV_OPER services.
- Any other ports that are defined in the MINPORTNO= and MAXPORTNO= range that is specified in the spdsserv.parm file.

How does SPD Server interact with multi-homed hosts?

A multi-homed host is a machine that has two or more IP addresses. For SPD Server to work properly on host machines that have more than one IP address, you must define which IP address you want to associate with the socket bind calls. Socket bind calls listen for the SPD Server Name Server and the SPDSSERV processes. You use the SPDSBINDADDR environment variable to define the preferred IP address. You set the SPDSBINDADDR environment variable in the **rc.spds** script that you use to initiate the SPD Server Name Server and SPDSSERV processes on the SPD Server host machine.

Can I use standard UNIX backup procedures?

Yes. SPD Server files are standard files. If all the components of a table are in the same directory, then you can use the standard backup utility. This is our recommendation. SPD Server includes an incremental backup utility. .

What do I need to know about SPD Server installation? How long does it take?

The SPD Server install is quick and easy to do. The hard-copy installation instructions and shell scripts that are included on the installation media guides you through the installation process. Installation and verification take less than an hour. You might need additional time if you have several SAS client platforms to update.

On UNIX, the SPD Server installation can be performed using a non-privileged UNIX account, although to implement all recommendations, UNIX root privilege is required.

Is it necessary to run UNIX SPD Server as root?

No. SAS recommends that you use a UNIX user ID **other than root** to run your production SPD Server environment. While there are no known security or integrity problems with the current SPD Server release, root access is not required to run the SPD Server environment when you properly configure the UNIX directory ownership and permissions on your LIBNAME domains. There is no real benefit from running the SPD Server package as root, other than possibly convenience. You should carefully consider whether any convenience you might obtain justifies the potential risk from running as root.

What is the SPD Server Name Server, and why do I need it?

All access to SPD Server is controlled and managed by the SPD Server Name Server. All clients first connect to the Name Server, which acts as a gateway to named SPD Server domains. The Name Server maintains a dynamically updated list of valid SPD Server hosts and LIBNAME domains. When a user client needs a domain connection, the Name Server parses the requested LIBNAME domain into a physical address, and then creates a proxy connection to the corresponding SPD Server host. The SPD Server Name Server means that users do not have to keep track of the physical addresses of SPD Server hosts. The only server that an SPD Server client has to know about is the Name Server, which handles the details of connecting SPD Server client users to the appropriate domains.

Does every SPD Server client need a UNIX ID or Windows Networking ID?

No. SPD Server does not use UNIX or Windows networking IDs for login security. Each SPD Server client must have a valid SPD Server ID in order to login to the server. Access to the server is controlled by this ID. Access to individual data is controlled by ACLs that are created by the owner of the data.

Can an SPD Server host, SPD Server Name Server and an SPD Server client all run on the same machine?

Yes, they can. In fact, this even boosts performance because the client engine uses direct access where possible instead of issuing requests to the server. For example, the SPD Server client can perform direct reads from disk. WHERE clause evaluation and index retrieval are faster, too.

Can I have multiple SPD Server hosts on the same machine?

Yes. They can either be all connected to the same SPD Server Name Server or different SPD Server Name Servers. Within each Name Server, all SPD Server LIBNAME domains must be unique.

How do I create LIBNAME domains?

LIBNAME domains are defined in a LIBNAME startup file. The required SPD Server command-line option, **`-libnamefile`**, specifies the LIBNAME startup file. Each entry in this file has the form

```
LIBNAME=ldname pathname= <LIBNAME-path-specification> ;
```

where

LIBNAME= specifies the argument for the LIBNAME domain name that SPD Server clients need to reference

PATHNAME= specifies the full UNIX or Windows path where the SPD Server data tables reside.

How do I specify a LIBNAME domain in SAS?

LIBNAME domains are defined by using a SAS LIBNAME statement. A sample syntax is

```
LIBNAME sample sasspds 'ldname' server=spdshost.spdsname user='johndoe' prompt=yes ;
```

where

sample is the name of the libref

sasspds is the name of the SPD Server engine

ldname is the LIBNAME domain

spdshost is the IP name of the node that is running the Name Server

spdsname is the port number that the Name Server uses

johndoe is the SPD Server login ID

prompt is the prompt for password Y | N

Is there anything else I have to change to run my existing SAS applications?

Typically, no. Once the librefs have been assigned, your existing SAS application runs unchanged.

How can I get existing data loaded into an SPD Server table?

There are several ways to accomplish this. Here are the three most common:

1. Use PROC COPY:

   ```
   PROC COPY
    in=old
    out=spds
    memtype=data ;
   run ;
   ```

   This copies the data and build any existing indexes automatically.

2. Use the DATA Step and SET statement:

   ```
   DATA spds.a ;
    set old.a ;
    run ;
   ```

   This copies the data. You have to specify the indexes that you want to build.

   ```
   DATA spds.a(index=(z));
    set old.a ;
   run ;
   ```

   This copies the data and create an index on variable Z.

3. Use the Microsoft Windows ODBC driver.

Also, see “Migrating Tables between SAS and SPD Server” on page 41, which examines table conversions.

Can SPD Server create indexes in parallel?

Yes, SPD Server can create multiple indexes at the same time. It does this by launching one thread per index and driving them all at the same time. You can accomplish this with

```
PROC DATASETS lib=spds ;
  modify a(asyncindex=yes) ;
  index create x ;
  index create y ;
  index create comp=(x y) ;
quit;
```

In the above example, X, Y, and COMP are created in parallel. Notice the ASYNCINDEX=YES data set option in the MODIFY statement.

```
%LET spdsiasy=YES ;
PROC DATASETS lib=spds ;
  modify a ;
   index create x ;
   index create y ;
  modify a ;
   index create
   comp=(x y)
   comp2=(y x) ;
quit ;
```

In the above example, X and Y are created in parallel; COMP and COMP2 are created in a second parallel index create as soon as the first pair completes. Notice the use of the SPDSIASY macro variable to specify parallel index creation. In this example, a table scan is required for each batch of indexes identified for creation in parallel: one table scan for the X and Y indexes and a second table scan for the COMP and COMP2 indexes.

How many indexes should you create in parallel? It depends on how many CPUs are in the SMP configuration, available disk space for index key sorting, and other tasks. Some results show that on an 8-way UltraSparc, you can create four indexes in almost the same time it takes to create 1. You can group index creates to minimize table scans or auxiliary disk space consumption, but generally there is an inverse relationship between the two: minimizing table scans requires more auxiliary disk space and vice versa. The Help documentation contains more information about "Parallel Index Creation" on page 181.

Does SPD Server append indexes in parallel?

Yes, SPD Server appends indexes in parallel by default.

What are ACLs and how do I use them to control access to data tables?

ACLs define who can access a data table and what type of access they are granted. Currently, there are four levels of access defined: Access List Entry, Owner Access, Group Access, and Universal Access. Every SPD Server user has access to at least one group. During login, an SPD Server user must specify a particular ACL group if the SPD Server password file has the user entered as a member of more than one group. Every data table has an ACL owner and the owner's ACL group attached to it. The precedence of the access levels is the following:

- Access List Entry
- Owner Access
- Group Access
- Universal Access

Types of access are Read, Write, Alter, and Control. To create access lists you must have CONTROL access. The owner by default has control access. For more information, refer to the Help section in the SPD Server 4.4 Administrator's Guide on The ACL Command Set.

How do I get a list of the SAS macro variables introduced for SPD Server?

In a SAS session, get into PROC SPDO and issue the SPDSMAC command. For example:

```
LIBNAME foo sasspds ... ;
PROC SPDO lib=foo ;
SPDSMAC ;
```

For more information, see the list of macro variables in the Help section on Chapter 12, "SPD Server Macro Variables ," on page 205.

What about unique indexes? Can I do something to speed appends?

You can use the SPDSAUNQ=YES on page 224 server option to speed up appends to unique indexes.

What about disk compression for SPD Server tables?

You can request compression for an SPD Server table by using the COMPRESS= data set option. You can also set a macro variable named SPDSDCMP to the same value that you would set in the COMPRESS= option. This causes compression on all data sets you generate without explicitly specifying COMPRESS= on each DATA step. SPD Server compresses your table set by "blocks" and the way you control this amount is through the IOBLOCKSIZE= table option. Once you create a compressed table, the compression block size (that is, the number obs/block) cannot be changed. You must PROC COPY the data set to a new data set with a different IOBLOCKSIZE= on the output data set.

In any case, you select the default SPD Server compression by asserting COMPRESS=YES or using %let SPDSDCMP=YES. The default compression algorithm is a run-length compression.

What about estimates for disk space consumption when using SPD Server?

Overview of Disk Space Consumption

The answer to this question depends on what type of component file within the SPD Server data you need to estimate. Recall that there are three classes of component files that make up an SPD Server table: metadata, data, and indexes. You always get the first two for every table. You get an index component file for each index that you create on the table.

Metadata space consumption

The approximate estimate here is:

```
SpaceBytes = 12Kb + (#columns * 120) + (5Kb * #indexes)
```

This estimate increases if you delete observations from the table or use compression on the table. In general, the size of this component file should not exceed approximately 400K.

Data space consumption

The estimate here is for uncompressed tables:

```
SpaceBytes = #rows * RowLength
```

Your space consumption for compressed tables obviously varies with the compression factor for your table as a whole.

Hybrid index space consumption

The hybrid index uses two data files. The .hbx file contains the global portion of the hybrid index. You can estimate space consumption approximately for the .hbx component of a hybrid index as follows:

If the index is NOT unique:

```
number_of_discrete_values_in_the_index * (22.5 +
  (length_of_columns_composing_the_index))
```

If the index IS unique:

```
number_of_descrete_value_in_the_index * (6 +
  (length_of_columns_composing_the_index))
```

The .idx file contains the per-value segment lists and bitmap portion of the hybrid index. Estimating disk space consumption for this file is much more difficult than the .hbx file. This is because the .idx file size depends on the distribution of the key values across the rows of the table. The size also depends on the number of updates and appends performed on the index. The .idx files of an indexed table initially created with "n" rows consumes considerably less space than the .idx files of an identical table created and with several append or updates performed afterward. The wasted space in the latter example can be reclaimed by reorganizing the index.

With the above in mind, a worst case estimate for space consumption of the .idx component of a hybrid index is:

```
8192 + (number_of_descrete_values_in_more_than_one_obs  * (24 +
  (avg_number_of_segments_per_value * (16 + (seg_size / 8)))))
```

This estimate does not consider 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 (seg_size/8) component of the algorithm.

To estimate the disk usage for a nonunique hybrid index on a column with a length of 8, where the column contains 1024 discrete values, and each value exists in an average of 4 segments, where each segment occupies 8192 rows, the calculation would be:

```
.hyb_size = 1024 * (22.5 + 8) = 31323 bytes

.idx_size = 8192 + (10000 * (24 + (4 * (16 + (8192/8))))) = 4343808 bytes
```

To estimate the disk usage of a unique hybrid index on a column with a length of 8 that contains 100000 values would be:

```
.hyb_size = 100000 * (6 + 8) = 1400000 bytes


.idx_size = 8192 + (0 * (...)) = 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.

There is a hidden workspace requirement when creating indexes or when appending indexes in SPD Server. This need arises from the fact that SPD Server sorts the rows of the table by the key value before adding the key values to the hybrid index. This greatly improves the index create and append performance but comes with a price requiring temporary disk space to hold the sorted keys while the index create and append is in progress. This workspace is controlled for SPD Server by the WORKPATH= parameter in the SPD Server host parameter file.

You can estimate workspace requirements for index creation as follows for a given index "x":

```
SpaceBytes ~ #rows * SortLength(x)
```

where #rows = Number of rows in the table if creating; number of rows in the append if appending.

```
if KeyLength(x) >= 20 bytes
  SortLength(x) = (4 + KeyLength(x))
if KeyLength(x) < 20 bytes
  SortLength(x) = 4 + (4 * floor((KeyLength(x) + 3) / 4))
```

For example, consider the following SAS code:

```
DATA foo.test ;
  length xc $15 ;
  do x=1 to 1000 ;
   xc = left(x) ;
   output ;
  end ;
run ;

PROC DATASETS lib=foo ;
  modify test ;
  index create x xc xxc=(x xc) ;
quit ;
```

For index X, space would be:

```
SpaceBytes = 1000 * (4 + (4 * floor((8 + 3) / 4)))
           = 1000 * (4 + (4 * floor(11 / 4)))
           = 1000 * (4 + 4 * 2)
           = 12,000
```

For index XC, space would be:

```
SpaceBytes = 1000 * (4 + (4 * floor(15 + 3) / 4)))
           = 1000 * (4 + (4 * floor(18 / 4)))
```

```
          = 1000 * (4 + 4 * 4)
          = 20,000
```

For index XXC, space would be:

```
SpaceBytes = 1000 * (4 + 23)
           = 1000 * 27
           = 27,000
```

There is one other factor that plays into workspace computation: Are you creating the indexes in parallel or serially? If you create the indexes in parallel by using the ASYNCINDEX=YES data set option or by asserting the SPDSIASY macro variable, you need to sum the space requirements for each index that you create in the same create phase.

As is noted in the FAQ example about creating SPD Server indexes in parallel, "Can SPD Server create indexes in parallel?" on page 303, the indexes X and Y constitute a create phase, as do COMP and COMP2. You would need to sum the space requirement for X and Y, and for COMP and COMP2, and take the maximum of these two numbers to get the workspace needed to complete the PROC DATASETS indexes successfully.

The same applies to PROC APPEND runs when appending to the table with indexes. In this case all of the indexes are appended in parallel so you would need to sum the workspace requirement across all indexes.

How can I estimate the transient space needed to perform PROC SORT / BY processing?

Workspace is required for SPD Server sorting just as it is required for SPD Server sorted index creation. There are two modes of sorting in SPD Server: tag and non-tag sorting. In either case you sort based on the columns selected in the BY clause. The difference is in the auxiliary data that is carried along by the sort in addition to the key constructed from the BY columns. The default for SPD Server is to use the non-tag sort.

In the case of non-tag sorting, SPD Server carries along the entire row contents (that is, all columns) as the auxiliary data for the key. In the mode of tag sorting, SPD Server only carries along the row ID that points back to the original table row as the auxiliary data. You control the amount of a sort problem that fits in memory at one time by the SPD Server parameter SORTSIZE. Obviously, for a given sort size the number of sort records that fits is a function of the sort mode(#records = SORTSIZE / (SortKeyLength + AuxillaryLength)). When the sort problem does not fit in one SORTSIZE bin, the bins are written to workspace on disk and then merged back to make the final sorted run.

Estimating the disk space required for SPD Server sorting depends on the mode.

For non-tag sorting the estimate is

```
SpaceBytes = #rows * (SortKeyLength + 4 + RowLength)
```

For tag sorting the estimate is

```
SpaceBytes = #rows * (SortKeyLength + 8)
```

So there is a very obvious question here: Since non-tag sort requires so much more space than a tag sort, why would you ever choose a non-tag sort, much less make it the default? The answer lies in the post-processing phase required for the tag sort. When the tag sort completes all you have is the sorted list of row IDs. You must probe the table using the row IDs to return the rows in the desired order. This generally means a highly randomized I/O access pattern to the original table that can add significantly to the time to complete the BY clause. There is definitely a trade-off between tag and non-tag sorting. The critical factors are the row length, the total number of rows to process, and the clustering of consecutive row IDs in the final ordering.

What should I set WORKPATH= to?

The default server parameter file (that is, the **spdsserv.parm** file) sets the WORKPATH= to `/var/tmp`. Generally, this is inadequate for even moderate SPD Server usage. More than likely, the path `/var/tmp` or `/tmp` at your site has a small amount of space allocated, and it is used by other system and application programs that create temporary files. In addition, these file systems are frequently configured as a memory-mapped file system by the system administrator (that is, mounted as a tmpfs file system). Our experience has been that neither `/var/tmp` nor `/tmp` is a suitable choice for WORKPATH, given the space and performance limitations of a typical system configuration.

We strongly recommend that you configure WORKPATH to use a volume manager file system that is striped across multiple disks, if possible, to provide the optimum performance and to allow adequate temporary workspace for the collection of SPD Server proxy processes that will be running concurrently on your server hardware.

How do I, as a LIBNAME domain owner, allow others to create tables in my domain?

For example, Tom is a LIBNAME domain owner, and he wants to give Fred access to create tables in Tom's domain. Tom needs to do the following:

Table 17.1 *SAS Code to Give Access to User "Fred"*

SAS code, by line	Remarks
`LIBNAME dmowner sasspds "tomdom" host="samson" serv="5555" user="tom" passwd="tompw" ;`	• **dmowner** is the libref for the location of the SPD Server data. • **tomdom** is the previously established SPD Server domain. • **host=** specifies the name of the computer where SPD Server resides. • **serv=** is followed by the port number of the SPD Server's Name Server. • **passwd=** is followed by the required password for tom.
PROC SPDO lib=dmowner ;	PROC SPDO opens the command set that allows the user **tom** to change ACLs in the **tomdom** domain using the libref **dmowner**.
set acluser tom ;	SET ACLUSER command allows ACLs under user ID **tom** to be modified.
add acl/LIBNAME ;	Command to add the ACL for a LIBNAME domain. LIBNAME is the syntax used to indicate the LIBNAME domain assigned, which is **tomdom** to the libref that PROC SPDO is started with, which is **dmowner**.

SAS code, by line	Remarks
modify acl/LIBNAME fred=(Y,Y,,) ;	Modifies the ACL in the LIBNAME domain ACL to give user ID **fred** Read and Write access to the **tomdom** domain.
quit ;	

Fred can now connect to the TOMDOM domain and create tables.

How does the system administrator list the access control lists for "user 1"?

To see the ACL privileges for a domain, the system administrator lists them for each user.

For this to work, your SPD Server user ID must be previously set up to have the SPECIAL (level 7) privilege, to use the ACLSPECIAL=YES option on a LIBNAME statement.

Table 17.2 *Code to List ACLs*

Command from command prompt >	Remark

LIBNAME test saspds
'temp'
server=servname.7880
prompt=yes ;
	Issue LIBNAME statement for test domain, specify server and port number, ask system for a password prompt.

user="username" aclspecial=YES ;	aclspecial=YES now gives "username" access to special ACL commands, such as setting a new user ID.
PROC SPDO lib=test ;	Connects to the temp LIBNAME domain using the libref test.
set acluser user1 ;	Sets the SPD Server user scope to user1.
list acl _all_ ;	Lists all ACLs owned by user1.

The resulting output, described in the table below, lists all of the tables in "test".

Table 17.3 *Results from List Command*

Resulting output from list acl _all_; command	Remarks
The SAS System 10:58 Tuesday, November 17, 2003	System message
ACL Info for A.DATA	This ACL affects table A if table A exists and user1 is the owner or has ACL control of table A.
Owner = USER1	USER1 created and owns the A.DATA ACL.

Resulting output from list acl _all_; command	Remarks
Group = TECH	This ACL was created while user1was connected with an ACL group of TECH. All group permissions affect the permissions of the members of the TECH ACL group.
Default Access (R,W,A,C) = (Y,N,N,N)	R=Read; W=Write; A=Alter (rename, delete, or replace tables) C=Control (define and update ACLs for a table) Y=Yes; N=No; Universal privileges are limited to read on table A.DATA.
Group Access (R,W,A,C) = (N,N,N,N)	Users in the ACL group TECH have no privileges on table A.DATA.
The SAS System 10:58 Tuesday, November 17, 2003	
ACL Info for NTE*.DATA	NTE*.DATA refers to a set of tables, which begin with NTE. ACLs of this kind are created using the generic option. If you create a specific ACL for a table that starts with NTE, the specific ACL overrides the generic ACL.
Owner = user1	
Group = TECH	
Default Access (R,W,A,C) = (N,N,N,N)	
Group Access (R,W,A,C) = (Y,Y,N,N)	Users from the ACL group TECH have Read and Write access to tables with names that start with NTE.

How do I change existing PROC SQL code that works with SAS to query SPD Server tables?

Overview

You do not have to change your PROC SQL code. The way to do this is to wrap your code inside a CONNECT statement, which points to the location of the SPD Server tables. This technique is referred to as the pass-through facility. Normal operating system and ACL privileges apply to the user ID making the query during the CONNECT process. Your PROC SQL code should work with a few exceptions. For more information, see "Differences between SAS SQL and SPD Server SQL " on page 146. .

Once you establish a working CONNECT statement that points to the location of your SPD Server tables, you can assign a LIBNAME to the SPD Server table path with a libref command. This enables the simple name that you assign to the SPD Server table to be used in the SQL query, which keeps your SQL query as short as possible.

Here are four progressive examples:

- "Example 1: PROC SQL query, designed to work with a SAS data set, with a two-level SAS filename example." on page 311 shows PROC SQL that works with SAS.

- "Example 2: PROC SQL query, without using the pass-through facility, pointing to an SPD Server table, with a two-level SAS filename." on page 311 shows how you can access SPD Server tables without using the pass-through facility.
- "Example 3: PROC SQL query, using pass-through facility, pointing to an SPD Server table, with a LIBNAME example, with SQL code modified, to avoid using a two-level SAS filename." on page 312 shows how you can access the SPD Server tables by changing your PROC SQL code.
- "Example 4: PROC SQL query, using the pass-through facility, pointing to an SPD Server table, executing a libref statement on the server, so that existing code can be used "as is"." on page 313 shows how you can use the original PROC SQL code from the first example, wrapped with a CONNECT statement, so that it can query SPD Server tables.

Example 1: PROC SQL query, designed to work with a SAS data set, with a two-level SAS filename example.

Table 17.4 *Example 1 Code*

Code	Remarks

`/* Issue a LIBNAME statement which   */`
`/* creates a LIBREF called "test"    */`
` LIBNAME test '/path/for/your/data' ;`
`/* Query using base LIBREF of test   */`
` PROC SQL ;`
` select sum(table1+table2)`
`   as pass,`
` carrier from test.carriers`
` where carrier in('AA','JI')`
`   and bstate='TX'`
` group by carrier ;`
` quit ;`
	This is an example of SQL code that works with SAS. The code contains a two-level SAS filename reference, which is typical for PROC SQL, but it does not work if we attempt to use it inside a pass-through CONNECT statement.
	
	Each of the following examples shows variations of this code, modified to access SPD Server information. We also discuss the pros and cons of each method.

Example 2: PROC SQL query, without using the pass-through facility, pointing to an SPD Server table, with a two-level SAS filename.

Why would you want to do this? You might NOT want to do this, because without the pass-through facility, all the processing is done on the CPU of the client machine. When processing large tables, this is impractical, if not impossible.

Table 17.5 Example 2 Code

Code	Remarks:

`/* Issue a SPD Server (mkt) library */`
`/* LIBNAME statement */`
 	
`LIBNAME mkt sasspds 'mkt'`
`server=servername.4228`
`user='anonymous' ;`
`PROC CONTENTS data=mkt.carriers ;`
`run ;`
`/* query spds LIBREF (mkt) */`
`/* (two-level SAS filename) */`
`PROC SQL;`
`select sum(table1+table2)`
`as pass, carrier`
`from mkt.carriers`
`where carrier in`
`('AA','JI')`
`and bstate='TX'`
`group by carrier ;`
`quit ;`
	This example shows you how to make a query against SPD Server tables, using your original SQL code, without using the SQL pass-through facility.

Example 3: PROC SQL query, using pass-through facility, pointing to an SPD Server table, with a LIBNAME example, with SQL code modified, to avoid using a two-level SAS filename.

Why would you want to do this? By modifying your SQL code slightly, you can use the pass-through facility to have SPD Server perform the work and send the results to the client.

Table 17.6 Example 3 Code

Code

`//* Query spds LIBREF (mkt) (pass-through one-level LIBREF )*/`
`PROC SQL;`
`connect to sasspds`
`(dbq='mkt'`
`serv='8770'`
`user='anonymous'`
`host='localhost') ;`
`select *`
`from connection`
`to sasspds`
 	
`(select sum(table1+table2)`
`as pass,`
`carrier`
`from carriers`
`where carrier`
`in('AA','JI')`
`and bstate='TX'`
`group by carrier) ;`
`quit ;`

Example 4: PROC SQL query, using the pass-through facility, pointing to an SPD Server table, executing a libref statement on the server, so that existing code can be used "as is".

Why would you want to do this? Without modifying your SQL code, you can use the pass-through facility so SPD Server performs the work and sends the results to the client.

Table 17.7 *Example 4 Code*

Code

```
 PROC SQL ;
 connect to sasspds
  (dbq='mkt'
   serv='8770'
   user='anonymous'
   host='localhost');
/* Issue passthru LIBREF (mkt) for use */
/* in two-level queries                */
 execute(LIBREF mkt)
   by sasspds;
/* Query the SPD Server LIBREF (mkt)   */
/* that is a pass-through LIBREF       */
select *
 from connection
 to sasspds

(select sum(table1+table2)
  as pass,
carrier from mkt.carriers
  where carrier in('AA','JI')
  and bstate='TX'
group by carrier) ;
quit;
```

Can I use pass-through async to create multiple indexes on a single existing table?

No. Multiple create indexes on the same existing table are not supported with async.

PROC DATASETS can be used to create indexes in parallel for a single existing table.

For example:

```
PROC DATASETS lib=foo ;
modify customer(
  asyncindex=yes
  bitindex=(state) ;
index create state ;
index create phoneno ;
index create custno ;
index create totsales ;
quit ;
```

Can I use pass-through async to create multiple indexes on existing tables?

Yes. As long as you create only one index per table, the index creation can be run with async.

For example, to create an index State on table Customer, an index Totals on table Billing, and an index Orderno on table Orders asynchronously, you use the following code:

```
execute(begin async operation)
  by sasspds ;

execute(create index state on customer(state))
  by sasspds ;

execute(create index totals on billing(totals))
  by sasspds ;

execute(create index orderno on orders(orderno))
  by sasspds ;

execute(end async operation)
  by sasspds ;
```

What size increases can I expect for tables that are stored in domains with BACKUP=YES?

Tables created in domains that have Backup=YES will have an additional 17 bytes per observation.

What files are created in the SPD Server WORKPATH directory?

Some SPD Server operations can create temporary files that are too large to fit in memory. The SPD Server WORKPATH directory contains these temporary files. The temporary files in the WORKPATH directory are also called *spill files*, because SPD Server spills intermediate files from volatile memory to a temporary file. Temporary files in the WORKPATH directory are removed when the operation that generated the temporary file completes. There are four types of temporary that SPD Server can store in the WORKPATH directory:

Parallel Sort Spill Files
: When SPD Server sorts a table, it uses multiple concurrent threads to sort portions of the table in memory. Each thread creates a sort bin to which it can spill temporary results files to. When the sort threads complete, the sort bin contents are merged to produce the final result. The sort bin files are named using the following convention:

```
spdssr_<pid>_<unique_thread_id>.0.0.0.spds9
```

`pid` is the identifier for the SPDSBASE user proxy that is performing the parallel sort operation.

Parallel GROUP BY Spill Files
: When SPD Server performs a parallel GROUP BY operation, it uses multiple concurrent threads to group intermediate results in parallel. Each thread creates a sort bin to which it can spill temporary results files to. When the GROUP BY threads

complete, the sort bin contents are merged to produce the final result. The GROUP BY bin files are named using the following convention:

```
spdspgb_<pid>_<unique_thread_id>.0.0.0.spds9
```

`pid` is the identifier for the SPDSBASE user proxy that is performing the parallel GROUP BY operation.

Parallel Join Spill Files

When SPD Server performs a parallel join operation, it uses multiple concurrent threads to join portions of the table in memory. Each thread creates a join bin to which it can spill temporary results files to. When the parallel join threads complete, the join bin contents are merged to produce the final result. The join bin files are named using the following convention:

```
spdspllj_<pid>_<unique_thread_id>.0.0.0.spds9
```

`pid` is the identifier for the SPDSBASE user proxy that is performing the parallel join operation.

SQL Temporary Work Files

SPD Server creates temporary work space files for SQL operations that require intermediate results to be spilled to a workspace file. These temporary workspace files are named using the following convention:

```
spds_<pid>_<unique_id>.0.0.0.spds9
```

`pid` is the identifier for the SPDSBASE SQL proxy that created the temporary workspace file.

Debug Log Files

SPD Server can create temporary work space files that contain SPD Server debugging information. The debugging information can be found in the SPD Server SAS log, under headings such as “WHERE Debug” or “SQL Planning.” These temporary workspace files are generally small and are named using the following convention:

```
spdslog_<pid>_<unique_id>.0.0.0.spds9
```

`pid` is the identifier for the SPDSBASE user proxy that created the temporary workspace file.

Your Turn