SAS® Scalable Performance Data Server 5.3: User's Guide

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® Scalable Performance Data Server 5.3: User's Guide*. Cary, NC: SAS Institute Inc.

SAS® Scalable Performance Data Server 5.3: User's Guide

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

October 2016

P1:spdsug

Contents

What's New in SAS Scalable Performance Data Server

Overview

SAS Scalable Performance Data (SPD) Server 5.3 includes support for secure sockets communication, a new driver that enables you to read and write SPD Server tables with two new SAS languages, and documentation enhancements.

Transport Layer Security (TLS)

Beginning with SAS SPD Server 5.3, SPD Server supports secure sockets communication by using Transport Layer Security (TLS), the successor to Secure Sockets Layer Security (SSL). TLS and SSL are cryptographic protocols that are designed to provide communication security. TLS and SSL provide network data privacy by encrypting client/server communication. In addition, TLS performs client and server authentication, and it uses message authentication codes to ensure data integrity.

In the initial release, TLS is supported in the SAS client only. SAS client use of TLS is enabled with the SPDSRSSL macro variable. For more information, see "SPDSRSSL Macro Variable" on page 232. UNIX clients must also specify an appropriate path for the SSLCALISTLOC= system option. This system option is typically set by the server administrator in a configuration file.

New Language Driver

The new language driver enables you to read and write SPD Server tables with the SAS DS2 Language and the SAS FedSQL Language, both of which were introduced with SAS 9.4. The driver is enabled in SPD Server SAS clients by specifying the LIBGEN=YES option in the SPD Server LIBNAME statement. You submit DS2 language statements to the server by using the DS2 procedure. You submit FedSQL language statements by using the FEDSQL procedure. For more information, see "Using the SAS DS2 and FedSQL Languages with SPD Server" on page 16 and "LIBGEN= LIBNAME Statement Option" on page 162.

SAS Federation Server Support for SPD Server Tables

Beginning in February of 2017, SPD Server tables can be accessed for reading, writing, and update by SAS Federation Server. SAS Federation Server is a data server that provides scalable, threaded, multi-user, and standards-based data access technology in order to process and seamlessly integrate data from multiple data sources. The server acts as a hub that provides clients with data by accessing, managing, and sharing SAS data as well as several popular relational databases. SAS Federation Server enables powerful querying capabilities, as well as centralized data source management. With SAS Federation Server, you can efficiently unite data from many sources, without moving or copying the data. For more information about how SPD Server tables are accessed with SAS Federation Server, see *SAS Federation Server: Administrator's Guide*.

Documentation Enhancements

- Beginning in SAS SPD Server 5.3, the documentation for using SPD Server with Hadoop is now contained in its own document: *SAS Scalable Performance Data Server: Processing Data in Hadoop*.
- The user's guide has been reorganized into the following sections: Introduction, Getting Started, SPD Server SQL Processor, Optimizing SPD Server Queries, SPD Server Reference, and ODBC and JDBC Clients.

  The user's guide also now provides reference as well as usage information about the SPD Server LIBNAME statement. See Chapter 3, "Connecting to the Server," and Chapter 17, "SPD Server LIBNAME Statement," on page 149.

  In addition, the guide provides reference information about explicit pass-through statements and server-specific SQL statements. See Chapter 18, "Explicit Pass-Through SQL Statements," and Chapter 19, "SPD Server SQL Statement Additions," on page 183.
- Information about the SPDO procedure, which serves as the operator interface for SPD Server, has been consolidated into one reference chapter, which is published in *SAS Scalable Performance Data Server: Administrator's Guide*.
- The documentation for the following LIBNAME options has been enhanced: LIBGEN=, TEMP=, UNIXDOMAIN=. See Chapter 17, "SPD Server LIBNAME Statement," on page 149.
- The documentation for the following macro variables has been enhanced: SPDSAUNQ=, SPDSDCMP=, SPDSNBIX=, SPDSNIDX=, SPDSSADD=, SPDSSIZE=, SPDSUSAV=, SPDSWCST=. See Chapter 21, "SPD Server Macro Variables," on page 207.
- The documentation for the following table options has been enhanced: COMPRESS=, ENCRYPT=, ENCRYPTKEY=, ENDOBS=, IOBLOCKSIZE=, NOINDEX=, STARTOBS=, SYNCADD=, UNIQUESAVE=. See Chapter 22, "SPD Server Table Options," on page 243.

Part 1

Introduction

Chapter 1

About This Book

Overview

This book describes how SAS Scalable Performance Server (SPD Server) operates, and how to load, create, and manage tables on SPD Server. It assumes that the server and client software have already been installed and configured.

Audience

The primary audience for this book is the database administrator or other person responsible for understanding, serving, and maintaining the organization's data. The book can also be used by application developers who want to create applications that read, write, and query data on SPD Server and by power users who want to create and secure their own tables.

Documentation Conventions

SPD Server supports SAS users and users who do not use SAS. Therefore, the documentation strives to use common terminology that both audiences can understand. This documentation uses the following conventions:

- SAS data sets are referred to as tables.
- When there is a need to distinguish between tables created with the Base SAS engine and tables created with SPD Server, the text refer to "tables" and "SPD Server tables."
- SAS variables are referred to as columns.
- SAS observations are referred to as rows.

The SAS Scalable Performance Data Server product is referred to as “SPD Server” throughout this documentation. The word “server” and “SPD Server” are both used to refer to the server process, depending on the context of the documentation.

Chapter 2

Overview of SAS Scalable Performance Data Server

Introduction to SAS Scalable Performance Data Server

SAS Scalable Performance Data Server (SPD Server) is a product designed for high-performance data delivery. Its primary purpose is to provide user access to large amounts of data for intensive processing (queries and sorts). The product provides full 64-bit processing, supporting up to 2 billion columns and for all practical purposes, unlimited rows of data (potentially petabytes of data).

The product includes the following:

- A server. The server exploits symmetric multiprocessing (SMP) hardware and software architectures in order to process data in concurrent threads in parallel on multiple processors. The server supports a client/server model for data access. Multiple clients can access the server concurrently.
- A SAS client. The SAS client supports SAS languages and procedures and brings the full analytic power of SAS to data on the server.
- ODBC and JDBC clients. These clients enable Windows users who do not use SAS to write JDBC and ODBC code to access the server.
- A C-like application programming interface (API) for writing applications that access SPD Server.
- A collection of utilities for maintaining the server.

The SAS SPD 5.3 server and SAS client operate on computers that run on the third maintenance release of SAS 9.4.

SAS users access the server by using SQL pass-through or by using the SAS language.

Benefits of SPD Server

The server provides the following high-performance features:

- A threaded server system to perform concurrent processing tasks that are distributed across multiple processors. The threading supports parallel loads, parallel index creation, and parallel queries.
- A partitioned, component file structure. Use of component files provides the ability to optimize the I/O throughput by spreading SPD Server metadata, data, and indexes across multiple data paths. The partitioned file structure bypasses the file size limits that are imposed by many applications and operating systems. By using partitioned component files, the server can support any file system transparently.
- Users access data by using domains and a name server, instead of connecting to data directly. Use of domains and a name server enables administrators to manage data storage hardware without affecting users.
- Dynamic cluster tables can be defined to enable users to access many server tables as if they were one table. Users can read and analyze large amounts of data. Administrators can switch out the tables included in the cluster, but still keep the cluster table available, providing for rolling updates. When SPD Server is used with SAS/CONNECT Multi-processing CONNECT software, cluster tables can quickly and easily deliver consolidated results from analytic processes that are run on multiple grid nodes. Multi-processing CONNECT software uses multiple CPUs to process tasks concurrently, in parallel. Each CPU delivers an output table, which becomes a member of the cluster table.
- The product enables you to register users and define security independently of the operating system.

Host Services for Clients

The server provides for the following host services to clients:

- concurrent Read access and retrieval of data.
- high-speed access to very large tables.
- server-side query processing. The server reads, sorts, and subsets entire server tables using a common storage facility, and then returns answer sets. A query subset replaces large file downloads to the client machine, reducing network traffic. The common storage facility enables multiple client users to use the same data on the server without each client having to transfer the data to their workstations.
- leverages client abilities. SPD Server divides the labor. The server retrieves, sorts, and subsets SPD Server data. The client reviews and analyzes the data that the server returns.
- multi-platform support. The product enables clients to share SAS data across computing platforms with other SAS users.

Accessing SPD Server Using SAS

You begin an SPD Server session by establishing a connection to the server from your SAS session. You can use SQL commands to start your SPD Server client session, or you can use a LIBNAME statement. Both methods use the SASSPDS engine and initiate communication between the SAS client machine and SPD Server.

When you access SPD Server with a LIBNAME statement, you can create and access SPD Server data with the DATA step, PROC SQL, PROC COPY, PROC DATASETS, PROC CONTENTS, and other SAS procedures.

SPD Server also supports implicit SQL pass-through and explicit SQL pass-through. When you use implicit SQL pass-through, you submit PROC SQL statements as you would to access any other data source. The client and the server optimize the query for you automatically. When you use explicit SQL pass-through, you submit SPD Server SQL. The server processes the query exactly as it is sent. SPD Server SQL is the same as the SAS SQL language, with minor modifications and some additional statements. For more information about the language, see Chapter 8, “Understanding the SPD Server SQL Processor,” on page 65.

SPD Server Additions to Base SAS

The SPD Server product provides the following server-specific language elements that are available in the SAS session:

- LIBNAME options
- table options
- macro variables
- formats and informats
- SQL statements

Additional SQL statements are available only via explicit SQL pass-through. One of the statements, RESET, enables you to customize the behavior of the server’s SQL processor.

SPD Server also provides the SPDO procedure. PROC SPDO is the operator interface for SPD Server. PROC SPDO is used to perform the following tasks:

- define and manage ACLs
- create and manage cluster tables
- define row-level security on tables
- truncate a table
- refresh SPD Server domains and server parameters on the fly
- manage client proxies
- execute SPD Server utilities.

For more information about the additional language elements, see the following:

- Chapter 17, “SPD Server LIBNAME Statement,” on page 149

- Chapter 22, “SPD Server Table Options,” on page 243
- Chapter 21, “SPD Server Macro Variables,” on page 207
- Chapter 20, “SPD Server Functions, Formats, and Informats,” on page 195
- Chapter 19, “SPD Server SQL Statement Additions,” on page 183

For more information about PROC SPDO, see SPDO Procedure in *SAS Scalable Performance Data Server: Administrator's Guide*.

Other Ways to Access SPD Server

SQL Access Library API

SPD Server provides the SQL access library API (application programming interface), which is known as SPQL, to enable you to write user applications that access the server's SQL processor. SPQL is a C-language compatible interface. For more information, see Chapter 23, “SPD Server Access Library API Reference,” on page 281.

ODBC and JDBC Access

For information about accessing SPD Server with ODBC and JDBC clients, see Chapter 25, “Using SPD Server with ODBC and JDBC Clients,” on page 295.

Using SPD Server

After the server is started, to use SPD Server, you must connect to the server and load or create data on the server. Then, you can query the data and write end-user applications that access the server.

After creating tables, be sure to back up your data. Backups are best done by administrators.For more information, see “Backing Up and Restoring SPD Server Data” in *SAS Scalable Performance Data Server: Administrator's Guide*.

Utilities for Maintaining SPD Server

The utilities for maintaining SPD Server are described in *SAS Scalable Performance Data Server: Administrator's Guide*.

Part 2

Getting Starting with SPD Server

Chapter 3

Connecting to the Server

Overview of Connecting to SPD Server

You can connect to the server by issuing a SAS LIBNAME statement, or by using the SQL CONNECT statement.

- The LIBNAME statement enables you to access server data by using the SAS DATA step and SAS procedures. When you use the LIBNAME statement and specify the IP=YES LIBNAME option, you invoke the implicit SQL pass-through facility for your SQL requests.
- The SQL CONNECT statement connects to the server from within an SQL query. The SQL CONNECT statement invokes the server's explicit SQL pass-through facility.

Regardless of the connection method that you use, your connection request must include the following information:

- the name of the server engine: SASSPDS
- the name of a server domain
- the name of the server host
- the port number of the name server

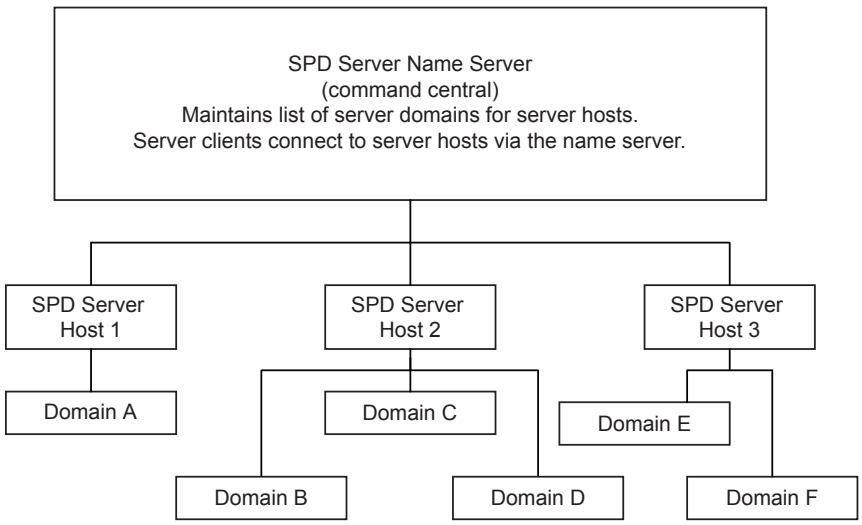
- user authentication parameters.

The domain names, host names, and name server port number that you can use are typically given to you by a server administrator. If ACL security is enabled, the administrator will also give you a server user ID. When UNIX file security is the only form of security that is active for the server, authentication is not required. All resources within the server domain are granted access by UNIX permissions for the server UNIX ID.

Understanding the Name Server

The name server is a server process that serves as the go-between for the SASSPDS engine and SPD Server hosts. Figure 3.1 on page 12 illustrates the role of the name server in making a server connection.

Figure 3.1 *Name Server, SPD Server Hosts, and Server Domains*



The name server serves as command central between clients and the server hosts. It maintains a list of the domains associated with each SPD Server host. Client sessions can connect to a host only through the name server. Direct connections between clients and hosts are not permitted. The name server resolves the submitted domain name to a physical path and then connects you to the server that serves the domain without requiring you to know physical addresses. A server administrator sets up the server domains in a parameter file for the server, which then registers its domains with the name server.

Connect to SPD Server with a LIBNAME Statement

Minimum Connection Parameters

Here is an example of the minimum information needed to establish a server connection with the LIBNAME statement. It establishes a connection to Domain C from the server configuration depicted in Figure 3.1 on page 12.

```
libname mydomain sasspds 'domainC' server=host2.5400
user="mySPDuserid" password="secret";
```

Here is the other information in the request:

mydomain
: a local library reference (libref).

SASSPDS
: the name of the server LIBNAME engine.

'DomainC'
: the server domain.

host2.5400
: the server host name and the port number of the name server. 5400 is the default port number for the name server. The port number might be different in your installation. You can also use a port name instead of a port number, if one has been configured. When a port name is used, SPD Server determines the network address for the named service in the /etc/services file. The default port name is **`spdsname`**.

"*mySPDuserID*"
: the server user ID given to you by the server administrator.

"*secret*"
: the password associated with the server user ID.

  The password that you specify must be valid for the form of authentication that your server is using. For example, if your server is using LDAP authentication, then you must specify your LDAP password. If your server is performing native authentication, you will be given an initial password by the administrator. You must change this password by using the CHGPASS= or NEWPASSWORD= LIBNAME option. For more information, see "Changing Server Passwords" on page 17. Your administrator will tell you the form of user ID authentication that is configured and the requirements.

Alternatives to the Basic Connection Statement

The example above shows one way to specify server connection parameters. There are other ways to specify the host name for the SAS session. Instead of using the SERVER= argument, use one of the following:

- Use the HOST= and SERVICE= arguments as follows:

  ```
  libname mydomain sasspds 'domainC' host="host2.company.com"
  service="5400" user="mySPDuserid" password="secret";
  ```

HOST= enables you to use the IP address or network node name to identify the server host. (SERVER= requires that the node name be used.) SERVICE= specifies the name server port number or port name. For more information, see "HOST= LIBNAME Statement Option" on page 160.

- Create a SAS macro variable named SPDSHOST or an environment variable named SPDSHOST to identify the host server. Whenever a LIBNAME statement does not specify a server host machine, the server looks for the value of SPDSHOST to identify the host server. Here is an example:

```
 %let spdshost=host2;
libname mydomain sasspds 'domainC' user="myuserID" password="secret";
```

Instead of specifying USER= and PASSWORD= to authenticate to the server, you can do one of the following:

- Specify USER= and PROMPT=YES as follows:

```
libname mydomain sasspds 'domainC' server=host2.5400
user="mySPDuserid" prompt=yes;
```

  The server will prompt you for a password. For more information, see "PROMPT= LIBNAME Statement Option" on page 169.

- If your installation is using SAS Metadata Server authentication, use the AUTHDOMAIN= argument. AUTHDOMAIN= allows authentication to the server by specifying the name of an authentication domain metadata object. Here is an example:

```
libname mydomain sasspds 'domainC' server=host2.5400
authdomain=spds;
```

  The AUTHDOMAIN= value is defined by the SAS Metadata Server administrator.For more information, see "AUTHDOMAIN= LIBNAME Statement Option" on page 154. Your server administrator will tell you if metadata server authentication is required.

Understanding User Validation and Authorization

ACL Security

When ACL security is enabled, the server uses a user's SPD Server user ID and ACLs to determine user access to the domain and domain resources. A domain owner has full access to all resources in his or her domain. For other users, the server grants access in the following order:

1. Uses the ACL permissions that belong to the server user ID.
2. Uses the ACL permissions that belong to the server user ID's default ACL group.

A server user ID can have from 5 to 32 ACL user groups defined, depending on how the server is configured. By default, the connection is validated against the permissions defined for the first group in your group list. To connect with the authorizations of a different group from your group list, you can use the ACLGRP= LIBNAME option. ACLGRP= enables you to specify a different group name. Here is an example:

```
libname mydomain sasspds 'domainC' server=host2.5400
user="mySPDuserid" password="secret" aclgrp='prod';
```

For more information, see "ACLGRP= LIBNAME Statement Option" on page 153.

Invoking Special Server Privilege

Server user IDs are registered in a password database. The password database supports privilege levels that confer special privileges. For example, users can be given a privilege to perform tasks like creating ACLs for other users. All connections from the SASSPDS engine are made as a regular user, regardless of the privileges defined in the database. To invoke special privilege in the SAS session, you must specify the ACLSPECIAL= LIBNAME option in the LIBNAME statement as follows:

```
libname mydomain sasspds 'domainC' server=host2.5400
user="mySPDuserid" password="secret" aclspecial=yes;
```

For more information, see "ACLSPECIAL= LIBNAME Statement Option" on page 154.

UNIX File Security Only

ACLs are optional. When only UNIX file security is used, all resources within a domain are granted access through the UNIX ID of the server process.

Invoking Implicit Pass-Through

The default connection reads data from the server and brings the data to the client for processing. To invoke SPD Server implicit pass-through for your SQL requests, specify the IP=YES LIBNAME option in the statement as follows:

```
libname mydomain sasspds 'domainC' server=host2.5400
user="mySPDuserid" password="secret" ip=yes;
```

If you plan to create server tables, also specify the DBIDIRECTEXEC= system option in the SAS session. IP=YES optimizes SELECT queries for the server. DBIDIRECTEXEC= optimizes CREATE TABLE operations as well.

```
option dbidirectexec=yes;
```

For more information about IP=YES, see "IP=YES LIBNAME Statement Option" on page 161. For more information about the DBIDIRECTEXEC= system option, see *SAS/ACCESS for Relational Databases: Reference*. SPD Server does not support the DIRECT_SQL= LIBNAME option or the SQLGENERATION= system option discussed in the SAS/ACCESS documentation.

Manage Network Traffic

If your server installation uses the same physical machine to run your server client process and your server host services, you can use the NETCOMP= and UNIXDOMAIN= options in the LIBNAME statement to improve client/server communication.

- NETCOMP= compresses the data stream in an SPD Server network packet.
- UNIXDOMAIN= uses UNIX domain sockets for data transfers between the client and SPD Server.

For more information, see NETCOMP= on page 166 and "UNIXDOMAIN= LIBNAME Statement Option" on page 175.

Temporary Domains

SPD Server enables you to create temporary server domains that exist only for the duration of the LIBNAME assignment. A temporary server domain creates a temporary space similar to the SAS Work library. To create a temporary server domain, specify a

real domain as usual and specify the TEMP=YES LIBNAME statement option as follows:

```
libname tmp sasspds 'domain' server=host2.5400
user="mySPDuserid" password="secret" temp=yes;
```

When you end your server session, all of the data objects, including tables, catalogs, and utility files in the TEMP=YES temporary domain are automatically deleted. For more information, see "TEMP= LIBNAME Statement Option" on page 173.

Using the SAS DS2 and FedSQL Languages with SPD Server

SPD Server 5.3 contains a driver that enables you to use two new SAS programming languages to read and write SPD Server tables: the SAS DS2 language and the SAS FedSQL language.

The SAS DS2 language is a SAS proprietary programming language that was introduced in SAS 9.4 that is appropriate for advanced data manipulation and applications. It also includes additional data types, ANSI SQL types, programming structure elements, and user-defined methods and packages.

The SAS FedSQL language is a SAS proprietary implementation of the ANSI SQL:1999 core standard that was introduced in SAS 9.4. It provides support for new data types and other ANSI 1999 core compliance features and proprietary extensions. FedSQL brings a scalable, threaded, high-performance way to access, manage, and share relational data in multiple data sources. FedSQL is a vendor-neutral SQL dialect that accesses data from various data sources without submitting queries in the SQL dialect that is specific to the data source. In addition, a single FedSQL query can target data in several data sources and return a single result table.

Access to the server with these languages is enabled by specifying the LIBGEN=YES option in the SPD Server LIBNAME statement. The LIBGEN=YES option configures the SPD Server connection to generate additional domain connections. When these connections are configured, you can submit DS2 language statements from your Base SAS session by using the DS2 procedure. You can submit FedSQL language statements by using FEDSQL procedure.

When using PROC DS2 and PROC FEDSQL to read and write SPD Server tables, there is no need to specify IP=YES to invoke SQL implicit pass-through. These procedures submit the DS2 and FedSQL language statements directly to SPD Server.

The following LIBNAME options are supported for use with PROC DS2 and PROC FEDSQL: ACLGRP=, ACLSPECIAL=, and TEMP=.

For more information about the server connection, see "LIBGEN= LIBNAME Statement Option" on page 162.For more information about the languages, see *SAS 9.4 DS2 Language: Reference, Sixth Edition*, and *SAS 9.4 FedSQL Language: Reference, Fifth Edition*. For more information about the DS2 and FEDSQL procedures, see *SAS 9.4 Procedures Guide, Sixth Edition*.

Other LIBNAME Options

For a complete list of LIBNAME options that are supported in a SASSPDS LIBNAME statement, see Chapter 17, "SPD Server LIBNAME Statement," on page 149.

Changing Server Passwords

If your installation is using native authentication, all server users must change passwords the first time they connect to SPD Server. The LIBNAME statement supports two ways to change your server password:

- Specify the CHNGPASS=YES option in the LIBNAME statement. When CHNGPASS= is specified, the server will prompt you for the new password. Here is an example:

```
libname mydomain sasspds 'domainC' server=host2.5400
user='newhire'
password='whizbang'
chngpass=yes;
```

- Specify the NEWPASSWORD= option with PASSWORD= in the LIBNAME statement. NEWPASSWORD= replaces the old password with a new password.

```
libname mydomain sasspds 'domainC' server=host2.5400
user='newhire'
password='whizbang'
newpassword='her2stay';
```

Connect to SPD Server with Explicit SQL Pass-Through

To connect to the server for explicit SQL pass-through, you must use PROC SQL.

1. Submit an SQL CONNECT statement. The SQL CONNECT statement must specify the SASSPDS engine and server connection options. The SQL CONNECT statement invokes the explicit SQL pass-through facility.
2. Submit SQL statements as follows:
   - Submit SQL statements that do not return a result set in the EXECUTE statement.
   - Submit queries using the SELECT...FROM CONNECTION statement.
3. Terminate the explicit pass-through session with the DISCONNECT statement.

Here is an example of the code necessary to send an explicit SQL pass-through request in the SPD Server environment:

```
proc sql;
connect to sasspds
   (dbq='mydomain'
    host='servername'
    service='5400'
    user='MySPDuserid'
    password='MyPasswd');
execute (SQL-statements) by sasspds;
select * from connection to sasspds
(SELECT-query);
```

```
disconnect from sasspds;
quit;
```

In the CONNECT statement:

- SASSPDS is the name of the SPD Server client engine.
- The arguments in parentheses submit server connection parameters:

  DBQ=
  specifies the server domain.

  HOST=
  specifies a node name or an IP address for the server host. The HOST= argument is optional. If you omit the argument, SPD Server uses the current value of the SAS macro variable SPDSHOST to determine the node name.

  SERVICE=
  specifies the port number for the name server. You can specify the port name instead, if one is configured. When you use a port name, SPD Server determines the network address from the named service in the/etc/services file. The default port name is spdsname.

  USER=
  specifies a server user ID.

  PASSWORD= (or PASSWD=)
  specifies the password associated with the server user ID. The password that you specify must be valid for the form of authentication that your server is using. For example, if your server is using LDAP authentication, then you must specify your LDAP password.

  Note: You can use PROMPT= instead of PASSWORD=.

  PROMPT=YES
  causes SPD Server to prompt for a password. The prompter is case-sensitive.

  Note: You can use PASSWORD= instead of PROMPT=.

Note: If UNIX file security is the only form of security that is active for the server, then USER= and PASSWORD= are not required. All resources within the server domain are granted access by UNIX permissions for the server UNIX ID.

Note: AUTHDOMAIN= can also be used instead of USER= and PASSWORD=.

EXECUTE statement:
The EXECUTE statement enables you to send SQL statements that do not return a result set to the server. The server's SQL processor supports the same SQL statements as PROC SQL, except SELECT. SELECT is not allowed. In addition, the SQL processor supports some SPD Server SQL statements that are available only in explicit pass-through. The PROC SQL statements function a little differently in the server SQL processor than they do in PROC SQL. For more information, see the following:

- "Differences between SAS SQL and SPD Server SQL" on page 67.
- Chapter 19, "SPD Server SQL Statement Additions," on page 183.

SELECT statement:
The SELECT statement establishes a pass-through connection for SELECT queries. See "Differences between SAS SQL and SPD Server SQL" on page 67 for information about the SQL processor's SELECT support.

The DISCONNECT statement ends the SQL pass-through session.

Nesting SQL Pass-Through Access

You can nest server pass-through access. Nesting allows access to data that is stored on two different networks or network nodes. There are two ways to nest access.

- You can use the SPDSENG database to reserve an SPD Server from within an existing SPD Server connection. Here is an example of a nested pass-through connection. On host Datagate, which is on a local network, SQL pass-through is nested to access the EMPLOYEE_INFO table. This table is available on the PROD host on a remote network. (You must have user access to the PROD host.)

```
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;
```

  The connection to the SPDSENG database is specified in a second SQL CONNECT statement, which is submitted in the EXECUTE statement. Note that the SELECT statement and DISCONNECT statement for the second domain are also nested in SELECT and DISCONNECT statements for the first domain.

- If you would prefer not to use the SPDSENG database to reference a server, you can use the LIBGEN=YES option in the LIBNAME statement. Libraries with the LIBGEN=YES option are automatically available in SQL environments. Here is an example of how LIBGEN=YES can be used to perform the same request as the one shown above.

```
libname domain2 sasspds "domain2" host="prod" serv="spdsname"
user='usr2' password='usr2_pw' IP=YES LIBGEN=YES;

proc sql;
 connect to sasspds (dbq='domain1' host='datagate' serv='spdsname'
user='usr1' password='usr1_pw');
   select * from connection to sasspds (select employee_no,
annual_salary from domain2.employee_info);
 disconnect from sasspds;
quit;
```

  For more information, see “LIBGEN= LIBNAME Statement Option” on page 162.

Chapter 4

Loading and Creating Data on the Server

SAS and SPD Server Tables

SPD Server tables have a different physical structure than SAS tables. To use a SAS table with SPD Server, you must convert the table from the Base SAS format to the SPD Server format. Likewise, to use a server table with Base SAS, you must convert the table from the SPD Server format to Base SAS format.

SPD Server's emphasis on complete LIBNAME compatibility makes it easy to convert from one format to the other. When you access SPD Server, the standard procedures used to create and copy tables in SAS apply to server tables as well. That is, when you use PROC COPY, the SAS DATA step SET statement, or the PROC SQL CREATE TABLE AS statement to duplicate a table from a SAS library in an SPD Server domain (or vice versa), the SAS client converts the data from the source format to the target format automatically. This copy and convert process is referred to as “loading.”

You can also create new tables in SPD Server in the same ways that you can create a new table in SAS.

Planning Your Server Tables

Here are best practices when creating and loading tables on SPD Server.

- Large tables have implications for file storage, disk space, and performance. When creating large tables, consult with your server administrator so that file storage and disk space for the server domain can be configured appropriately. You will also want to create the tables with an appropriate partition size. A proper partition size is important for optimal striping of data paths. For large tables, a partition size of 1 GB might be appropriate. Ask your administrator what the value of the MINPARTSIZE= server setting is for your server. MINPARTSIZE= specifies the minimum partition size that can be used by your server. If MINPARTSIZE= is much lower than 1 GB, consider specifying a larger partition size for your table with the PARTSIZE = table option or SPDSSIZE macro variable. SAS macro programs are available from SAS Technical Support to help you calculate a proper partition size. The partition size cannot be changed on an existing table.
- Index or order your data to optimize subsetting of the data by the server. For more information, see “ Understanding SPD Server Indexing” on page 29.
- Compress your data. You can enable table compression in your SAS session with COMPRESS= table option or the SPDSCOMP macro variable.
- When creating tables with the SQL procedure, set IP=YES in the LIBNAME statement. (In library definitions that are created in SAS Management Console, this option is set in the **Advanced** tab.) IP=YES invokes SQL implicit pass-through, which optimizes processing between the client and the server.

For more information about server table options, see Chapter 22, “SPD Server Table Options,” on page 243. For more information about server LIBNAME options, see Chapter 17, “SPD Server LIBNAME Statement,” on page 149. For information about server macro variables, see Chapter 21, “SPD Server Macro Variables,” on page 207.

Formatting Your Data

SPD Server supports the more commonly used SAS formats and informats to enable you to associate a format with a column. It also supports user-defined formats. For a listing of the supported formats and informats, see Chapter 20, “SPD Server Functions, Formats, and Informats,” on page 195.

Table-Loading Techniques

Overview

This section illustrates some of the methods that are available for loading tables into SPD Server. You can load SAS data into SPD Server tables using PROC COPY, DATA step programs, PROC APPEND, and SCL applications. You can also use SQL pass-through to load the server tables. The server SQL statement extensions, LOAD TABLE and COPY TABLE, provide further support. These statements load tables from one SPD

Server domain to another. You will want to use the method that best fits your needs for source location, indexing, and creating subsets of the source data.

The examples in this section load a SAS table named Cars into SPD Server. The SAS table is available from the Sashelp library, which is shipped with all SAS software. The library is available by using the libref Sashelp in your SAS requests.

Load a SAS Table with PROC COPY

PROC COPY can be used to copy the entire content of a SAS library or specified tables from a SAS library to SPD Server. This example shows how to load all the tables in the Sashelp library to SPD Server with PROC COPY. The PROC COPY statement reads the SAS tables in the Sashelp library and writes server tables to the Conversion_Area domain. By default, PROC COPY automatically re-creates any indexes. However, it supports an option to suppress index creation. For more information about the options available with PROC COPY, see COPY Procedure in the *Base SAS Procedures Guide*.

```
libname spds sasspds 'conversion_area'
server=husky.spdsname
user='siteusr1'
prompt=yes;

proc copy in=sashelp out=spds;
run;
```

Using the same LIBNAME statement, this example shows how to load only the Sashelp.Cars table on SPD Server with PROC COPY. You identify the table to copy with the SELECT statement.

```
proc copy in=sashelp out=spds;
select cars;
run;
```

Load a SAS Table with the DATA Step

This DATA step example creates server table Spds.Cars2 from SAS table Cars. The DATA step copies the data but does not rebuild indexes. If you want indexes, you must create them.

```
data spds.cars;
set sashelp.cars ;
run ;
```

Parallel Table Load Technique Using the DATA Step and PROC APPEND

Using the DATA step, you can create an empty table first, defining indexes on the desired columns. Then, use PROC APPEND to populate the table and indexes. The example below demonstrates the technique. This example creates server table Spds.Cars3 from SAS table Cars.

```
/* Create an empty server table with the same */
/* columns and column attributes as the existing */
/* SAS table. */

data spds.cars3 (index=(make origin type));
set sashelp.cars(obs=0);
```

```
run;

/* Use PROC APPEND to append the data in SAS table */
/*  Cars to server table Cars. The append to the */
/* server table and its indexes will occur in parallel. */

proc append
base=spds.cars3
data=sashelp.cars;
run;
```

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.

Load a SAS Table with Implicit SQL Pass-Through

In these PROC SQL examples, the server table Spds.Cars4 is created from SAS table Cars. Be sure to set the IP=YES LIBNAME option in the LIBNAME statement to invoke implicit SQL pass-through. Also, set the DBIDIRECTEXEC=YES system option. The first example copies the data in its entirety. Indexes are not copied. Indexes must be defined separately.

```
libname spds sasspds 'conversion_area'
server=husky.spdsname
user='siteusr1'
prompt=yes
ip=yes;

option dbidirectexec=yes;

proc sql;
create table spds.cars4 as
select * from sashelp.cars;
quit;
```

This example uses a subset of the columns from the SAS table to create server table Spds.Cars5:

```
proc sql;
create table spds.cars5 as
select make, model, origin, type, msrp from sashelp.cars;
quit;
```

Loading Tables between Server Domains

Copy a Server Table

This example copies the SPD Server table Cars3 that was created in “Parallel Table Load Technique Using the DATA Step and PROC APPEND” on page 23 and creates a new server table, Copycars, using the COPY TABLE statement. The COPY TABLE statement functions similarly to PROC COPY. That is, it copies the source table and its indexes in their entirety by default, but enables you to suppress index creation if you choose. You can also specify to order the copied data with a BY statement. For more information, see “COPY TABLE Statement” on page 187. The creation of table Copycars and its indexes occurs in parallel.

```
proc sql;
connect to sasspds (host="husky"
                    service="spdsname"
                    dbq="conversion_area"
                    user="siteusr1"
                    prompt=yes);
execute(
copy table copycars
  from cars3
) by sasspds;
disconnect from sasspds;
quit;
```

Load a Server Table

This example creates new server table Carload from server table Cars3 that was created in "Parallel Table Load Technique Using the DATA Step and PROC APPEND" on page 23. The LOAD TABLE statement creates a new table using content from an existing table. It can load all the data, as shown below. In addition, it supports a SELECT clause and a WHERE clause to enable you to subset the data. For an example of subsetting data with LOAD TABLE, see "Load and Subset a Server Table" on page 25. LOAD TABLE does not re-create indexes. If you want indexes, you must define them. The table creation occurs in parallel.

```
proc sql;
connect to sasspds (host="husky"
                    service="spdsname"
                    dbq="conversion_area"
                    user="siteusr1"
                    prompt=yes);
execute(
load table carload with
   index make
     on (make),
   index origin
     on (origin),
   index model
     on (model)
   as select *
   from cars3
) by sasspds;
disconnect from sasspds;
quit;
```

Load and Subset a Server Table

In this example, you create a subset of SPD Server table Cars3 using the LOAD TABLE statement. The new server table is named Fordcars. The SELECT statement specifies to include only columns Make, Model, Origin, Type, MSRP, and Invoice in the new server table. A WHERE clause specifies to include only rows that have Make="Ford". The creation of table Fordcar and its indexes occurs in parallel.

```
proc sql;
connect to sasspds (host="husky"
                    service="spdsname"
                    dbq="conversion_area"
                    user="siteusr1"
                    prompt=yes);
```

```
execute(
load table fordcars with
   index origin
     on (origin),
   index model
     on (model)) by sasspds;
select * from connection to sasspds
(as select Make, Model, Origin, Type, MSRP, Invoice from cars3
   where make="ford");
disconnect from sasspds;
quit;
```

Table Creation Techniques

Create a Table with the DATA Step

In this DATA step example, you create a new table named SPDS.OLD_AUTOS on the server.

```
libname spds sasspds 'conversion_area' server=husky.spdsname
user='siteusr1' prompt=yes;

data spds.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
;
```

Create a Table with PROC SQL

In this PROC SQL example, you create a new table named SPDS.LOTTERYWIN on the server. The LIBNAME statement includes the option IP=YES to invoke implicit SQL pass-through.

```
libname spds sasspds 'conversion_area'
server=husky.spdsname
user='siteusr1'
prompt=yes
ip=yes;

proc sql;
create table spds.lotterywin (ticketno num, winname char(30));
insert into spds.lotterywin values (1, 'Wishu Weremee');
quit;
```

Enabling User Access to SPD Server Tables

When ACL security is enabled (recommended for all sites), SPD Server grants access rights only to the owner (creator) of the SPD Server resource. Resource owners must grant permissions for the resource to other users.

Resource owners can grant access to all SPD Server users, to an ACL group, and to individual users.

The following properties are available to grant ACL permissions to server users:

READ
: read or query access to the resource

WRITE
: append to or update the resource

ALTER
: rename, delete, or replace a resource, and add or delete indexes associated with a table

CONTROL
: permission to modify the permissions of other users and groups that are associated with this resource.

SPD Server also supports row-level security.

You grant permissions to others by defining an ACL on the resource. ACLs are created with the SPDO procedure. Work with a server administrator to create ACLs and to define row-level security for your tables. For more information about SPD Server security, see "Security Overview " in *SAS Scalable Performance Data Server: Administrator's Guide*.

Chapter 5

Indexing and Sorting Tables

Understanding SPD Server Indexing

Overview of Indexing

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. Indexes can be a source of sort order when performing BY clause processing. Indexes are also used directly by some SAS applications. For example, PROC SQL uses indexes to efficiently evaluate equijoins.

The server supports indexes for queries that require global table views (such as queries that contain BY clause processing or SQL joins) and segmented views (such as parallel processing of WHERE clause statements).

The server can thread WHERE clause evaluations for tables that are not indexed. However, indexes enable rapid WHERE clause evaluations. You should index large tables to optimize server performance. For information about indexing with WHERE, see “Server Indexing with WHERE Clause” on page 129.

Index creation is a CPU-intensive process. When sufficient processing power is available, parallel index creation in the server is highly scalable. The creation process for

each index is threaded. A single index creation can use multiple CPUs on a server if they are available, which greatly improves performance. The server efficiently indexes tables of varying size and data distributions.

There are several ways to define indexes on table columns.

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 statements. 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.

Note: To create multiple indexes requires enough WORKPATH= disk space to create all of the key sorts at the same time. Consult with your server administrator to find out if you have enough disk space.

PROC DATASETS has the flexibility to allow batched parallel index creation by using multiple MODIFY groups. "Parallel Index Creation " on page 31 inserts INDEX CREATE statements between two successive MODIFY statements resulting in a parallel creation group.

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. The 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.

Index Creation Techniques

This section illustrates the various index creation techniques that are available. The sample code shows how both simple indexes and composite indexes are created.

Create Server Indexes in a DATA Step

The following DATA step code creates the server table MyTable. The code uses the INDEX= table option to create a simple server index X on column X, and a composite server index Y on columns (A B).

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

Create Server Indexes with PROC DATASETS

The following PROC DATASETS code creates a simple index and a composite index on server table MyTable.

```
proc datasets lib=spdslib;
  modify mytable;
  index create x;
  index create y=(a b);
quit;
```

Create Server Indexes Using PROC SQL

The following code creates the same simple and composite server indexes that were created in the previous example using PROC SQL.

```
proc sql;
    create index x on spdslib.mytable(x);
    create index y on spdslib.mytable(a,b);
quit;
```

Create Server Indexes Using SQL Explicit Pass-Through

The following code uses SQL explicit pass-through to create a simple index and a composite index:

```
proc sql;
    connect to sasspds (dbq="Conversion_Area" server=husky.spdsname
user='siteusr1' prompt=yes);
    execute( create index x on mytable(x) ) by sasspds;
    execute( create index y on mytable(a,b) ) by sasspds;
quit;
```

Parallel Index Creation

This example creates a SAS table named patient_info and uses PROC DATASETS to create indexes for the table. The SPDSIASY macro variable is set to request parallel execution. The MODIFY statements in the PROC DATASETS request are specified in a way that will support parallel execution.

```
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;
   run;
   modify patient_info;
      index create
        last_name
        first_name;
   run;
   modify patient_info;
      index create
        whole_name=(last_name first_name)
        class_sex=(patient_class patient_sex);
   run;
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.

Using PROC CONTENTS to See Index Information

Sometimes you want to see information about indexes that are associated with a particular table. The PROC CONTENTS table option VERBOSE= provides additional detail about all of the indexes that are associated with a server table. For example, the following PROC CONTENTS code uses the VERBOSE= option to show details about two indexes:

```
proc contents data=mainhs.class (verbose=yes);
run;
```

The result shows the minimum and maximum values for the two indexes in the table and the number of discrete values for each index:

```
Alphabetic List of Index Info:                .
Index                                                    Name
KeyValue (Min):                          Alfred
KeyValue (Max):                          William
# of Discrete values:                     19
Index                                                    age_sex
KeyValue (Min):                          11.000000
KeyValue (Max):                          16.000000
# of Discrete values:                     11
```

```
Data Partsize                                                    16776672
```

Sorting Data

Overview of Sorting Data

SPD Server supports implicit and explicit sorts. An implicit sort is unique to the server. Each time you submit a SAS statement with a BY clause, the server sorts your data, unless the table is already sorted or indexed by the BY column. All BY statements and WHERE statements that appear in a DATA step or SAS procedure are automatically passed to the server. The data returned by the server is a subset based on the WHERE statement, and that subset is implicitly sorted based on the BY statement. Because this happens, there is no need to precede the DATA step or procedures with a PROC SORT. However, if you want to perform an explicit sort, you can use PROC SORT.

Advantages of Implicit Server Sorts

Many SAS job streams are structured with code that alternates PROC SORT followed by another procedure invocation, where the PROC SORT step is needed only for the execution of the other procedure's invocation. When sort order is relevant only to the following step, with SPD Server, you can eliminate the PROC SORT step and just use the BY clause in the procedure. This eliminates the extra data transfer (to PROC SORT from the server and then back from PROC SORT to the server) to store the sorted result. Even if the 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.

Using the 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 and there's no need for a PROC SORT step before the DATA step.

```
data first last;
   set sport.expraqs;
   by price;
   if first.price then output first;
   if last.price then output last;
run;
```

Chapter 6

Creating and Using Dynamic Cluster Tables

Introduction to Dynamic Cluster Tables

Overview

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. One way that the server enables access to large amounts of data is by offering dynamic cluster tables. A dynamic cluster table is a collection of SPD Server tables that are presented to the end-user application as a single table through a metadata layer acting similar to a view. The cluster table is Read-only. However, features are available that enable creators to add, remove, and replace member tables while keeping the cluster table available to end users.

This ability to update the cluster while it remains online provides for rolling updates. When SPD Server is used with SAS/CONNECT Multi-processing CONNECT (MP CONNECT) software, cluster tables can quickly and easily deliver consolidated results from analytic processes that are run on multiple grid nodes. The MP CONNECT software uses multiple CPUs to process tasks concurrently, in parallel. Each CPU delivers an output table, which becomes a member of the cluster table.

Benefits of Dynamic Cluster Tables

- Dynamic cluster tables are virtual table structures. Because cluster metadata is used to manage the data that is contained in the members, the clusters do not require significant disk space beyond that already taken by the member tables.
- Because a dynamic cluster table consists of numerous smaller server tables, the architecture enables parallel loading and processing. Cluster table loads and refreshes can be broken down into multiple tasks that are performed concurrently.
- SPD Server provides a simple, straightforward interface for creating and managing dynamic cluster tables. These statements are available in PROC SPDO and as explicit pass-through statements. For reference information, see the "SPDO Procedure" in *SAS Scalable Performance Data Server: Administrator's Guide*.

Server Authorizations and Cluster Tables

- You must have Control access to any member tables that you use in a dynamic cluster table.
- Access control lists (ACLs) can be defined on a dynamic cluster table after it is created. The permissions that are specified in the dynamic cluster table ACL are applied when the server accesses the dynamic cluster table. Any individual ACL that is defined on a member table does not apply during the time that the member table is part of a dynamic cluster table.

- You must have Control access to the dynamic cluster table itself to destroy the table or undo the cluster.

Anonymous User

There is an ANONYMOUS user ID that any SPD Server user can specify with no password. Any resource (table, view, catalog) that is created with the ANONYMOUS user ID can be viewed and controlled by all users who have access to the domain in which the resource exists. As such, any table created by ANONYMOUS can be used in a dynamic cluster table created by any server user. In addition, all server users can read a dynamic cluster table that was created by ANONYMOUS.

The ANONYMOUS user can place ACLs on a resource to limit access to the resource. ANONYMOUS cannot access any other server user's resources. Server administrators can remove ANONYMOUS access to the server. Consult with your server administrator to find out whether ANONYMOUS access is available on your server.

Creating Dynamic Cluster Tables

Requirements

A dynamic cluster table can be created with one or more server tables. When multiple server tables are used, the tables should have related content. In addition, the tables must meet the following requirements:

- The tables must all be in the same domain.
- The tables must have matching table, column, and index attributes. For more information, see "Member Table Requirements for Creating Dynamic Cluster Tables" on page 50.

Basic Syntax for Creating a Dynamic Cluster Table

You create a dynamic cluster table with the CLUSTER CREATE statement. The easiest way to issue CLUSTER CREATE and other cluster statements is in PROC SPDO. You can also submit the statements using explicit SQL pass-through.

Regardless of how it is submitted to the server, the general form of the CLUSTER CREATE statement is as follows:

```
CLUSTER CREATE cluster-table-name

MEM|MEMBER=member-name-1
MEM|MEMBER=member-name-n
<DELETE=YES|NO>
<MAXSLOT=n>
<UNIQUEINDEX=YES|NO>;
```

cluster-table-name
specifies the name of the cluster table to be created.

member-name
specifies a member table name. Specify a MEM= (or MEMBER=) argument for each member table that you want to include. You must specify at least one member table.

<DELETE=YES | NO>
specifies whether the cluster table and its members can be destroyed with the CLUSTER DESTROY statement. The default setting is NO. When DELETE=NO, you must use the CLUSTER UNDO statement to unbind the cluster before you can delete member tables. Specify YES if you want the ability to use the CLUSTER DESTROY statement to delete the cluster and all its members.

<MAXSLOT=*n*>
specifies the maximum number of slots, or member tables, to be allocated for this cluster table. The default server setting for the MAXSLOT= parameter is -1. This value permits dynamic growth of the number of member tables, up to the specified system maximum value. The system maximum value for the number of slots is specified by the MAXGENNUM column setting in the server's configuration file. If there is a known maximum number of slots to be enforced for a cluster table, it is more efficient to specify the limitation using the MAXSLOT= argument.

<UNIQUEINDEX=YES | NO>
specifies whether the unique indexes that are defined in the member tables should be validated and marked as unique in the dynamic cluster table. The default server setting is YES. The processing that is required to validate the unique indexes depends on the number of rows in the tables. Processing can take considerable time for larger tables. In addition, if you choose to use the validation process but the indexes are not unique, the CLUSTER CREATE statement fails. Specify UNIQUEINDEX=NO if you want to turn off the index validation process.

Example of Creating a Dynamic Cluster Table

Suppose your company generates a server table that contains monthly sales transactions. You have 24 tables, representing the past 24 months. You've been asked to link the tables so that reports can be written that compare sales figures to assist in identifying any trends. You've been told that the reports will be needed twice annually.

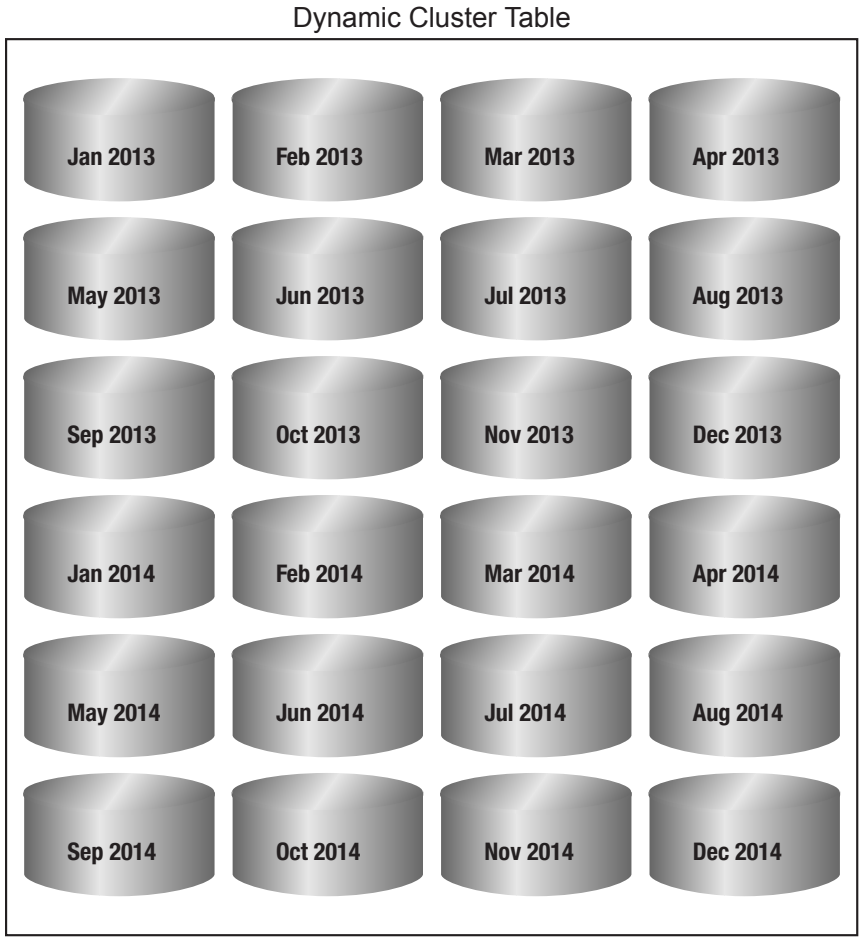
The following example shows the PROC SPDO code that you can use to create a dynamic cluster table named Sales_History with the 24 tables.

Figure 6.1 on page 39 depicts a dynamic cluster table with 24 members.

```
proc spdo library=libref;
   cluster create Sales_History
      mem=sales201301
      mem=sales201302
      mem=sales201303
      mem=sales201304
      mem=sales201305
      mem=sales201306
      mem=sales201307
      mem=sales201308
      mem=sales201309
      mem=sales201310
      mem=sales201311
      mem=sales201312
      mem=sales201401
      mem=sales201402
      mem=sales201403
      mem=sales201404
      mem=sales201405
```

```
    mem=sales201406
    mem=sales201407
    mem=sales201408
    mem=sales201409
    mem=sales201410
    mem=sales201411
    mem=sales201412
quit ;
```

Figure 6.1 *Dynamic Cluster Table*



The PROC SPDO LIBRARY= argument specifies the libref that represents the SPD Server domain that contains the tables to be clustered. The CLUSTER CREATE statement specifies to create a cluster table named Sales_History. The MEM= argument identifies the members of the dynamic cluster table. The other parameters are left to use their default values:

- The cluster table and its members cannot be destroyed.
- There is no limit on the number of members, beyond the system configured value, so additional months of sales data can be added as they become available.
- Unique indexes are validated.

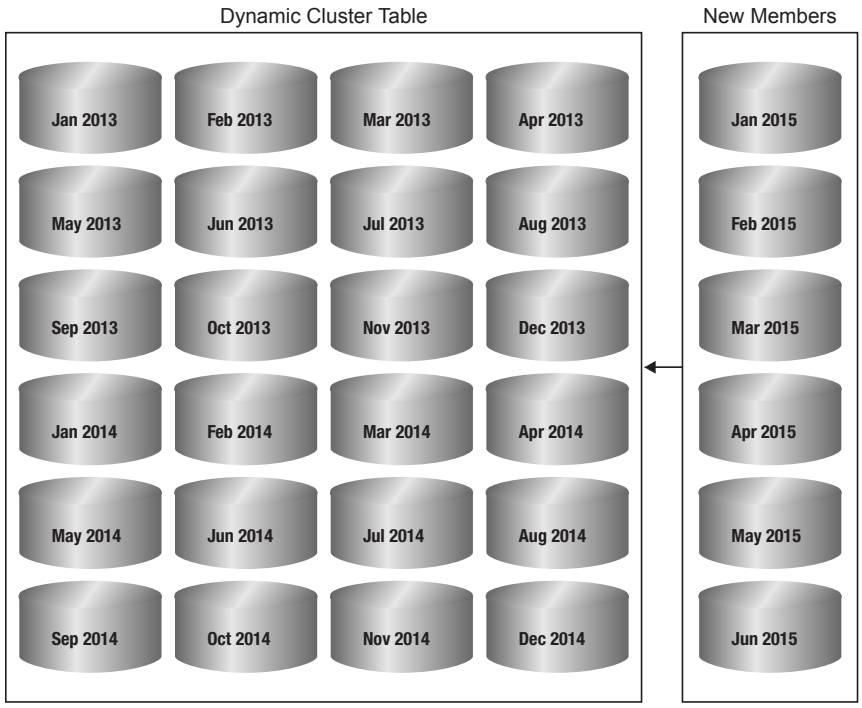
Adding Members to a Dynamic Cluster Table

Suppose that six months later, you have six additional server tables and want to add them to the dynamic cluster table. The tables have table and column attributes that are identical to the existing member tables in the cluster. To add the tables to the original cluster table, you use the CLUSTER ADD statement. The CLUSTER ADD statement also takes arguments *cluster-name* and MEMBER= (MEM=). The following example shows how to issue the CLUSTER ADD statement in PROC SPDO. The PROC SPDO LIBRARY= argument specifies a libref that identifies the target domain.

```
proc spdo library=libref;
   cluster add Sales_History
      mem=sales201501
      mem=sales201502
      mem=sales201503
      mem=sales201504
      mem=sales201505
      mem=sales201506;
quit;
```

Tables that are added with the CLUSTER ADD statement are appended to the end of the member table list. Figure 6.2 on page 40 illustrates the process of adding tables to a cluster.

Figure 6.2 *Adding Member Tables to a Dynamic Cluster Table*



The examples at the end of this chapter contain more extensive code examples of adding to a dynamic cluster table.

Modifying a Dynamic Cluster Table

The initial attributes that are defined for a dynamic cluster table cannot be changed. That is, you cannot modify the number of member slots, the unique index validation setting, or the DELETE= setting of a dynamic cluster table after it has been created. However, if the original server tables did not have the MINMAXVARLIST= attribute defined, you can define this attribute for the cluster table. The MINMAXVARLIST= attribute identifies table columns for which the server will maintain minimum and maximum values in the cluster table's metadata.

An optional MINMAXVARLIST attribute can be added to the cluster table with the CLUSTER MODIFY statement. The form of the CLUSTER MODIFY syntax is as follows:

```
CLUSTER MODIFY cluster-table-name
MINMAXVARLIST=(column-1 column-2 ... column-n);
```

cluster-table-name
specifies the name of the dynamic cluster table to be modified.

column-n
specifies the name of a table column.

The columns that you specify must not already have a definition for the MINMAXVARLIST attribute.

Note: Before you submit the CLUSTER MODIFY statement, you must take the cluster table offline. The CLUSTER MODIFY statement requires that you have exclusive access to the table. When you execute this statement, the server automatically unclusters the dynamic cluster table, makes the requested column modifications to the individual member tables, re-creates the cluster table, and performs a full table scan to initialize the MINMAXVARLIST values in each member table. You must wait for the process to complete before making the table available again. The processor time required for the CLUSTER MODIFY statement is directly related to the sizes of the member tables in the dynamic cluster table.

If an error occurs while the CLUSTER MODIFY statement is running, the dynamic cluster table is not re-created. You will need to re-create it by using the CLUSTER CREATE statement.

Refreshing Dynamic Cluster Tables

Overview of Refreshing Dynamic Cluster Tables

Over time, member tables can age out and need to be removed to make room to accommodate the addition of more current member tables. Or a member table might need to be replaced by an updated version of itself. The process of replacing or updating one or more member tables is referred to as "refreshing" the cluster table.

The server supports several ways to refresh a cluster table:

- You can replace an existing member table with a new member table in place with the CLUSTER REPLACE statement. CLUSTER REPLACE inserts the new member table into the slot vacated by the old member table. You can replace one member

table at a time. CLUSTER REPLACE is not supported on cluster tables that are created with UNIQUEINDEX=YES.

- You can remove one or more tables from the cluster's member list with the CLUSTER REMOVE statement. CLUSTER REMOVE removes the member tables and closes their slot in the member list. You can remove multiple member tables at one time with CLUSTER REMOVE.
- You can append new tables to the end of the member list with the CLUSTER ADD statement.
- You can undo the cluster with the CLUSTER UNDO statement. Then, after making necessary changes to the individual member tables, you can later re-create the cluster with the CLUSTER CREATE statement. When you are making these changes, you must take the cluster offline.

When a member table is removed from a cluster with the CLUSTER REPLACE or CLUSTER REMOVE statement, or a new member is added to the cluster with the CLUSTER ADD statement, users that currently have the cluster table open for reading will remain connected to the table. However, they will not see the changes until the next time they open the cluster table.

Member tables that are removed from a cluster table remain in the domain as regular server tables, although the tables are in a read-only state. If you have a need to update a table that was removed from a cluster, you can use the CLUSTER FIX statement to restore the table to a writable state. For more information, see "Restoring Removed or Replaced Cluster Table Members" on page 44.

Example of Refreshing a Dynamic Cluster Table with CLUSTER REPLACE

The following shows the PROC SPDO code that you use to replace a table in a dynamic cluster table:

```
proc spdo library=libref;
   cluster replace Sales_History oldmem=sales201503 newmem=sales201503-2;
 quit;
```

The PROC SPDO LIBRARY= argument specifies a libref that identifies the server domain. Here is what is done in the CLUSTER REPLACE statement:

- Sales_History is the name of the cluster table to be modified.
- The OLDMEM= argument identifies the member table to be removed.
- The NEWMEM= argument specifies the server table to be inserted in its place.

In this example, the dynamic cluster table is refreshed to replace member table Sales201503 with new member Sales201503-2.

Example of Refreshing Dynamic Cluster Tables with CLUSTER REMOVE and CLUSTER ADD

Here is the PROC SPDO code that you might use to refresh a dynamic cluster table by removing and adding tables.

```
proc spdo library=libref;
   cluster remove Sales_History
      mem=sales201301
      mem=sales201302
      mem=sales201303
```

```
      mem=sales201304
      mem=sales201305
      mem=sales201306;
 cluster add Sales_History
      mem=sales201507
      mem=sales201508
      mem=sales201509
      mem=sales201510
      mem=sales201511
      mem=sales201512;
quit;
```

The PROC SPDO LIBRARY= argument specifies a libref that identifies the server domain that contains the cluster table. Here is what is done in the CLUSTER REMOVE and CLUSTER ADD statements:

- Sales_History is the name of the cluster table to be modified.
- The MEM= argument identifies the tables to be removed and added.

In this example, you are removing the first six tables from cluster Sales_History's member list and appending six new ones to the end of the list.

The CLUSTER REMOVE statement does not require that the member tables be contiguous. However, CLUSTER ADD always appends to the end of the list.

Refreshing Dynamic Cluster Tables with CLUSTER UNDO and CLUSTER CREATE

When you refresh a dynamic cluster table using the CLUSTER UNDO and CLUSTER CREATE statements, CLUSTER UNDO unbinds the dynamic cluster table. For an example of how the CLUSTER UNDO statement is used, see "Undo a Dynamic Cluster Table" on page 43. Then you essentially create a new cluster table with the CLUSTER CREATE statement. This new cluster table can include a combination of original member tables and new or updated member tables, in any combination. During the process, the cluster table is unavailable.

The examples at the end of this chapter contain code that unbinds a dynamic cluster table and re-creates the cluster with different member tables.

Undo a Dynamic Cluster Table

Undoing the dynamic cluster table reverts the table back to its unbound server tables. You might want to use the CLUSTER UNDO statement for the following reasons:

- to refresh a dynamic cluster table that needs many updates. Undo the cluster, and update the member tables as needed. Then, re-create the dynamic cluster table with CLUSTER CREATE.
- to eliminate the cluster and return all of the member tables to their unbound state.

Here is the PROC SPDO code that you would use to undo cluster table Sales_History:

```
proc spdo library=libref;
   cluster undo Sales_History;
quit;
```

Restoring Removed or Replaced Cluster Table Members

You use the CLUSTER FIX statement when you need to restore removed or replaced cluster member tables to a writable state. The following PROC SPDO code repairs server table Sales201512, which once was a member of cluster table Sales_History.

```
proc spdo library=libref;
   cluster fix sales201512;
quit;
```

Destroying Dynamic Cluster Tables

You use the CLUSTER DESTROY statement when you want to delete or destroy an existing cluster table and all of its members. The general form of the CLUSTER DESTROY statement is as follows:

```
CLUSTER DESTROY cluster-table-name;
```

cluster-table-name
the name of the dynamic cluster table that you want to destroy.

Querying and Reading Member Tables in a Dynamic Cluster

You can read specific member tables in a dynamic cluster table by using the MEMNUM= table option. The MEMNUM= table option takes a number that indicates the member table's position in the member table list or the literal LASTCLUSTERMEMBER. When you use the MEMNUM= option, the server opens only the specified member table, instead of opening all of the member tables that belong to the dynamic cluster.

You can determine the member number of a table in a dynamic cluster table by submitting the CLUSTER LIST statement to the server or by using PROC CONTENTS on the dynamic cluster table. Both methods list the member tables of the dynamic cluster.

The general form for the CLUSTER LIST statement is as follows:

```
CLUSTER LIST cluster-table-name
<OUT=output-SAS-table-name>
</VERBOSE>;
```

cluster-table-name
the name of the dynamic cluster table for which you want to see the member list.

<OUT=*output-SAS-table*>
(optional) writes the results of the CLUSTER LIST statement to an output SAS table of the specified name. The table is written to the SAS Work library. To create the

table in a different location, specify a Base engine libref. By default, the CLUSTER LIST statement creates output columns Cluster Name and Member Name.

</VERBOSE>
By default, the CLUSTER LIST statement returns the cluster table name and member names. Specifying /VERBOSE adds columns for Column Name, Minimum Value, and Maximum Value to the output table.

The following example code uses PROC SPDO to generate a list of the member tables in the dynamic cluster table Sales_History, and writes the output to a SAS table named MyLib.OutFile.

```
proc spdo library=libref;
cluster list Sales_History out=mylib.outfile;
```

The following example code uses PROC SPDO to create the dynamic cluster table UsSales. The candidate member tables (Ne_Region, Se_Region, and Central_Region) have the MINMAXVARLIST attribute defined on the numeric column Store_ID in each member table. Then, a CLUSTER LIST statement is issued with the /VERBOSE option. The CLUSTER LIST output displays the dynamic cluster name, the names of each member table in the dynamic cluster, and the MINMAXVARLIST information for each member table. The output data set is written to a SAS library named MyLib.

```
proc spdo library=libref;
 cluster create ussales
 mem=ne_region
 mem=se_region
 mem=central_region;

cluster list ussales out=mylib.outfile /VERBOSE;
MINMAXVARLIST COUNT=1
varname=store_id
Numeric type.

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

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

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

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

Here is the output:

The SAS System

The SPDO Procedure

Cluster List

Cluster Name	Member Name	Variable Name	Minimum Value	Maximum Value
USSALES	NE_REGION	STORE_ID	1	20
USSALES	SE_REGION	STORE_ID	60	70
USSALES	CENTRAL_REGION	STORE_ID	60	70

The following code uses the MEMNUM= table option with a number:

```
proc print data=mylib.ussales (MEMNUM=2);
 run;
```

The following code uses the MEMNUM= table option to query the last member table in the dynamic cluster table ussales:

```
proc print data=mylib.ussales
  (MEMNUM=LASTCLUSTERMEMBER);
   run;
```

Comprehensive Dynamic Cluster Table Examples

Example 1: Create a Dynamic Cluster Table

The following example creates a dynamic cluster table named Sales_History. The example assumes the existence of 12 server tables for monthly 2014 sales and 6 tables for monthly 2015 sales in a domain pointed to by the libref MyLib. The tables are named sales201401, sales201402, sales201403, sales201404, sales201405, sales201406, sales201407, sales201408, sales201409, sales201410, sales201411, sales201412, sales201501, sales201502, sales201503, sales201504, sales201505, and sales201506. The tables are indexed.

```
/* declare main columns */
%let host=kaboom;
%let port=5400;
%let spdssize=256M;
%let spdsiasy=YES;

libname mylib sasspds "sales"
&host..&port
user='anonymous'
ip=YES;
/* Use PROC SPDO to create the dynamic cluster */
/* table sales_history                         */
PROC SPDO library=mylib; ;
   cluster create sales_history
      mem=sales201401
```

```
      mem=sales201402
      mem=sales201403
      mem=sales201404
      mem=sales201405
      mem=sales201406
      mem=sales201407
      mem=sales201408
      mem=sales201409
      mem=sales201410
      mem=sales201411
      mem=sales201412
   quit;
```

Example 2: Add Tables to a Dynamic Cluster

The following example adds member tables to the dynamic cluster table, Sales_History, that was created in "Example 1: Create a Dynamic Cluster Table" on page 46. The Sales_History table currently contains 12 members. This example augments the 12 member tables for 2014 with six new member tables that contain sales data for January through June of 2015.

```
/* declare main vars */
%let host=kaboom;
%let port=5400;
%let spdssize=256M;
%let spdsiasy=YES;

libname  mylib sasspds "sales"
   &host..&port;
   user='anonymous'
   ip=YES;

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

PROC SPDO library=mylib;
   cluster add sales_history
   mem=sales201501
   mem=sales201502
   mem=sales201503
   mem=sales201504
   mem=sales201505
   mem=sales201506;
quit;

/* Verify the presence of the added tables */
proc contents data=mylib.sales_history;
run;
```

Example 3: Refresh Dynamic Cluster Table with CLUSTER REPLACE

This example performs a refresh of the dynamic cluster table Sales_History by using the PROC SPDO CLUSTER REPLACE statement. The CLUSTER REPLACE statement

enables you to refresh one member table in a dynamic cluster without interrupting continuous cluster operations by undoing and re-creating the cluster.

Note: A member cannot be replaced if the cluster was created with UNIQUEINDEX=YES.

```
/* declare main vars */
%let host=kaboom;
%let port=5400;
%let spdssize=256M;
%let spdsiasy=YES;

libname mylib sasspds "sales"
   &host..&port
   user='anonymous'
   IP=YES ;

/* Use PROC SPDO to refresh the member tables */
/* in the dynamic cluster table Sales_History */
/* by replacing the member from December 2014 */
/* with a member from January 2015.           */

PROC SPDO library=mylib;
   cluster replace sales_history
      oldmem=sales201412 newmem=sales201501;
   quit;

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

proc contents data=mylib.sales_history;
run;
```

Example 4: Refresh Dynamic Cluster Table with CLUSTER REMOVE and CLUSTER ADD

This example performs a refresh of the dynamic cluster table Sales_History by using the PROC SPDO CLUSTER REMOVE and CLUSTER ADD statement set. The CLUSTER REMOVE and CLUSTER ADD statement set enables you to refresh one or more member tables in a dynamic cluster without interrupting continuous cluster operations by undoing and re-creating the cluster.

```
/* declare main vars */
%let host=kaboom;
%let port=5400;
%let spdssize=256M;
%let spdsiasy=YES;

libname mylib sasspds "sales"
   &host..&port
   user='anonymous'
   IP=YES ;

/* Use PROC SPDO to refresh the member tables */
/* in the dynamic cluster table Sales_History */
/* by replacing the members from July 2014 to */
```

```
/* December 2014 with members from January    */
/* 2015 to June 2015.                         */

PROC SPDO library=mylib;
   cluster remove sales_history
   mem=sales201407
   mem=sales201408
   mem=sales201409
   mem=sales201410
   mem=sales201411
   mem=sales201412';

  cluster add sales_history
   newmem=sales201501
   newmem=sales201502
   newmem=sales201503
   newmem=sales201504
   newmem=sales201505
   newmem=sales201506;
  quit;

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

proc contents data=mylib.sales_history;
run;
```

Example 5: Undo and Refresh Dynamic Cluster Table

This example uses an older server method to refresh a dynamic cluster table by unbinding the cluster, changing the member tables, and then re-binding the cluster. This method remains functional. Most users will find that the newer server statements (CLUSTER REMOVE, CLUSTER ADD, and CLUSTER REPLACE) produce identical results without requiring the dynamic cluster to be disassembled.

This example illustrates use of the CLUSTER UNDO and CLUSTER CREATE statements to refresh dynamic cluster table Sales_History. First, the 18-member dynamic cluster table Sales_History is unbound. The 12 member tables that contain 2014 sales data are omitted when the dynamic cluster table Sales_History is re-created. When the table is re-created, only the six member tables that contain 2015 sales data are included. These combined actions refresh the contents of the dynamic cluster table Sales_History.

```
/* declare main vars */
%let host=kaboom;
%let port=5400;
%let spdssize=256M;
%let spdsiasy=YES;

libname mylib sasspds "sales"
   &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 2015 only */
```

```
PROC SPDO library=mylib;
   cluster undo sales_history;
   cluster create sales_history
      mem=sales201501
      mem=sales201502
      mem=sales201503
      mem=sales201504
      mem=sales201505
      mem=sales201506;
  quit;

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

proc contents data=mylib.sales_history;
run;
```

Member Table Requirements for Creating Dynamic Cluster Tables

Overview

To create dynamic cluster tables in the server, the tables must have matching table, column, and index attributes. If there are attribute mismatches in table, column, or index attributes, the dynamic cluster table is not created, and the 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 server log. The server log lists which attribute is mismatched in the member table.

All of the member table attributes that are described in the following topics must match in order for the server to successfully create a dynamic cluster table.

Table Attributes

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

CONSTRAINT
: WHERE constraint

DISKCOMP
: compression algorithm

DSORG
: table organization

DS_ROLE
: table option for ROLE

DSTYPE
: SAS data-set type

ENCODING_CEI
encoding CEI for NLS (for compressed tables)

FLAGS
- compressed table
- encrypted table
- backup table
- NLS columns in table
- MINMAXVARLIST columns in table
- SAS encryption password in table

IOBLOCKSIZE
I/O block size

IOBLOCKFACTOR
I/O block factor

LANG
data set language tag

LTYPE
data set language type tag

NINDEXES
number of indexes

NVAR
number of columns

OBSLEN
row length

SASPW
SAS encryption password

SEMTYPE
data set semantic type

Column Attributes

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

NAME
column name

LABEL
column label

NFORM
column format

NIFORM
column informat

NPOS
column offset in row

NVARO
column number in row

NLNG
column length

NPREC
column precision

FLAGS
- NLS encoding supported
- MINMAXVARLIST column

NFL
format length

NFD
format decimal places

NIFL
informat length

NIFD
informat precision

NSCALE
scale for fixed-point decimal

NTATTR
column type attributes

TYPE
column type

SUBTYPE
column subtype

Index Attributes

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

NAME
index name

KEYFLAGS
- unique index
- nomiss index

LENGTH
index length

NVAR
number of columns in index

NVAR0
column number in index

Optimizing Dynamic Cluster Tables

Dynamic Cluster BY Clause Optimization

When you use server dynamic clusters, you can create huge tables. If a SAS job needs to manipulate a huge table, you can sort the tables for more efficient processing. Traditional processing of huge tables can overuse or overwhelm available resources. Insufficient run-time or processor resources can prohibit you from running full table scans and manipulating table rows, which are required to sort huge tables for subsequent processing.

The server provides dynamic cluster BY clause optimization to reduce the need for a large amount of processor resources when evaluating BY clauses. Dynamic cluster BY clause optimization uses the server to join individually created server member tables so that the tables appear to be a single table. But the individual member tables are also kept intact. Dynamic cluster BY clause optimization uses the SORT attribute of the member tables to bypass most of the sorting that is required to perform the implicit BY clause ordering. The server uses the SORT attribute of each member table to merge the member tables in the dynamic cluster in order by each member table's order. No additional server workspace is required, and the ordered table rows are returned quickly because the member tables do not need to be sorted.

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

You run a BY clause that matches the SORT attribute column order of the member tables of the dynamic cluster table. The server processes the BY clause without using sort workspace and does not experience first-row latency. The server uses the presorted member tables to perform an instantaneous interleave. Because dynamic cluster BY clause optimization uses the presorted member tables, you can perform operations on huge tables that would be impossible to handle otherwise.

For example, suppose your system has sufficient CPU, memory, and workspace resources to sort a 50 GB table in a reasonable amount of time. However, suppose this system accumulates 50 GB of new data every month. After 12 months, the table requires 600 GB of storage. The system cannot sort 600 GB of data to process queries that are based on the previous 12-month period. To use dynamic cluster BY clause optimization in this situation:

1. Create a dynamic cluster table from the twelve 50 GB member tables. You have a 600 GB dynamic cluster table.
2. Store data for each successive month in a server member table.
3. Sort each table and add it to the 600 GB dynamic cluster table.
4. Use dynamic cluster BY clause optimization to run SAS steps that use BY clauses on the 600 GB dynamic cluster table.

   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. The operation completes in the time that it takes to interleave the individual member

tables. The process uses no server workspace and does not cause any implicit BY sort delays.

Dynamic cluster BY clause optimization is triggered when all member tables have an applicable SORT attribute ordering for the BY clause that is asserted. When the SORT attribute ordering is strong (validated), the server does not verify the order of BY columns that are returned from the member table. When the SORT attribute ordering is weak (such as from a SORTEDBY assertion in a data set option), the server verifies the order of BY columns that are returned from the member table. If the server detects an invalid BY column order, it terminates the BY clause and displays the following error message:

```
ERROR: Clustered BY member violates weaksort order during merge.
```

Combining WHERE Clauses with Dynamic Cluster BY Clause Optimization

You can use dynamic cluster BY clause optimization to combine BY clause optimization with certain WHERE clauses on dynamic cluster tables. The 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 to be true for every row in the member table. To be trivially false, a WHERE clause must find the clause condition to be false for every row in the member table.

The server keeps metadata about indexed values that are in dynamic cluster table member tables. The server can determine whether the WHERE clause criteria are true or false, based on the dynamic cluster table's member table metadata. 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, and 4=OCT-DEC). Suppose 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, the server uses BY clause optimization to determine that the WHERE clause is trivially true. The server evaluates the expression only on the member tables for April, May, and June, and does not use any server workspace. The WHERE clause is determined to be trivially true or trivially false for each member table of the dynamic cluster table in advance. Then BY clause optimization performs BY processing only on the appropriate member tables.

Dynamic Cluster BY Clause Optimization Example

Consider a database of medical patient insurance claims that contains quarterly claims tables that are named ClaimsQ1 and ClaimsQ2. The following code does these tasks:

1. Sorts each quarterly claims table into columns that are named PatID (for patient ID) and ClaimID (for claim ID).
2. Combines the member tables into a dynamic cluster table that is named ClaimsAll.

```
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;
cluster create ClaimsAll;
quit;
```

The following DATA step MERGE statement is 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 was not a dynamic cluster table, the DATA step MERGE statement would create an implicit sort from the BY clause on the respective server tables. However, ClaimsAll is a dynamic cluster table with member tables that are presorted. As a result, dynamic cluster BY clause optimization uses BY clause processing to merge the sorted member tables instantaneously without using any server workspace or creating any delays. The example merges the transaction data named NewOnes into new rows that are appended to the data for the next quarter.

The member tables 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 table ClaimsQ1 because the WHERE clause is true for all dates in the first quarter. The WHERE clause is trivially false for the table ClaimsQ2 because the WHERE clause is false for all dates in the second quarter. 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. BY clause optimization skips the 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. They do not match 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, the server does not use BY clause optimization when it runs the code.

Unsupported Features in Dynamic Cluster Tables

Because of differences in the load and read structures for dynamic cluster tables, the following standard features that are available in SAS tables and server tables are currently not supported in SPD Server 5.2:

- You cannot directly append or update data in a dynamic cluster table. To append a new member table to a dynamic cluster table, create the new member table with the data to append. Then use the CLUSTER ADD statement to add the new member to the table.
- To update an individual member table in a dynamic cluster table, create the new member table with the data to append. Then use the CLUSTER REPLACE statement to replace the old member table with the new one. For more information, see “Example of Refreshing a Dynamic Cluster Table with CLUSTER REPLACE ” on page 42.
- To refresh a dynamic cluster table by removing numerous old member tables and replacing them with new member tables, create the new member tables with the data to append. Then use the CLUSTER ADD and CLUSTER REMOVE statements to first remove and then replace the old member tables with new ones. For more information, see “Example of Refreshing Dynamic Cluster Tables with CLUSTER REMOVE and CLUSTER ADD” on page 42.
- You can still use classic server PROC SPDO CLUSTER UNDO and CLUSTER CREATE statements to refresh the member tables in a dynamic cluster table by unbinding the dynamic cluster table and then re-binding it using new member tables. This process temporarily makes the cluster table unavailable to other users, unlike the process used by the CLUSTER REPLACE and CLUSTER REMOVE / ADD statements.
- Record-level locking is not allowed.
- The server backup and restore utilities are not available.

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

Chapter 7

Creating and Using Server Views

Overview of Server SQL Views

The server supports the creation of SQL views. A view is a virtual table that is based on the result set of an SQL statement. A server view can reference only server tables. You must use SQL explicit pass-through syntax to create server views:

```
execute(
    create view <viewname>
    as <SELECT-statement>)
BY [sasspds|alias];
```

When you create an SQL view, a view file is created in the specified domain with the name <*viewname*>.view.0.0.0.spds9. After you create an SQL view, you can use the server view as a table in server SQL queries.

The server also supports creation of materialized views. In a materialized view, the results of the view statement are computed and saved in a temporary server table when the view is created. When the view is referenced, the contents of the temporary table are delivered, unless the underlying data is found to be changed. Then, the view functions like a regular view and automatically re-creates the temporary table. Materialized server views take a longer time to create than regular views, but they display results much more quickly than regular views for data that doesn't change frequently.

View Access Inheritance

The server uses View access inheritance to control access to tables that are referenced by the server views. View access inheritance gives a user who has access to a View access to the individual component tables that make up the view. For example, user Stan creates tables WinterSales and SpringSales, and then Stan creates a view that joins the two tables. Stan creates an ACL that gives user Kyle Read access to the view. 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 / Kyle=(y,n,n,n);
quit;
```

Server View access inheritance is available only when it is invoked with the SQL explicit pass-through syntax. If a user accesses a view directly through SAS SQL or a

SAS DATA step, the user must also have direct access to the component tables that are referenced in the view. In this case, the ACL credentials of the user are applied to the component view tables. This restriction limits the usefulness of the server views that are accessed via SAS SQL to cases where a SAS SQL user creates a virtual table to simplify SQL coding.

The server SQL views that reference DICTIONARY tables cannot be used by SAS SQL.

Materialized Views

Overview of Materialized Views

For a standard SQL view, the results are computed each time the view is referenced in a subsequent SQL statement. For a materialized view, the results of the view statement are computed and saved in a temporary server table when the view is created. As long as the input tables that the view consists of are not changed, 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 make up the view are modified, the materialized view recomputes the results the next time the view is referenced and refreshes the temporary table with the new results. The temporary results table for a materialized view exists for as long as the view exists. When a user deletes or drops a materialized view, the temporary results table is deleted as well.

You must use the explicit SQL pass-through facility to create a materialized view. Specify the keyword MATERIALIZED in the CREATE VIEW syntax to identify the view as a materialized view. When you create a materialized view, the CREATE VIEW operation does not complete until the temporary results table is populated. This process can add substantial time to the execution of a CREATE VIEW statement.

Each time you reference a materialized view in an SQL statement, a check determines whether any of the input tables that are used to produce the temporary results table have been modified. If none of the tables have been modified, the temporary table is substituted in place of the view file in the SQL statement. If any of the input tables have been modified, the SQL statement executes and uses the changed tables. The statement functions like a standard SQL view reference. A background thread is also launched. The background thread is independent of the SQL statement execution. This thread refreshes the temporary results table. Until the refresh is completed, any incoming references to the view are treated as standard view references.

When you create a materialized view, an additional server table is created in the same domain as a standard SQL view file.

You cannot view or access the materialized view table 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.

Create a Materialized View

To create a materialized view, use the following SQL explicit pass-through syntax.

```
execute (
  create materialized view <viewname>
```

```
  as <SELECT-statements> )
by [sasspds | alias];
```

The MATERIALIZED keyword is necessary only in the CREATE VIEW statement. For all other references, use only the view name to reference the materialized view. The same is true in the DROP VIEW statement. For example, to drop a materialized view, use the following syntax.

```
EXECUTE (Drop View <viewname> ) BY [sasspds | alias];
```

Benefits of Materialized Views

A materialized 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 that use 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 fewer, a query that references a materialized view takes seconds to execute. If you create a standard view, every time the view is referenced results in 20 minutes of execution time. You should measure the performance benefits on a case-by-case basis.

You can base your decision of whether to use a standard view or a materialized view on 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 before any updates occur, then the materialized view should provide superior performance. In cases when you can create the defined view quickly, you probably do not need a materialized view. If the input tables are frequently updated in comparison to how often the view is referenced, a standard view is probably more efficient.

Accessing Materialized Views

You must query or access a server materialized view though an SQL explicit pass-through connection. Attempts to access the server materialized views via native SAS will result in an error.

The example statements below illustrate how to access a server materialized view:

```
select *
  from connection
  to sasspds
   (select .... from <viewname> ...);
```

or

```
execute(create table <tablename>
  as select ...
  from <viewname> ...
  by [sasspds or <àlias>] );
```

Materialized View Example

The following code creates and uses a materialized view. The code creates the input tables X and Z. Table X has three columns (a,b,c), and table Z has four columns (a,b,c,d).

```
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);
disconnect from sasspds;
quit;
```

Part 3

SPD Server SQL Processor

Chapter 8

Understanding the SPD Server SQL Processor

SPD Server Supported SQL

The SPD Server SQL processor supports all of the SQL statements that the SAS SQL procedure supports (except CREATE VIEW) when no SQL pass-through is used or when implicit SQL pass-through is used. For information about these statements, see *SAS SQL Procedure User's Guide*.

For explicit SQL pass-through, the server SQL processor supports the SAS SQL statements, with some modifications. In addition, the SQL processor supports the following additional SQL statements:

BEGIN ASYNC OPERATION and END ASYNC OPERATION
: enable you to send blocks of SQL statements to the server asynchronously, so that they can be processed in parallel.

CREATE [MATERIALIZED] VIEW
: creates a regular or materialized view of SPD Server tables.

COPY TABLE
: copies an SPD Server table, with or without indexes.

LIBREF
: defines server domains from within an EXECUTE statement without your needing to reissue PROC SQL.

LOAD TABLE
: creates a new SPD Server table from an existing SPD Server table, with or without indexes

RESET
: enables you to customize SPD Server SQL Planner settings.

The SPD Server SQL Planner provides the following SQL processing:

- parallel WHERE clause processing
- parallel GROUP BY processing
- BY data grouping with ORDER BY
- parallel index creation.

For both implicit and explicit pass-through, there are some differences in SPD Server SQL versus SAS SQL.

For more information about the additional SQL statements for explicit pass-through, see Chapter 19, "SPD Server SQL Statement Additions," on page 183.

Understanding the Server's SQL Pass-Through

SQL pass-through functionality provides the ability to execute as many queries and perform as many calculations as possible by the server SQL processor rather than by the SAS client. Processing within the server is faster than passing data back and forth between the client and the server for processing. The server's SQL pass-through facility passes SQL code to the server for processing either implicitly or explicitly based on how you connect to the server.

SQL Explicit Pass-Through

An SQL explicit pass-through connection is a connection to the server using the CONNECT statement from PROC SQL or another SQL-aware procedure.

You specify server SQL statements in a PROC SQL EXECUTE statement or in a subsequent SELECT * FROM CONNECTION statement. When you use an explicit SQL pass-through connection, all tables that are referenced in the SQL statement must be server tables or an error occurs. The server SQL engine must be able to successfully parse the submitted SQL statement. If the server cannot successfully parse the statement, the request fails.

The SQL code that you submit with an explicit SQL connection is passed exactly as written to the server's SQL processor. Use SQL explicit pass-through when you want to optimize the SQL yourself, or when you want to control exactly which commands are sent to the server's SQL processor.

SQL explicit pass-through sends SQL code very efficiently, because there is no automatic translation of your SQL code. However, there is no optimization done to improve performance of a query. You must take advantage of the optimization features of the server to ensure that the code performs as efficiently as possible (for example, parallel GROUP BY processing or BY data grouping). For information about optimizing explicit SQL, see Chapter 9, "SQL Planner Options," on page 77.

SQL Implicit Pass-Through

An SQL implicit pass-through connection is a connection to the server using a SASSPDS LIBNAME statement with the option IP=YES. IP=YES invokes the SQL implicit pass-through facility. With this connection, the client generates automatically optimized server SQL code when you call the SAS SQL procedure. The generated SQL is automatically optimized and then "passed through" to the server's SQL processor. For an example of an implicit pass-through request, see "Create a Table with PROC SQL" on page 26.

When you use an SQL implicit pass-through connection, the SAS SQL Planner parses SQL statements to determine which, if any, portions can be passed to the server SQL engine. In order for a submitted SQL statement to take advantage of SQL implicit pass-through SQL, the tables that are referenced in the SQL statement must be server tables, and the server SQL engine must be able to successfully parse the submitted SQL statement. If the server cannot successfully parse the statement, SAS SQL retries the query on the client.

Logging or Suppressing Errors When Submitting SQL Implicit Pass-Through SQL Code

If the server cannot process the SQL implicit pass-through query that is submitted through SAS PROC SQL, PROC SQL simplifies the query and iteratively retries the simplified query until it succeeds.

By default, the server does not report in the SAS log implicit pass-through queries that could not be handled by the server's SQL processor. These are reported in the server log.

To turn on the server SQL implicit pass-through error reporting in the SAS log, set the SPDSIPDB macro variable (**`%let SPDSIPDB=YES;`**). An SQL pass-through failure is recorded in the SAS log as a **`Note`**, not as an **`Error`**.

Differences between SAS SQL and SPD Server SQL

Reserved Keywords

SPD Server uses keywords to initiate statements or to refer to syntax elements. In SPD Server SQL, keywords are treated as reserved words. For example, you can use the words "where" and "group" only in certain ways because SPD Server uses WHERE and GROUP BY clauses. You cannot use them for identifiers because this use introduces ambiguity. For example, **`select count(*) from sales.receipts where table='April';`** is a valid but ambiguous statement.

If you use a keyword as an identifier and the request is submitted via explicit SQL pass-through, the server will return an error and the request will fail. If you use a keyword as an identifier in an implicit SQL pass-through request, the request will fail and will not return an error. You must set **`%let SPDSIPDB=YES`** to see the failure message.

For both implicit and explicit SQL pass-through failures, the location of the failure is indicated in the failure message with a # (number sign). For example, in the following message, the # indicates a parsing error on the keyword MATCH:

```
SPDS_NOTE:  Parse Failure:  select TXT_3.Messages from #MATCH TXT_1 inner join ...
```

In contrast, SAS SQL allows keywords in some, but not all, syntax locations. SAS SQL also uses underscores to denote errors.

The following list contains current SPD Server keywords. Some of the words are reserved for future enhancements to SPD Server SQL.

add	date	grant	missing	select
all	dec	group	modify	set
alter	decimal	gt	natural	smallint
and	default	having	ne	some
any	delete	in	no	table
as	desc	index	not	then
asc	describe	indexes	notin	to
async	dictionary	informat	null	trailing
begin	disconnect	inner	num	trim
between	distinct	insert	numeric	true
both	double	int	on	union
by	drop	integer	operation	unique
calculated	else	intersect	option	unknown
cascade	end	into	or	update
case	engname	is	order	upper
char	engopt	join	outer	using
character	eq	label	overlaps	validate
column	except	le	partial	values
connect	execute	leading	precision	varchar
connection	exists	left	privileges	verbose
contains	false	lib	public	view
contents	float	libref	real	when
copy	for	like	references	where
corr	format	load	reset	with
corresponding	from	lower	restrict	without
create	full	lt	revoke	yes
cross	ge	match	right	

Table Options and Delimiters

SPD Server explicit SQL pass-through uses brackets to delimit table options. SAS SQL uses parentheses as delimiters.

Here is an example of how a table option is specified for CREATE TABLE in explicit SQL pass-through:

```
execute(
create table spptbv05x[netpacksize=12288](
    z char(5),
    x num,
    y numeric)
) by sasspds;
```

Here is an example of how to specify a table option in an INSERT statement:

```
execute(
  insert into spptbv05x [syncadd=yes]
     values("one",1,50)
     values("two",2,30)
     values("three",3,30)
     values("four",4,60)
     values("five",5,70)
     values("six",6,80)
) by sasspds;
```

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.

Scalar expressions represent a single data value, either a numeric value or a string from a constant specification. Examples include the following:

- 1
- 'hello there'
- '31-DEC-60'd
- a function: for example: avg(a*b)
- a column name: for example: sales.product_id
- the CASE expression
- a subquery that returns a single run-time value

.

Boolean predicates are either true or false. They are used in WHERE clauses, in HAVING clauses, and in the CASE expression. They cannot be used in SELECT clauses or assigned to columns in an UPDATE statement. Mixing scalar expressions and Boolean predicates result in errors. Here is an example:

```
select * from connection to sasspds
   (select * from sales where x=1 and 10);
```

This example produces this error:

```
 SPDS_ERROR: Parse Failure: select * from x where x=1 and 10#;
```

The # (number sign) indicates where the parsing error occurred.

INTO Clause

SPD Server SQL does not support the INTO clause. For example, SPD Server SQL does not support the following statement:

```
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 character (~) only to negate the equals operator (=), as in ~= (not equals). SAS SQL supports the 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 the BETWEEN operator, as in ~BETWEEN (not between). SPD Server does not recognize this expression.

Nested Queries

SAS SQL permits subqueries without delimiting parentheses in more places than does SPD Server SQL. SPD Server SQL uses parentheses to explicitly group subqueries or expressions that are nested in a query statement whenever possible. Queries with nested expressions execute more reliably and are also easier to read.

USER Value

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 most of the functions that SAS supports. For more information, see *SAS Functions and CALL Routines: Reference*.

SPD Server SQL Dictionary Tables

The server provides dictionary tables that enable you to get metadata information about user objects such as tables, columns, indexes, and ACLs. SAS also has dictionary tables, but the SAS dictionary tables and the server dictionary tables are different and cannot be used interchangeably. The server uses some dictionary tables that SAS does not support, such as DICTIONARY.ACLS, DICTIONARY.PWDB, and DICTIONARY.CLUSTERS.

The following dictionary tables are available to a server user with explicit SQL pass-through:

- DICTIONARY.MEMBERS– Use DICTIONARY.MEMBERS to list SPD Server resources in the domains that are available using SQL. These include data tables, views, and cluster tables.
- DICTIONARY.TABLES– Use DICTIONARY.TABLES to get information about data tables in the domain.
- DICTIONARY.COLUMNS– Use DICTIONARY.COLUMNS to get information about columns for data tables in the domain.
- DICTIONARY.INDEXES– Use DICTIONARY.INDEXES to get information about indexes for data tables in the domain.
- DICTIONARY.VIEWS– Use DICTIONARY.VIEWS to get information about views in the domain.

- DICTIONARY.ACLS– Use DICTIONARY.ACLS to get information about ACLs.
- DICTIONARY.PWDB– Use DICTIONARY.PWDB to get information about server users.
- DICTIONARY.CLUSTERS– Use DICTIONARY.CLUSTERS to get information about cluster tables.
- DICTIONARY.SYSINFO– Use DICTIONARY.SYSINFO to get system information.

Use the DESCRIBE TABLE statement to print information about the dictionary tables. The DESCRIBE TABLE output is written to the SAS log. Here is a sample output for each dictionary table:

```
 execute (describe table dictionary.members)    by sasspds;

SPDS_NOTE: SQL table DICTIONARY.members was created like:


create table DICTIONARY.members
 (
  LIBNAME char(8) label='Library Name',
  MEMNAME char(32) label='Member Name',
  MEMTYPE char(8) label='Member Type',
  ENGINE char(8) label='Engine Name',
  INDEX char(32) label='Indexes',
  PATH char(1024) label='Pathname'  );

execute (describe table dictionary.tables)
  by sasspds;

SPDS_NOTE: SQL table DICTIONARY.tables was created like:
create table DICTIONARY.tables
  (
   LIBNAME char(8) label='Library Name',
   MEMNAME char(32) label='Member Name',
   MEMTYPE char(8) label='Member Type',
   MEMLABEL char(256) label='Data Set Label',
   TYPEMEM char(8) label='Data Set Type',
   CRDATE num format=DATETIME informat=DATETIME label='Date Created',
   MODATE num format=DATETIME informat=DATETIME label='Date Modified',
   NOBS num label='Number of rows',
   OBSLEN num label='row Length',
   NVAR num label='Number of Variables',
   PROTECT char(3) label='Type of Password Protection',
   COMPRESS char(8) label='Compression Routine',
   REUSE char(3) label='Reuse Space',
   BUFSIZE num label='Bufsize',
   DELOBS num label='Number of Deleted rows',
   INDXTYPE char(9) label='Type of Indexes',
   LOCALE char(32) label='Locale',
   ENCODING num label='Encoding_Cei'  );

execute (describe table dictionary.columns) by sasspds;

SPDS_NOTE: SQL table DICTIONARY.columns was created like:
create table DICTIONARY.columns
  (
```

```
   LIBNAME char(8) label='Library Name',
   MEMNAME char(32) label='Member Name',
   MEMTYPE char(8) label='Member Type',
   NAME char(32) label='Column Name',
   TYPE char(4) label='Column Type',
   LENGTH num label='Column Length',
   NPOS num label='Column Position',
   VARNUM num label='Column Number in Table',
   LABEL char(256) label='Column Label',
   FORMAT char(16) label='Column Format',
   INFORMAT char(16) label='Column Informat',
   IDXUSAGE char(9) label='Column Index Type'  );

execute(describe table dictionary.indexes) by sasspds;

SPDS_NOTE: SQL table DICTIONARY.indexes was created like:
create table DICTIONARY.indexes
   (
    LIBNAME char(8) label='Library Name',
    MEMNAME char(32) label='Member Name',
    MEMTYPE char(8) label='Member Type',
    NAME char(32) label='Column Name',
    IDXUSAGE char(9) label='Column Index Type',
    INDXNAME char(32) label='Index Name',
    INDXPOS num label='Position of Column in Concatenated Key',
    NOMISS char(3) label='Nomiss Option',
    UNIQUE1 char(3) label='Unique Option'  );
execute (describe table dictionary.views) by sasspds;

SPDS_NOTE: SQL table DICTIONARY.views was created like:
create table DICTIONARY.views
  (
   LIBNAME char(8) label='Library Name',
   MEMNAME char(32) label='Member Name',
   MEMTYPE char(8) label='Member Type',
   ENGINE char(8) label='Engine Name'  );

execute (describe table dictionary.acls) by sasspds;

SPDS_NOTE: SQL table DICTIONARY.acls was created like:
create table DICTIONARY.acls
  (
   LIBNAME char(8) label='Library Name',
   MEMNAME char(32) label='Member Name',
   MEMTYPE char(8) label='Member Type',
   NAME char(32) label='Column Name',
   OWNER char(8) label='Owner',
   GROUP char(8) label='Group',
   DEFACS char(56) label='Default Access',
   GRPACS char(56) label='Group Access'  );

execute (describe table dictionary.pwdb) by sasspds;

SPDS_NOTE: SQL table DICTIONARY.pwdb was created like:
create table DICTIONARY.pwdb
  (
```

```
   USER char(8) label='User',
   AUTH_LVL char(5) label='Authorization Level',
   IP_ADDR char(16) label='IP Address',
   DEFGRP char(8) label='Default Group',
   OTHGRPS char(224) label='Other Groups',
   EXPIRE char(6) label='Expire Period',
   MOD_DATE char(32) label='Password Last Modified',
   LOG_DATE char(32) label='Last Login',
   TIMEOUT char(8) label='Timeout Period',
   ALLOWED char(10) label='Failed Login Attempts Allowed',
   STRIKES char(6) label='Failed Login Attempts'  );

execute (describe table dictionary.clusters) by sasspds;

SPDS_NOTE: SQL table DICTIONARY.clusters was created like:
create table DICTIONARY.clusters
  (
   LIBNAME char(8) label='Library Name',
   CLSTNAME char(32) label='Cluster Name',
   TYPE char(5) label='Cluster Type',
   MBRNAME char(32) label='Cluster Member',
   );

execute (describe table dictionary.sysinfo) by sasspds;

SPDS_NOTE: SQL table DICTIONARY.sysinfo was created like:
create table DICTIONARY.sysinfo
  (
   SYS_ENC char(32) label='Server Host Default Encoding',
   SYS_OS char(64) label='Server Operating System',
   SYS_NAME char(32) label='Server Host Name',
   SYS_SPDS char(32) label='SPDS Version Number'  );
```

Use the SELECT statement to return dictionary information about actual resources in a domain. SELECT outputs are written to the SAS Output window using the SAS Output Delivery System. For example, to list the SPD Server resources in the domain Test, submit this SELECT statement:

```
select * from connection to sasspds
(select * from dictionary.members);
```

The server returns output similar to the following:

The SAS System

Library Name	Member Name	Member Type	Engine Name	Indexes	Pathname
X0000001	X	DATA	SPDSENG	no	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	Z	DATA	SPDSENG	no	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	CARS	DATA	SPDSENG	no	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	CARS2	DATA	SPDSENG	no	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	CARS3	DATA	SPDSENG	yes	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	XZVIEW	VIEW	SPDSENG	no	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	CARLOAD	DATA	SPDSENG	yes	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	AUDICARS	DATA	SPDSENG	yes	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	COPYCARS	DATA	SPDSENG	no	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	ENCTABLE	DATA	SPDSENG	no	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	FORDCARS	DATA	SPDSENG	yes	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/
X0000001	REGTABLE	DATA	SPDSENG	no	:13874/bigdisk/lax/pubs_d4spds53/test_domains/lax94d01/coretst4/

To get information about a specific data table, submit a SELECT statement similar to the following:

```
select * from connection to sasspds
(select * from dictionary.tables where memname='AUDICARS');
```

The server returns output similar to the following:

The SAS System

Library Name	Member Name	Member Type	Data Set Label	Data Set Type	Date Created	Date Modified	Number of Observations	Observation Length	Number of Variables	Type of Password Protection	Compression Routine	Reuse Space	Bufsize	Numb Del Observat
X0000001	AUDICARS	DATA			27JAN16:14:18:55	27JAN16:14:18:55	19	83	6	---	NO	no	65487	

Part 4

Optimizing SPD Server Queries

Chapter 9

SQL Planner Options

Overview of SQL Planner Options

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

Specifying SQL Planner Options

Specify SQL Options by Using Explicit Pass-Through Code

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

Most usage examples of the SQL Planner reset option in this document use SQL explicit pass-through code.

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

proc sql;
connect to sasspds (
  dbq='MyDomain'
  host='husky'
  service='5600'
  user='siteusr1'
  password='passwrd') ;

execute(reset PRINTLOG)
  by sasspds ;

execute(SQL statements)
  by sasspds ;

disconnect from sasspds ;
 quit ;
```

The example includes two EXECUTE statements. The first EXECUTE statement specifies the reset option. The SQL statements to which the reset option applies are specified in the second EXECUTE statement.

Specify SQL Options for SQL Implicit Pass-Through Code

For SQL implicit pass-through requests, you declare SQL reset options by using the SPDSSQLR macro variable. The following example shows how to use the macro variable.

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

%let spdssqlr= INOBS=1000 ;
libname spdslib sasspds 'mydomain' host="husky" service="5600" user="siteusr1"
 password="passwd";

proc sql ;
...SQL statements... ;
quit ;
```

You submit your PROC SQL request as usual. The macro variable simply alters the context in which the request is executed.For more information about this macro variable, see "SPDSSQLR Macro Variable" on page 236.

General SQL Planner Options

_method

A method tree is produced in your output if you specify the _method reset option for the SQL Planner. You read the SQL Planner method tree from bottom row to top row. The _method reset option writes a method tree in the output that shows how the SQL was planned and 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

Example: Reading the Method Tree

The following example 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 ;
```

The following example method tree is 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
```

You can review the sequence of methods that were invoked by reading the tree from bottom to top.

```
SPDS_NOTE: SQL execution methods chosen are:
<0x00000001006BBD78> step 9
<0x00000001006BBBF8>     step 8
<0x00000001006BBB38>               step 7
<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 server query optimizer considers a number of join strategies. Some of the join strategies require memory buffers. In these cases, BUFFERSIZE= specifies the amount of memory that the server should reserve for memory buffers.

For example, the server SQL might consider a hash join when an index join is not possible. A hash join reconfigures the smaller table in memory as a hash table. SQL sequentially scans the larger table and performs a hash lookup row-by-row against the

smaller 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 64 K. You can specify the amount of memory that you want the server to use for hash joins.

Usage:

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

n
the maximum number of rows in the smaller table for a hash join that can use the inset size hash join optimization.

DETAILS=

Use the DETAILS= reset switch to provide additional information in the SAS log about the SQL joins that the server made.

Usage:

```
execute(reset
  details=("what_join$"|"why_join$"|"what_join$why_join$")
  by sasspds ;
```

DETAILS="what_join$"
adds additional information in the SAS log documenting the join plan that was selected.

DETAILS="why_join$"
adds additional information in the SAS log documenting why the join plan that was selected was chosen.

DETAILS="what_why_join$"
adds additional information in the SAS log documenting the join plan that was selected, and why the join plan that was selected was chosen.

EXEC / NOEXEC

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

Usage:

```
/* This SQL explicit Pass-Through  */
/* 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 ;
```

HASHINSETSIZE

You use the server SQL planner HASHINSETSIZE reset option to influence when the hash join inset size optimization can be used. The hash-join, inset size optimization gathers join keys from the smaller join table, and then generates a query to the larger table. The query to the larger table selects only rows that can be joined to the smaller table. The join keys for the selected rows of the larger table are then hashed with the smaller table in order to perform the hash join.

Usage:

```
execute(reset
  hashinsetsize=n)by sasspds;
```

n
the maximum number of rows in the smaller table for a hash join that can use the inset size hash join optimization.

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 in the range 0–1 that acts as a minimum threshold value for the server cardinality ratio when selecting a join method. The server cardinality 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 server cardinality ratio are continuous values between 0 and 1. The server compares the calculated cardinality ratio for a server index join to the value that you specify in the INDEXSELECTIVITY= option. If the calculated cardinality ratio is greater than or equal to the value that is specified in the INDEXSELECTIVITY= option, SPD server chooses the index join method. The default setting for the INDEXSELECTIVITY= option is 0.7.

How does the server calculate the cardinality ratio? The cardinality 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 cardinality ratio approaches 0, which indicates low cardinality, the greater the number of duplicate values that exist in the rows of the outer table. As the value of the cardinality ratio approaches 1, which indicates high cardinality, the fewer the number of duplicate index values in the rows of the outer table. For example, a cardinality ratio of 1/1, or 1, represents a unique index value for every row in the outer table, a unique index. A cardinality ratio value of 1/2, or 0.5, represents a unique index value for every two rows in the outer table. A cardinality 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, which represents 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 server cardinality ratio of 0.7 (a unique index value per 1.43 rows in the outer table). The expected result set is 100*1.43, or 143 rows.

Higher cardinality and higher index cardinality ratios are associated with an efficient index join. Cardinality ratios near 1 result in more efficient processing during probes between the outer table rows and the inner table index, because each probe has fewer rows to retrieve. In turn, the work that the server index must do to find and retrieve the matching rows during the join operation is maximized, which results 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 cardinality criteria more selective by decreasing the allowable average number of rows per probe of the inner table. Setting INDEXSELECTIVITY= equal to 1.0 allows only a join with a unique index. Setting INDEXSELECTIVITY= to a value greater than 1.0 allows no index joins. Decreasing the value of INDEXSELECTIVITY= makes the cardinality criteria more flexible 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 cardinality.

Usage:

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

INOBS

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

Usage:

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

The integer value <*n*> is the number of rows that you want to read.

JTECH PREF | JOINTECH_PREF

You use the server SQL Planner JTECH PREF | JOINTECH_PREF reset option to control how the server SQL Planner executes join statements. The option has four settings: seq, merge, hash, and index.

Usage:

```
execute(reset
  jointech_pref=<seq|merge|hash|index>)
   by sasspds ;
```

JOINTECH_PREF=seq
: The server performs sequential loop joins. Sequential loop joins are brute force joins that match every row of the first table to every row of the second table.

JOINTECH_PREF=merge
: The server performs sort merge joins. Sort merge joins force a sort on all tables that are involved in the join.

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

JOINTECH_PREF=index
The index join requires an index on the join column of one table, the indexselectivity requirement must be met (see the indexselectivity reset option), and reading the table via the index is more beneficial than doing a full table read. Preferring the index join removes the beneficial index read check.

MAXHASHJOIN

You use the server SQL planner MAXHASHJOIN reset option to control how many hash joins can be planned in a single statement.

Usage:

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

n
the number of hash joins that can be planned.

Note: Hash joins in the server can be memory intensive. Increasing the number of hash joins is likely to increase the memory requirements for the query plan.

OUTOBS

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

Usage:

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

The integer value <*n*>is the number of rows that you want to create or print.

OUTRSRTJNDX / NOOUTRSRTJNDX

Use the OUTRSRTJNDX / NOOUTRSRTJNDX option to configure the sort behavior for a server join index. OUTRSRTJNDX sorts the outer table for a join index by the join key. This setting is the default 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 server SQL Planner to turn on or off the printing of the SQL statement text to the server log.

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 a server LIBNAME. SAS PROC SQL views use a generic transport format to represent numeric values, which the server converts to native numeric values. When extremely large or extremely small numeric values are conveyed in a SAS PROC SQL view to the server, extreme values might not be as precise during the 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 the server SQL pass-through uses a view that was created by PROC SQL, the 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 the server SQL pass-through uses a view that was created by PROC SQL, the server prints the following note in the SAS log:

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

SPDSIPDB

When you use the server SQL implicit pass-through facility, the server must first parse and prepare the SQL implicit pass-through statement, and then the server must execute the SQL implicit pass-through statement. Both the Prepare and Execute operations must complete successfully in order for the SQL implicit pass-through statement to be performed.

If the server cannot execute the implicit SQL submitted to SAS PROC SQL, PROC SQL will simplify the query, and the server will iteratively retry the simplified SQL query until it succeeds. By default, when implicit PROC SQL pass-through queries to the server fail, the event is not reported in the SAS log.

To enable SQL implicit pass-through statement error reporting in the server SAS log, set the SPDSIPDB implicit SQL code reporting macro to YES. SQL implicit pass-through statement errors appear in the SAS log as a **NOTE:** entry, and not as an **ERROR:** entry.

Example

```
%let SPDSIPDB=YES;
```

If undeclared, the default setting for the SPDSIPDB macro variable is NO.

UNDO_POLICY=

Use the UNDO_POLICY option in the server PROC SQL and RESET statements to configure the 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 the 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
: the default setting for the server. This setting 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.
```

Chapter 10

Join Planner

Understanding the SPD Server Join Planner

The server Join Planner is a rules-based planner. The join planner searches for a pairwise equijoin match in a particular order. The first plan that meets requirements is selected. If the join is an *n*-way join, each pairwise join of the *n*-way join is planned until all of the joins are exhausted.

Each pairwise join follows the same selection order to determine which join plan is selected. The order of the join planner for a pairwise equijoin is as follows:

1. The server searches for an acceptable star schema optimization.
2. The server searches for an index join.
3. The server searches for a hash join.
4. The server searches for a merge join. Preferences are given to parallel merge joins.
5. The server searches for a sequential loop join.

There are several server SQL reset switches that affect the join planner:

- The server star schema optimization reset switch NOSTARJOIN disables star joins.
- The index join reset switch INDEX_SELECTIVITY can change the relative usefulness of the index for the join type. High index selectivity settings can affect whether the join planner chooses the index join.
- The hash join reset switch MAXHASHJOINS can increase or decrease the number of hash joins that can be planned for a single query. The hash join BUFFERSIZE reset switch can increase or decrease the amount of memory that is allocated for hash joins.
- The merge join reset switch NOPLLJOIN disables parallel merge joins.

You can favor a join plan by using the JOINTECH_PREF reset switch. Favoring a join plan does not guarantee that the favored join plan will be used, however. For example, if

you favor a hash join, the server still requires sufficient BUFFERSIZE memory allocation to plan the hash join.

You can use the DETAILS= "WHAT_JOINWHY_JOIN" reset switch to print additional information in the SAS log to determine which join method the SPD Join planner selected, and why it was selected. The WHY_JOIN information includes how the reset switches affected the join planner.

Join Planner Reset Option Examples

Join Planner DETAILS= Reset Switch

The following example shows use of the DETAILS reset switch on a join between two tables. In this case, table A contains an index on the join column.

```
proc sql;
connect to sasspds(
   dbq='mydomain'
   host="myhost"
   serv="14500"
   user='anonymous');

execute(reset
   details="why_join$what_join$")
by sasspds;

execute (create table
  tblout as select *
  from tablea, tableb
where
   a1 = a2)
by sasspds;

 **WHY_JOIN( 1)?: Plan an Inner Join
**WHY_JOIN( 1)?: INDEX available on 1 tables
**WHY_JOIN( 1)?: Index Join pass 1
**WHY_JOIN( 1)?: Inner table [X0000001].TABLEA Index a1
**WHY_JOIN( 1)?: Idx dup_ratio(1.00) >= indexselectivity(0.70)
**WHY_JOIN( 1)?: Est inner rows to read via idx(100.0)
**WHY_INDX( 1)?: Good dup_ratio and inner table index is beneficial
SPDS_NOTE: PROC SQL planner chooses indexed join.
SPDS_NOTE: Table X0000001.TBLOUT created, with 100 rows and 4 columns.
```

The WHAT_JOIN$ details produce the server note that reads **`PROC SQL planner chooses indexed join.`**. This note indicates that the index join was selected. The WHY_JOIN$ details provide information that shows that the join performed is an inner join. Table A has an index on column A1. The duplicate variable ratio on the index is favorable (as compared to the index selectivity). As a result, the index join is selected.

Using JOINTECH_PREF Reset Switch to Alter an Index Join to a Hash Join

The following example uses the reset switch JOINTECH_PREF to persuade the server to choose a hash join over an index join.

```
execute(reset
  details="why_join$what_join$"
  jtech_pref=hash)
by sasspds;

execute (create
  table tblout
  as select *
  from tablea, tableb
  where a1 = a2)
by sasspds;

**WHY_JOIN( 1)?: Plan a Inner Join
**WHY_NIDX( 1)?: Magic=103 (jtech_pref=hash) prohibits index
**WHY_MERG( 1)?: Index join not selected, do merge join
**WHY_JOIN( 1)?: Magic=103 (jtech_pref=hash) skips JM table order check
**WHY_HASH( 1)?: merge xformed to hash join, num_hashjoins=1
SPDS_NOTE: PROC SQL planner chooses hash join.
**WHY_HASH( 1)?: Inset optimization, hashkeys(100) le hashinsetsize(1024)
SPDS_NOTE: Table X0000007.TBLOUT created, with 100 rows and 4 columns.
```

The WHAT_JOIN$ details produce the server note **`PROC SQL planner chooses hash join`**, which indicates that the index join was selected.

The WHY_JOIN$ details indicate that the join is an inner join, and that an index join is not selected because the JTECH_PREF is set to hash. The join was successfully transformed to a hash join (implying that there was sufficient buffer size to do the hash). The hash join inset optimization was used because the number of hash keys in the smaller table (100) is less than or equal to the hash inset size limit (1024).

N-Way Join Example

When you use an *n*-way join, the server returns WHAT_JOIN$ and WHY_JOIN$ information for each pairwise join of the *n*-way join.

```
execute(reset
  details="why_join$what_join$"
  _method jointech_pref=none)
by sasspds;

execute (create table tblout
  as select *
  from tablea, tableb, tablec
  where a1 = a2
  and a2 = a3)
by sasspds;

**WHY_JOIN( 1)?: Plan a Inner Join
**WHY_NIDX( 1)?: No INDEX on join column
**WHY_MERG( 1)?: Index join not selected, do merge join
**WHY_JOIN( 2)?: Plan a Inner Join
**WHY_JOIN( 2)?: INDEX available on 1 tables
**WHY_JOIN( 2)?: Index Join pass 1
**WHY_JOIN( 2)?: Inner table [X0000010].TABLEA Index a1
**WHY_JOIN( 2)?: Idx dup_ratio(1.00) > indexselectivity(0.70)
```

```
**WHY_INDX( 2)?: Favorable inner table index dup_ratio

SPDS_NOTE: PROC SQL planner chooses indexed join.

**WHY_HASH( 1)?: merge xformed to hash join, num_hashjoins=1

SPDS_NOTE: PROC SQL planner chooses hash join.

**WHY_HASH( 1)?: Inset optimization, hashkeys(100) le hashinsetsize(1024)
```

The WHAT_JOIN$ details produce two server notes. The first note in the SAS log above reads **`PROC SQL planner chooses indexed join`**. The second note reads **`PROC SQL planner chooses hash join`**. These notes indicate that two pairwise joins were required for the query: an index join and a hash join.

The WHY_JOIN$ details show how each pairwise join was planned. The order of the join is indicated by the additional numeric values in the log. **`WHY_JOIN( 1)`** is the first pairwise join plan, and **`WHY_JOIN( 2)`** is the second pairwise join plan. It is a good idea to include the DETAILS="WHY_JOIN$_WHAT_JOIN$" switch in your reset command when you create an *n*-way join. It adds helpful information to the SAS log that enables you to easily determine which tables are involved in each pairwise join of the *n*-way join.

The _method information for the above join is as follows:

```
SPDS_NOTE: SQL execution methods chosen are:
   sqxcrta
     sqxjndx(2)
       sqxjhsh(1)
          sqxsrc ( [X0000010].TABLEB )
             sqxsrc ( [X0000010].TABLEC )
          sqxsrc ( [X0000010].TABLEA )
```

The _method information shows that TABLEB and TABLEC will be used by the sqxjhsh (hash join) method. The results of the join will be used with TABLEA for the sqxjndx (index join) method. The numeric in the join method chosen matches up with the numeric in the WHY_JOIN$ information. In other words, the sqxjhsh(1) hash join method was selected as the result of the WHY_JOIN(1) plan, and the sqxjndx(2) index join method was selected as a result of the WHY_JOIN(2) plan.

Chapter 11

Parallel Join Facility

Understanding the Parallel Join Facility

Overview of the Parallel Join Facility

The Parallel Join facility is a feature of the server SQL Planner that decreases the processing time that is required to create a pairwise join between two server tables. The savings in processing time is created when the server performs the pairwise join in parallel.

The SQL Planner first searches for pairs when the server source tables are to be joined. When the Planner finds a pair, it checks the join syntax for that pair to determine whether the syntax meets all of the requirements for the Parallel Join facility. If the join syntax meets the requirements, the pair of tables are joined by the Parallel Join facility.

Criteria for Using the Parallel Join Facility

The criteria for using the server Parallel Join facility can be more complex than simply requiring a pairwise join of two 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 a parallel join.

The parallel sort-merge method is a good, all-purpose parallel join strategy that requires no intervention from you. The tables for the sort-merge method do not need to be in the same domain. 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 that are being joined. Simultaneously, it performs 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 you are performing an outer join, left join, or right join in parallel. Parallel outer joins, left joins, or right joins can use only two concurrent threads. 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 server parallel sort then sorts the rows within a range.

You can use the parallel range join method only on tables that are in the same domain. If either of the two tables are updated after the join index is created, you must rebuild the join index before you can use the parallel range join method. 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 the server to perform increasingly larger table row scans. These larger scans can consume processor resources at a high rate during concurrent join operations. For more information about creating join indexes, see "Index Utility" in *SAS Scalable Performance Data Server: Administrator's Guide*.

How does the 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 because 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 server Parallel Join facility is its integration with the 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, the 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 server Parallel Join facility, you can use the parallel group-by method on multiple tables.

Parallel Join Reset Options

PLLJOIN / NOPLLJOIN

The PLLJOIN / NOPLLJOIN option enables and disables the server Parallel Join facility.

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

CONCURRENCY=

The CONCURRENCY=<*n*> option sets the concurrency level that the server Parallel Join facility uses. The integer value *n* specifies the number of levels. In most cases, you should not change the default server concurrency setting, which is half of the available number of processors.

Your concurrency value should not exceed 2. A concurrency of 1 or 2 for parallel merge join still provides parallelism and has been shown to give optimal performance based on benchmark testing results. A concurrency of 1 means that two threads are working in parallel. A concurrency of 2 means that three threads are working in parallel.

```
execute(reset concurrency=2)
  by sasspds ; /* enables 2 concurrency levels */
```

PLLJMAGIC

The PLLJMAGIC option specifies how SPD server performs parallel joins.

```
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 Examples

Parallel Join Example 1

The example is a basic SQL query that creates a pairwise join of two 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

This example is an SQL query that uses more than two server tables. The SQL Planner performs a parallel join on Table1 and Table2. It then use a non-parallel method to join the results of the first join and Table3. The second join uses a non-parallel join method because the criteria for a parallel join were not met. A parallel join can be performed only on a pairwise join of two server tables and the query calls three 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

You can use multiple parallel joins in the same SQL query, as long as the SQL Planner can perform the query by using more than one pairwise join. In this parallel join example, a more complex query contains a union of two separate joins. Both joins are pairwise joins of two server tables. There is a pairwise join between Table1 and Table2. A pairwise 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;
```

Chapter 12

Parallel Group-BY Facility

Understanding the Parallel Group-By Facility

Overview of the Parallel Group-By Facility

The server SQL Planner optimizations improve the performance of the more frequent query types used in data mining solutions. One of the SQL Planner optimizations is the Parallel Group-By capability. Parallel Group-By is a high-performance parallel summarization of data that is executed using SQL. Parallel Group-By is often used in SQL queries (through the use of subqueries) 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 by 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 that it can handle.

The Parallel Group-By support in the server is integrated into the WHERE clause planner code so that it boosts the capabilities of the server SQL processor. Any section of code that matches the Parallel Group-By trigger pattern will use Parallel Group-By support.

Enhanced Group-By Functions

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

Nested Queries Meet Group-By Syntax Requirements

Because the Parallel Group-By functionality is integrated into the server WHERE clause planner, many sections of queries can take advantage of performance enhancements such as parallel processing. Some common performance enhancements are subqueries that generate value lists in an IN clause, views that 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* value) or aggregate (summary) functions that involve a single column (with the exception of COUNT(*), which accepts an asterisk argument). You must specify at least one aggregate function. You can use an alias for project items (for example, **`SELECT avg(salary) AS avgsal FROM...`**). These aliases can appear in any *where-expression*, *having-expression*, *groupby-list*, or *orderby-list* value. The following aggregate functions are supported: COUNT, AVG, AVG DISTINCT, COUNT DISTINCT, CSS, MAX, MIN, NMISS, SUM, SUM DISTINCT, SUPPORTC, RANGE, STD, STDERR, USS, and VAR. MEAN is a synonym for AVG. FREQ and N are synonyms for COUNT, but these values do not accept the asterisk argument.

table-name
Table names can be one-part or two-part identifiers (for example, MyTable or Foo.MyTable). Identifiers such as Foo.MyTable require a previous LIBNAME statement to define the domain identifier (for example, Foo).

where-expression
This value is optional.

groupby-list
This value is optional. The value must be column names or projected aliases.

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

orderby-list
This value is optional. The value must be projected column names, aliases, or numbers that represent the position of a projected item (for example, **`SELECT a, COUNT(*) ORDER BY 2`**).

Formatted Parallel Group Select

By default, the columns of a Group-By statement are grouped by their unformatted value. You can use SQL pass-through parallel GROUP BY to group data by the columns output data format. For example, you can 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. If you group by the unformatted value, these dates will be

put into two separate groups. However, if you group by the formatted month name, these values will be put into the same month grouping of January.

You enable or disable SQL explicit pass-through formatted Parallel Group-By with the following EXECUTE statements:

```
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 following example code 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" on page 199.

Parallel Group-By SQL Reset Options

The server provides the following Parallel Group-By SQL reset options:

GRPSEL / NOGRPSEL
This option enables or disables the server Parallel Group-By facility.

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

FMTGRPSEL / NOFMTGRPSEL
This option enables or disables the server Parallel Group-By use of formats.

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

SCANGRPSEL / NOSCANGRPSEL
Use this option to turn on and off the server Index Scan facility. The default server setting uses the Index Scan facility.

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

/* Enable index scan facility */
execute(reset scangrpsel)
 by sasspds ;
```

Chapter 13

STARJOIN Facility

Understanding the STARJOIN Facility

Overview of the Server STARJOIN Facility

The server SQL Planner includes the STARJOIN facility. The 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, which 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 star join can minimize overhead data redundancy during query evaluation. If the server STARJOIN facility is not enabled or if the server SQL does not detect a star schema, then the SQL is processed using pairwise joins.

A star join differs from a pairwise join in that in the server, a pairwise join requires one step for each table to complete the join. A properly configured star join requires only

three steps to complete, regardless of the number of dimension tables. If a star schema consists of 25 dimension tables and one fact table, the star join is accomplished in three steps. But joining the tables in the star schema using pairwise joins requires 26 steps.

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

Star Schemas

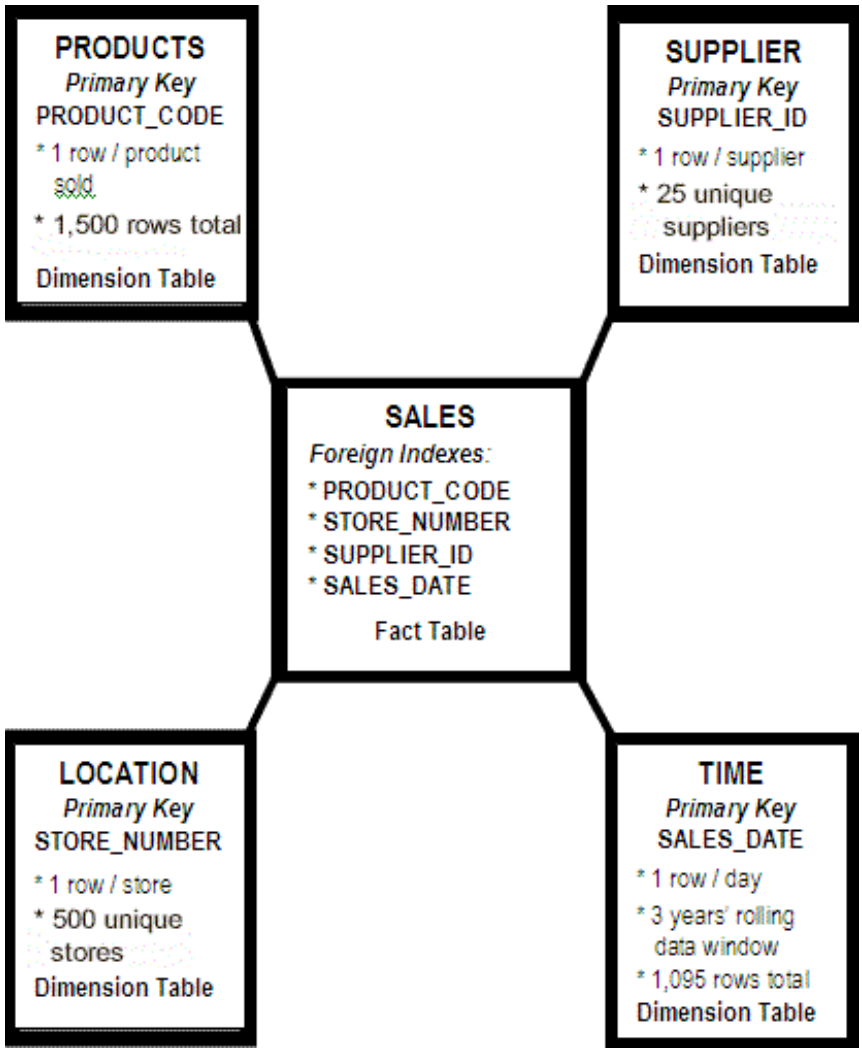
Overview of Star Schemas

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

Star schemas are the simplest data warehouse schema. They consist 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 that are in the 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.

The following figure is an example star schema. The dimension tables Products, Supplier, Location, and Time surround the fact table Sales.

Figure 13.1 *Example Star Schema*



Example Dimension Tables Information

In the preceding figure, the Products table contains information about 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 Products table contains 1,500 rows.

The Supplier table contains information about 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 Supplier table contains 25 rows.

The Location table contains information about the stores that sell 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.

The Time table is a sequential sales transaction table. Each row in the Time table represents one day out of a rolling 3-year, 365-day-per-year calendar. The row for each day contains information such as the date, day of week, month, quarter, year, and so on. The table contains 1,095 rows.

Fact Table Information

The fact table, Sales, 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 that contain facts or information that is 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 can contain as many as 1,500 x 25 x 500 x 1,095 = 20,531,250,000 rows.

Server STARJOIN Requirements

For the server SQL to take advantage of the STARJOIN Planner, the following conditions must be true:

- STARJOIN optimization is enabled in the server.
- The server star schema use a single, central fact table.
- All dimension tables in the server star schema are connected to the fact table.
- The server dimension tables appear in only one join condition.
- The server fact tables are equally joined to dimension tables.
- Dimension tables that do not use subsetting have a simple index on the dimension table's join column.

When you submit the server SQL that does not meet these STARJOIN conditions, the server performs the requested SQL task using the server's pairwise join strategy. For examples of valid, invalid, and restricted candidates for the server STARJOIN facility, see “STARJOIN Examples” on page 116.

Invoking the Server STARJOIN Facility

The server knows when to use the STARJOIN facility because of the topology of the data and the query. The server invokes STARJOIN based on the SQL that you submit. When you submit SQL and STARJOIN optimization is enabled, the server checks the SQL for admissible STARJOIN patterns. The server SQL identifies a fact table by scanning for a common, equally joined table among multiple join predicates in a WHERE clause. When the server SQL detects patterns that have multiple, equally joined operators that share a common table, the common table becomes the star schema's fact table.

When you submit an SQL statement to the server that uses structures that indicate the presence of a star schema, the STARJOIN validation checks begin.

Indexing Strategies to Optimize STARJOIN Query Performance

Overview of Indexing Strategies

When you have determined the baseline criteria for creating an SQL STARJOIN in the server, you can configure the indexes to influence which strategy the server STARJOIN facility chooses.

With the IN-SET strategy, the 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 indexes so that the STARJOIN facility chooses the IN-SET strategy, create a simple index on each SQL column in the fact table and

dimension table that you want to use in a join relation. A simple index prevents STARJOIN Phase I from rejecting a Phase I dimension table so that it becomes a non-optimized Phase II table. Simple indexes also facilitate the Phase II fact-table-to-dimension-table join lookup.

Indexing to Optimize the IN-SET Join Strategy

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';
```

The server IN-SET join strategy is the preferred strategy for almost every star join. If you want the 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 fact table F, create simple indexes on columns F.D1KEY and F.D2KEY.
- On 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 the simple indexes are the indexes that will enable the STARJOIN IN-SET join strategy.

Indexing to Optimize the COMPOSITE Join Strategy

For the COMPOSITE join strategy, the dimension tables with WHERE clause subsetting are collected from the set of equally joined predicates. You need a composite index 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, based on the best and simplest left-to-right match of the columns in the COMPOSITE join.

If the subsetting in a star join is limited to a single dimension table, then you can enable the COMPOSITE join strategy by creating a simple index on the join column of the single dimension table.

For the example code in "Indexing to Optimize the IN-SET Join Strategy" on page 109 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 you need the COMPOSITE index named COMP1 in order 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. As the number of dimension tables and join relations increases, the resulting increase in size can become unmanageable. The performance of the IN-SET strategy is robust enough 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 13.1 on page 107 has four dimension tables (Supplier, Products, Location, and Time) and one fact table (Sales). The schema has 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 stores that are in North Carolina:

```
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 '01JAN2015'd and '31JAN2015'd;

quit;
```

During optimization, the STARJOIN Planner examines the WHERE clause subsetting in the query to determine which dimension tables are processed first.

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 '01JAN2015'd and '31JAN2015'd`**) cause the server to process the Location and Time tables first. The remaining dimension tables, Supplier and Products, are processed second.

The server STARJOIN facility uses the first dimension tables to reduce the rows in the fact table to candidate rows that contain the matching criteria. The facility uses the values in each dimension table key 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. 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 range 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 2015.

Because the fact table and dimension tables for the STORE_NUMBER column have simple indexes, the STARJOIN facility chooses the IN-SET strategy. The facility subsets the STATE column values to **`'NC'`** in order to build the set of store numbers that are associated with North Carolina locations. The STARJOIN facility can use the set of

North Carolina store numbers to generate an SQL **`where ... in`** expression. SQL uses the **`where ... in`** expression to efficiently subset the candidate rows in the fact table before the final SQL expression is evaluated.

Overview of STARJOIN Optimization

The 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 of the process probes fact table indexes and selects a STARJOIN strategy.
3. Phase II of the process performs index lookups and joins subsetted fact table rows with Phase II tables.

Enabling STARJOIN Optimization

The server STARJOIN optimization is enabled by default. For information about statement options that enable or disable the STARJOIN facility in the server, see "STARJOIN RESET Statement Options" on page 114.

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

After the STARJOIN Planner validates the join subtree, join optimization begins. Join optimization is the process that searches for the most efficient SQL strategy to use to join the tables in the star schema.

The first step in the server's join optimization is to examine the dimension tables that were called by SQL for structures that the 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 SQL that you submit filters or subsets the table's contents determine its classification. The 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 fewer), or a dimension table whose SQL queries contain one or more filtering criteria that are 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 shown in Figure 13.1 on page 107, which contains the fact table Sales and the dimension tables Products, Supplier, Location, and Time.

Suppose that you submit an SQL query that requests transaction reports for all suppliers and for all products that meet the following criteria from the fact table Sales:

- The store location is North Carolina.
- The time period is the month of January.

The SQL query subsets the Location and Time tables, so the 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 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 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 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 the Phase I index probe optimizes the candidate rows in the fact table, the probe examines index structures to determine the best STARJOIN strategy to use. There are two 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. You can determine which strategy the server chooses by providing the required table index types in the SQL that you submit.

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 fact table candidate rowset is reduced to transactions from only North Carolina stores, which 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, the server selects the STARJOIN strategy by using the following logic:

- If the probe finds one or more simple indexes on fact table and dimension table SQL join columns, and does not find spanning composite indexes on the fact table, the server selects the STARJOIN IN-SET strategy.
- If the probe finds an optimal spanning composite index on the fact table, and does not find simple indexes on fact table and dimension table SQL join columns, the server selects the STARJOIN COMPOSITE strategy.
- If the probe finds both simple and spanning composite indexes, the server generally selects the STARJOIN IN-SET strategy. If 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, the server selects the IN-SET strategy.

- If the probe does not find suitable indexes for either STARJOIN strategy, the server does not use STARJOIN. Instead, it joins the subtree using the standard server pairwise join.

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

The IN-SET join strategy will cache temporary Phase 1 probes in memory, when possible, for use by Phase 2. The caching can result in significant performance improvements. By using in-memory lookups into the dimension tables for Phase 2 probes, rather than performing more costly file system probes of the dimension tables, significant performance improvements can result. The amount of memory allocated for Phase 1 IN-SET caching is controlled by the STARSIZE= server parameter. You can use the STARJOIN DETAILS option to see which partial results of the Phase 1 IN-SET strategy are cached, and whether sufficient memory was allocated for STARJOIN to cache all partial results.

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. In the term *IN-SET*, *IN* 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. 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 that is 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 subsetting statement for a dimension table:

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

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. Therefore, 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
 ('01JAN2015'd,'02JAN2015'd,'03JAN2015'd,
  '04JAN2015'd,'05JAN2015'd,'06JAN2015'd,
  '07JAN2015'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 subsetting of the aggregated dimension table. The 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 matrices 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 Phase II completes the joins with the Phase I dimension tables, Phase II performs index lookups from the fact table to the Phase II dimension tables. For Phase II dimension tables, indexes should be created on all columns that join with the fact table.

When the 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 SQL results set that you want.

STARJOIN RESET Statement Options

The server uses RESET statements in the server SQL to provide information about and to configure the server STARJOIN settings.

RESET NOSTARJOIN=[0/1]

The NOSTARJOIN option suppresses the use of the server STARJOIN optimizer in the planning and running of SQL statements that have valid STARJOIN patterns or star schemas. When NOSTARJOIN is enabled, the server ignores STARJOIN and uses pairwise joins to plan and run SQL statements. The default setting is NOSTARJOIN=0, which means that STARJOIN is enabled, and STARJOIN optimization occurs when SQL recognizes a valid server pattern or star schema.

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

NOSTARJOIN=0

enables the server STARJOIN facility.

NOSTARJOIN=1
disables the server STARJOIN facility.

Note: The statements NOSTARJOIN and NOSTARJOIN=1 are equivalent.

RESET STARMAGIC=*nnn*
STARMAGIC is the STARJOIN counterpart to the SQL MAGIC number option. You can use STARMAGIC options to manually adjust certain internal STARJOIN heuristics to improve certain join strategies.

The STARMAGIC option uses bit flags to configure the STARJOIN code. You can select different controls by adding the values for the bit flags below:

```
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
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.

RESET DETAILS="stj$"
The RESET DETAILS option prints details about your server STARJOIN facility settings. All internal STARJOIN debugging information is tied to the stj$ DETAILS key. You issue the stj$ reset option to display available information as the server attempts to validate a join subtree. The RESET DETAILS="stj$" option is very useful for debugging STARJOIN and SQL statement execution.

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

Example: STARJOIN RESET Statements

The following example connects to SASSPDS. Then the code issues the "stj$" RESET option to display all available information as the server attempts to validate the join subtree for the SQL on a star schema. The STARMAGIC=16 setting disables the STARJOIN COMPOSITE join strategy (STARJOIN COMPOSITE joins are enabled by default in the server). The NOSTARJOIN=0 setting means that STARJOIN is enabled (or resets a disabled STARJOIN facility) and ensures that STARJOIN optimization occurs if the server SQL recognizes a valid server pattern or star schema. (The STARJOIN facility is enabled by default in the server.)

After you submit 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;
```

STARJOIN Examples

Example 1: Valid SQL STARJOIN Candidate

The following code is an example of an SQL submission that the server can use as a star schema. The submission is a valid candidate for the following reasons:

- 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 the server cannot use as a star schema because no single central fact table can be identified.

```
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;
```

The server cannot use the SQL submission in this example as a star schema This 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 the server supports calculated or created columns. The following code is an example of an SQL submission that creates columns. This code 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 code creates a column called State_Abv. The server STARJOIN facility supports created columns if the appropriate indexes on the join columns exist in the fact table and dimension tables.

Chapter 14

Optimizing Index Scans and Correlated Queries

Optimizing Index Scans

The server SQL gives you the ability to use quick index scans on large tables. Rather than scan entire tables sequentially, which can have millions or billions of rows, the server SQL can scan cached index metadata. The 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.

All the index scan capabilities are available for both standard server tables and 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 that is included with the index scan support that does not require an index on the table. For example, consider the following code:

```
select count(*) from tablename;
```

This code returns the number of rows in the large table Tablename without performing a row scan of the table. Table metadata returns the correct number of rows. As a result, the response is as fast as an index scan, even on an unindexed table.

COUNT(*) functions with WHERE clauses require each column referenced in the WHERE clause to have an index in order for the index scan feature to improve performance. For example, suppose the server table Foo has indexes on numeric columns A and B. The following COUNT(*) functions benefit from the 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 server table Bar has indexes on numeric columns X and Y. The following SQL submissions 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 server table Oops has indexes on numeric columns X and Y. Column Z is not indexed. The following server SQL statement is entirely evaluated by table scan. Index scanning is not performed on any of the functions.

```
select min(x),
       min(y),
       count(x),
       count(y),
       count(z)
 from Oops;
```

To take advantage of index scans, you could resubmit the previous statement in the following way:

```
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 to exploit the performance gains. The function COUNT(y) continues to be evaluated by table scan. You can combine the COUNT(*) function with other functions and benefit from index scan performance gains. For the server table Oops, with indexes on numeric columns X and Y, the following server SQL statement benefits from index scan performance:

```
select min(x),
       range(y),
       count(x),
       count(*)
from Oops;
```

The server index scan is an extension of the server Parallel Group-By facility. The query must first be accepted by the Parallel Group-By facility to be evaluated for an index scan. For more information, see “Understanding the Parallel Group-By Facility” on page 99 . 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

A correlated query is a SELECT expression in which a predicate within the query has a relationship to a column that is defined in another scope. Business and analytic intelligence tools often generate SQL queries that are nested three or four layers deep. Queries with cross-nested relationships consume significant processor resources and require more time to process. Algorithms in the SQL Planner of the server implement techniques that significantly improve the performance of correlated queries for patterns that permit query rewrites or query decorrelation.

The SQL Planner improves correlated query performance by changing complex rules about nested relationships into a series of simple steps. The server can process the simple steps much faster than it can process the complex rules that apply to 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 the Planner finds a complex nested relationship, it restructures or recodes the SQL query into a simpler form by using temporary server tables.

Correlated Query Options

The server has the following SQL options for use with correlated query rewrites.

_QRW / NO_QRW

Use the _QRW / NO_QRW option to configure the server to enable or disable the query rewrite facility diagnostic output, which includes debugging and tracing information. The debugging and tracing output is generated when the server query rewrite facility detects subexpressions. The query rewrite facility then rewrites and executes the SQL code. The SQL code produces the intermediate results and the final rewritten SQL statement. By default, the server _QRW option for diagnostic output is not enabled.

The _QRW=1 / _QRW=0 option and the NO_QRW=0 / NO_QRW=1 option do the same thing as the _QRW / NO_QRW option.

```
/* 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 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. Typically, you do not specify this option unless you want to test whether an SQL statement runs faster without rewrite optimization, or if you suspect that the resulting rowset that you get from a query rewrite evaluation is incorrect.

The _QRWENABLE=1 / _QRWENABLE=0 option does the same thing as the _QRWENABLE / NO_QRWENABLE option. The query rewrite facility is enabled in the server by default.

```
/* 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 ;
```

Here is an example:

```
%let spdshost=localhost;
%let spdsport=5400;
%let user=anonymous;

libname spdslib sasspds 'tmp'
  host="&spdshost"
  serv="&spdsport"
  user="&user"

LIBGEN=YES IP=YES;

data spdslib.a;
       do i=1 to 10;
          x=i;
          output;
       end;
run;

 data spdslib.b;
  do i=1 to 100;
    x=i;
      y=1+floor(100*ranuni(9999));
   output;
  end;
run;

%let spdssqlr=_qrw;

proc sql _method;

   select * from spdslib.a
    where x in (select x from (select b.x from spdslib.b where b.y gt 50))
;
```

```
quit;
```

Chapter 15

Server-Side Sorting

Server-Side Sorting

Overview of Server-Side Sorting

In most instances, using a BY clause in SAS code submitted to a server table triggers a BY clause evaluation by the server. This BY clause assertion to the server might or might not require sorting to produce the ordered rowset 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, you can use a server SQL pass-through COPY statement to avoid the overhead of the data round-trip.

The 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. The 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, the 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 a server table with a BY clause to order them in a desired way, the 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 rowset 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 the server.

Chapter 16

WHERE Clause Planner

Optimizing WHERE Clauses

Overview of Optimizing WHERE Clauses

SPD Server includes advanced methods to optimize WHERE clauses. Before SAS SPD Server 4.0, the rule-based, heuristic WHERE clause planner WHINIT was used to manually tune queries for performance. Now the server provides dynamic WHERE clause costing, which is an automatic feature that can replace the need to manually tune queries. The server's dynamic WHERE-costing uses factors of cardinality and distribution to calculate relative processor costs of various WHERE clause options. The server administrators can set server parameter commands in the spdsserv.parm parameter file, or users can set macro variables to turn dynamic WHERE-costing on and off. If dynamic WHERE-costing is turned off, the server reverts to using the rules-based WHERE clause planner.

WHERE Clause Definitions and Terminology

WHERE clauses
selection criteria for a query that specify one or more Boolean predicates. Implementing the criteria, the server selects only rows that satisfy the WHERE clause.

Predicates
the building blocks of WHERE clauses. Use them stand-alone or combine them with the operators AND and OR to form complex WHERE clauses. Here is an example of a WHERE clause:

```
"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, use **where x > 1 and not (y in (1 2 3))**.

Boolean logic
determines whether two predicates joined with an AND or OR are true (satisfy the specification) or false (do 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, and p3) 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
structures associated with tables that permit the server to quickly access rows that satisfy an indexed predicate. In an example WHERE clause, **`where x = 10 and y > 11`**, the server selects the best index on column X to directly retrieve rows that have a value of 10 in the X column. If no index exists for X, the server must sequentially read each row in the table searching for X equal to 10.

Simple index
index on a single column.

Composite indexes
composite indexes index two or more columns. The list of columns in an index is sometimes called the index key.

Parallelism
the server capability that enables multiple threads to execute in parallel. Using multiple processors in parallel mode is sometimes called “divide and conquer” processing. The server uses parallelism to evaluate the multiple indexes that are involved in more complicated WHERE clauses.

Server Indexing with WHERE Clause

Overview of Server Indexing with WHERE

The six evaluation strategies that the WHERE clause planner uses are EVAL 1, EVAL 2, EVAL 3, EVAL 4, EVAL 5, and EVAL 6. The different EVAL strategies calculate the number of rows that will be required to execute a given query.

True rows 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 indexes. EVAL 2 and EVAL 6 evaluate true rows of a table without using indexes. EVAL strategies are explored in more detail in “WHERE Clause EVAL Strategies” on page 131.

Server tables can have one or more indexes. A table can use a combination of four different indexing strategies. 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. Server indexing evaluates the processor cost of a WHERE clause. The section “WHERE-Costing Using Cardinality Ratio and Distribution Values” on page 131 shows how factors of cardinality and distribution are used to choose the evaluation strategy that will perform the WHERE clause at the smallest processor cost.

SPD Indexes

The server uses segmented indexes. 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-indexes each points to blocks of rows in the table. By default, the server creates an index segment for every 8192 rows in a table.

The SPD segmented index facilitates the 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 macro 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.

Indexing with WHERE

A parallel WHERE clause on a table that is indexed is done in two phases. The first phase, pre-evaluation, uses the 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. The thread uses the server index to read the segment bitmap. The per-segment bitmaps identify the segment rows that satisfy the query for that particular column. If you include more than one indexed column in the WHERE clause, the server retrieves the per-segment bitmaps for each column in parallel (as are the segments for each column). After the server retrieves all the bitmaps for each column of the segment, it determines which rows satisfy the query and then returns those segment rows to the client. The multi-threaded per-segment queries begin execution at the same time, but 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. For more information about using indexed columns with WHERE clause evaluations, see “Understanding the WHERE Clause Planner” on page 131 .

When a table is modified as the result of an append or update, all server indexes on the table are updated. When the index is updated, the per-value segment lists can potentially fragment or some disk space might be wasted. A highly fragmented 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 more information, see “Index Utility” in *SAS Scalable Performance Data Server: Administrator’s Guide*.

MINMAX Variable List

SPD Server has a table option called MINMAXVARLIST=. The primary purpose of MINMAXVARLIST= is for use with 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 a 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 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 that 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 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.

For more information, see "MINMAXVARLIST= Table Option" on page 260.

Understanding the 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. Before SAS SPD Server 4.0, 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 the server does the tuning work for the user by automatically costing the different approaches to index evaluation.

WHERE-Costing Using Cardinality Ratio and Distribution Values

Two key factors are used to evaluate, or cost, WHERE clause indexes: cardinality ratio and distribution.

The *cardinality ratio* refers to the proportion expressed by the number of distinct values in the index divided by the number of rows in a table. When many rows in a table hold the same value for a given variable, the variable value is said to have a low cardinality ratio. An example of a table with a low cardinality ratio might be a table of unleaded gasoline prices from service stations in the same area of a large city. Tables that have a low cardinality ratio feature many rows, but only a few unique row values.

Conversely, when a table has only one or a few rows that contain the same variable value, then that table can be described as having a high cardinality ratio. An example of a table with a high cardinality ratio might be an office phone directory, where the variable for phone extension is always unique. Tables that have a high cardinality ratio tend to contain many rows with very few repeating, or non-unique values.

The cardinality ratio for an index is in the range 0–1. Indexes with a high cardinality ratio value of 1.0 are completely unique with no repeated values. Indexes with a low cardinality ratio generate a score that approaches zero as the number of unique variable values diminish. The closer to zero, the lower the cardinality ratio of the index.

Distribution refers to the sequential proximity between rows for values of a variable that are repeated throughout the variable's table distribution. When a certain value for a variable exists in many rows 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

Server indexing keeps track of the cardinality ratio 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 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 indexes. EVAL 2 and EVAL 6 evaluate true rows of a table without using indexes.

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 indexes. EVAL 1 combines multiple segment bitmaps from queries that use multiple indexes 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 eliminates 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
a single index sequential process. Use EVAL 3 when the number of rows returned by an index is unique or nearly unique (when cardinality ratio is high). EVAL 3 returns a list of true rows for the entire table. EVAL 3 supports only the equality operators **EQ** and =.

EVAL 4
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 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, suppose that when X is indexed, the server uses EVAL 5 to evaluate the following 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(distinct), NMISS(), and RANGE() functions. The EVAL 5 strategy cannot be used on SQL expressions, which use functions other than those listed above.

EVAL 6
emulates the behavior of EVAL 2. With EVAL 6, the query is a candidate for Hadoop WHERE processing. If the Hadoop WHERE processing fails, EVAL 6 reverts to the EVAL 2 operation. EVAL 6 takes true rows as determined by EVAL 1, EVAL 3, or EVAL 4, and then eliminates 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.

The WHERE clause planner in SAS SPD Server 3.x relied heavily on EVAL 1 and EVAL 2 threaded strategies to evaluate most clauses. Sometimes the SAS SPD Server 3.x EVAL 1 and EVAL 2 strategies would over-thread and over-manipulate indexes during the WHERE clause evaluation. This resulted in reduced performance or excessive

resource consumption. Beginning with SAS SPD Server 5.2, which introduced WHERE clause costing, 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 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 indexes 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. With the server WHERE clause costing in place, EVAL 3 and EVAL 4 strategies are more suitable evaluation engines that conserve resources and boost processor performance.

The second tree, which does not use the EVAL 2 method, is scanned for predicate expressions that return values with a low cardinality ratio. When low cardinality ratio predicate expressions are identified, they are ranked. The predicate expression with the lowest cardinality ratio 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 predicate expression with the highest cardinality ratio 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 indexes when a single index is sufficient.

When the WHERE clause planner determines that no predicate expressions meet the low cardinality ratio 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 the 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

required amount of index work 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, the server uses segmented costing. In segmented costing, true segments are passed to EVAL 2 for processing. EVAL 2 processes only table segments that can provide true rows for the WHERE clause.

Turning WHERE Clause Costing Off

You can use the server spdsserv.parm parameter file to configure the default WHERECOSTING parameter setting to ON. To turn off WHERE clause costing within the scope of a job, you can use macros or a DATA step:

- The SPDSWCST=NO macro setting turns off WHERE clause costing. If you turn off WHERE costing in the spdsserv.parm parameter file, or if you use the macro setting SPDSWCST=NO, the WHERE clause planner reverts to a non-costing, rules-based algorithm.
- The SPDSWSEQ=YES macro overrides WHERE clause costing, and enables you to force a global EVAL3 or EVAL4 strategy.
- The WHERECOSTING parameter can be removed or set to NOWHERECOSTING in the spdsserv.parm parameter file 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 the 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 selects 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 the following:

- 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. Here is an example of a predicate with an OR expression: 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 a 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 the 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 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 rows 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 the server can identify correctly ALL resulting rows, without help from the SAS System. First, the server will call EVAL1, an indexed method, to quickly access a list of rows 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 rows.

When output from EVAL1 displays the suffix 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.

The 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 mininum/maximum range. WHINIT output keywords can indicate pruning activity. For example, if WHINIT had determined that the values for D (in the WHERE clause) are between 5 and 13, then the predicate `where d = 3` could never be true. In this case, WHINIT would have pruned this predicate because it is logically impossible, or FALSE. Pruning can also affect higher nodes. If the **d = 3** predicate were deemed FALSE, then the AND subtree 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** - The server can evaluate ALL of the WHERE clause when determining which rows satisfy the clause.
- **SOME** - The server can handle SOME or part of the WHERE clause. It will then need SAS to help identify resulting rows.
- **NONE** - The server cannot evaluate this WHERE clause. SAS will perform all evaluations.
- **TRUE** - The server has determined that the entire WHERE clause is TRUE, and that all the rows satisfy the given WHERE clause.
- **FALSE** - The server has determined that the WHERE clause is FALSE. That is, no rows can satisfy the WHERE clause.
- **RC=number** - An internal error has occurred. The error number is displayed.
- **EVALx** - the EVAL strategies that the planner will use. *x* can be 1, 2, 3, or 4.

Composite Index Permutations

A composite index can involve one or more in sets of 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 the 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 fewer input/output operations on the index.

How to Affect the WHERE Planner

SPDSWCST Macro Variable

To turn off dynamic WHERE-costing, specify the following:

```
%let SPDSWCST=NO;
```

SPDSWDEB Macro Variable

To turn on WHINIT planning output, specify the following:

```
%let SPDSWDEB=YES;
```

SPDSIRAT Macro Variable

To affect the WHERE planner SPD index pre-evaluation, specify the following:

```
%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%. This means that if an index value is contained in 75% 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% 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% **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 that 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.

SPDSNIDX Macro Variable or NOINDEX= Table Option

To suppress WHINIT use of any index, specify the no index server macro variable or the corresponding server table option:

```
%let SPDSNIDX=YES;

data _null_;
set foo.a (noindex=yes);
```

SPDSWSEQ Macro Variable

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 rows that can satisfy this predicate, EVAL3 might be a better choice. To force WHINIT to choose EVAL3/4, specify the following:

```
%let SPDSWSEQ=YES;
```

Note: When SPDSWSEQ=YES, it overrides server WHERE clause costing decisions.

[NO]WHERECOSTING Server Parameter Option

Controls whether the server uses dynamic WHERE-costing. When dynamic WHERE-costing is disabled, 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 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.

Why and When to Suppress Indexes

Most rule-based planners, including WHINIT from the 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 is 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 rows. 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 as follows:

```
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 rows. In this case, suppressing the **abcd** index might be a good idea.

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 that uses the SAS OBS= data set option to SPD Server .

The SAS OBS= data set option causes processing to end with the specified (nth) row in a table. Because parallel WHERE clause processing is threaded, subsetting a table and using OBS= might not produce identical results from run to run. Different batch jobs using the same WHERE clause code might produce slightly different results.

When a parallel WHERE-clause evaluation is split into multiple threads, the 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= 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 server thread count value, you can use the SPDSTCNT macro variable:

```
%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 the in this example:

```
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 following thread count statement to the beginning of each job.

```
%let SPDSTCNT=1;
```

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 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. This means that no rows can satisfy this WHERE clause. Thus, as a result of the pre-evaluation, no rows 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 columns **i** and **j**, which is an equality index predicate. Column **k** is **not** included because it involves an inequality operator (greater than). Because there are no other indexes for column **k**, this predicate is assigned to EVAL2 . EVAL2 will post-filter the rows 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**.

Part 5

SPD Server Reference

Chapter 17

SPD Server LIBNAME Statement

Overview of the SPD Server LIBNAME Statement

The SPD Server LIBNAME statement enables you to establish a connection to a server domain from your SAS session and associate a libref with the connection. A *libref* is a logical name of your choosing that serves as a library reference. After you define a libref, you can specify the libref with a table name in the SAS DATA step and SAS procedures. This enables you to read, create, and update the table in the referenced domain, if you have appropriate access. The libref and table name are specified in the

form *libref.tablename*. In SPD Server, the LIBNAME statement is also used to change server passwords.

LIBNAME Statement Syntax

The server LIBNAME statement has the following syntax:

```
libname libref engine 'domain' connection-options
<user-validation-options>
<data-processing-options>
<password-management-options>;
```

Required Arguments

libref
a shortcut name or a nickname for the location where your SPD Server files are stored. The name can be up to eight characters long and must conform to the rules for SAS names. For more information about SAS names, see *SAS Language Reference: Concepts*.

You will use the libref as one part of a two-part table name in the form *libref.table-name* to identify your SPD Server tables in SAS statements.

If you want to use one-part names to reference tables (*MyTable*) instead of two-part names, you can assign the libref USER. SPD Server will store tables identified with one-part names (or with two-part names that specify the libref USER) in the domain pointed to by the USER libref. For more information about the USER libref, see "User Library" in "SAS System Libraries" in *SAS Language Reference: Concepts*.

To be able to create temporary tables in the domain, specify the TEMP= LIBNAME option in the LIBNAME statement. For more information, see "TEMP= LIBNAME Statement Option" on page 173.

engine
the name of the engine that will be used to access SPD Server. The valid value is **SASSPDS**.

'*domain*'
the name of a server domain. The domain must have been defined in advance by a server administrator.

connection-options
provide information to connect to SPD Server. To connect to the server, you must specify a server host name and the port number of the SPD Server name server. If your server is using ACL security, you must also authenticate yourself to the server. You have the choice of several arguments for both tasks.

AUTHDOMAIN=
Allows authentication to the server by specifying the name of an authentication domain metadata object. For more information, see "AUTHDOMAIN= LIBNAME Statement Option" on page 154.

Note: If you use AUTHDOMAIN=, do not specify USER= and PASSWORD= or PROMPT=.

HOST=
specifies the network node name or IP address of the server host (for example: nsname or 123.456.763). When you use HOST=, you must specify the name server port number or port name in the SERVICE= argument. For more information, see "HOST= LIBNAME Statement Option" on page 160.

Note: You can use either HOST=, SERVER=, or the %SPDSHOST macro variable to identify the server host to the SAS session. The HOST= argument is preferred when you need to follow FTP conventions.

TIP Use an IP address with HOST= when your host is multi-homed (contains multiple network cards).

PASSWORD=
specifies the password associated with the specified server user ID. For more information, see "PASSWORD= LIBNAME Statement Option" on page 168.

Note: When native authentication is configured, you must change your password the first time that you use SPD Server. For more information, see "Changing Server Passwords" on page 17.

Note: You can use either PASSWORD= or PROMPT= to supply the password. PROMPT= is considered to be more secure.

Note: If you do use PASSWORD=, we recommend encoding the password with the PWENCODE procedure. You must run the PWENCODE procedure in SAS. See the Base SAS documentation for more information about the PWENCODE procedure.

PROMPT=YES | NO
Specifies whether to prompt the server user for a password. The default value is NO. Specify YES to request a prompt. For more information, see "PROMPT= LIBNAME Statement Option" on page 169.

Note: Use either PASSWORD= or PROMPT= in the LIBNAME statement. Do not use both.

SERVER=
specifies the network node name of the server host and the name server port number or service name. Two examples are "nsname.5400" or "nsname.spdsname". The default port number for SPD Server is 5400. The default service name is spdsname, if one is configured. Your installation might have been configured with a different port number and service name. For more information, see "SERVER= LIBNAME Statement Option" on page 170.

Note: You can use either SERVER=, HOST=, or the %SPDSHOST macro variable to specify the server host for your SPD Server connection. SERVER= is compatible with SAS/SHARE software.

SERVICE=
specifies the name server port number or service name. The default service name is spdsname. This service name must have been previously configured. Your administrator will tell you if it is available. This argument is used in conjunction with HOST=. Alias: SERV=.

USER=
specifies a server user ID. When ACL security is active, this user ID is given to you by the server administrator. When only UNIX file security is active, specify ANONYMOUS.For more information, see "USER= LIBNAME Statement Option" on page 175.

Optional Arguments

The following table contains a functional list of optional LIBNAME statement arguments supported by the SASSPDS engine.

Option	Description
User Validation Options	
ACLGRP=	Names an ACL group that has been previously assigned to the server user ID. A server user ID can have definitions for 5 to 32 user groups, depending on how the server was configured.
ACLSPECIAL=	Invokes special access to server resources in the server domain for a server user. The special privilege must already exist in the password database.
Data Processing Options	
BYSORT=	Specifies whether to use implicit automatic server sorts on BY clauses.
DISCONNECT=	Controls how user proxy resources are assigned for a server user.
ENDOBS=	Specifies the end row number in a user-defined range for processing.
IP=	Invokes implicit SQL pass-through processing in the SPD Server session.
LIBGEN=	Configures the SPD Server connection to generate additional domain connections.
LOCKING=	Record-level locking allows multiple users concurrent Read and Write access to server tables.
NETCOMP=	Compresses the data stream for a server network packet.
PASSTHRU=	Invokes implicit SQL pass-through processing in the SPD Server session.
SCHEMA=	Specifies the server domain name.
SHARE=	Enables enhanced sharing of user proxies.
STARTOBS=	Specifies the start row number in a user-defined range for processing.
TEMP=	Controls the creation of a temporary server domain for this LIBNAME assignment.
TRUNCWARN=	Suppresses hard failure on NLS transcoding overflow and character mapping errors.
UNIXDOMAIN=	Specifies to use UNIX domain sockets for data transfers between the client and SPD Server.
Password Management Options	

Option	Description
CHNGPASS=	Specifies to prompt a server user for a new password.
NEWPASSWORD= or NEWPASSWD=	Specifies a new password for a server client user.

Dictionary

ACLGRP= LIBNAME Statement Option

Names an ACL group that has been previously assigned to the SPD Server user ID.

Valid in: SPD Server LIBNAME Statement

Note: Option to identify the server user.

Syntax

ACLGRP=*acl-group*

Required Argument

acl-group
an ACL group that the server administrator assigned to your server user ID. (You can be assigned from 5 to 32 ACL groups, depending on how your server is configured.)

Details

By default, the server connection uses the first ACL group in your group list to make a connection. Use the ACLGROUP= option to specify a different group name from the list.

Example

```
%let spdshost=samson;
libname mylib sasspds 'spdsdata'
   user='receiver'
   aclgrp='PROD'
   prompt=yes;
```

Note: Password values are case sensitive. If the server administrator assigns a lowercase password value, you must enter the password value in lowercase.

ACLSPECIAL= LIBNAME Statement Option

Invokes special access to SPD Server resources in the server domain for a server user. The special privilege must already exist in the password database.

Valid in: SPD Server LIBNAME Statement

Note: Option for Access Control Lists (ACLs).

Syntax

ACLSPECIAL=YES | NO

Required Arguments

YES
invokes special access to all server resources in the domain. These privileges override normal ACL restrictions and enable the user to access the resources of other users, as well as to create or modify ACLs of other users.

NO
denies special access to all server resources in the domain. Access is evaluated based on the ACL security defined on each domain resource. This is the default value.

Details

Server user IDs are registered in a password database. The password database supports privilege levels that confer special privileges. All connections from the SASSPDS engine are made as a regular user, regardless of the privileges a server user has defined in the password database. The ACLSPECIAL= LIBNAME statement option invokes special privileges from the password database in the SAS session. If you specify ACLSPECIAL=YES and do not have special privilege in the database, the server returns an error.

Example

Invoke special privileges for TheBoss, to enable him to Read, Write, Alter, and Control all tables in the Conversion_Area domain. (The administrator has defined TheBoss as 'special'.)

```
libname mylib sasspds 'conversion_area'
   server=husky.5400
   user='theboss'
   prompt=yes
   aclspecial=yes ;
```

AUTHDOMAIN= LIBNAME Statement Option

Enables connection to SPD Server by specifying the name of an authentication domain metadata object.

Valid in: SPD Server LIBNAME Statement

Note: Option to identify the server user.

Syntax

AUTHDOMAIN=*authentication-domain*

Required Argument

authentication-domain
name of an authentication domain metadata object.

Details

When you use AUTHDOMAIN=, you must specify the server host in the LIBNAME statement. However, the authentication domain references credentials so that you do not need to specify the USER= and PASSWORD= LIBNAME statement options.

```
AUTHDOMAIN=SPDS
```

An administrator optionally creates an authentication domain definition at the same time that they are creating user definitions with the User Manager in SAS Management Console. The authentication domain is associated with one or more login metadata objects. These objects provide access to the server and are resolved by the SASSPDS engine calling the SAS Metadata Server and returning the authentication credentials.

The authentication domain and the associated login definition must be stored in a metadata repository, and the metadata server must be running in order to resolve the metadata object specification.

Example

```
libname mylib sasspds "spdsdata"
  host="husky"
  service="5400"
  authdomain=spds;
```

BYSORT= LIBNAME Statement Option

Specifies whether to use implicit automatic SPD Server sorts on BY clauses.

Valid in: SPD Server LIBNAME Statement

Default: YES

Note: Processing option

Syntax

BYSORT=YES | NO

Required 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.

Details

The default behavior of SPD Server is to perform a sort whenever the server encounters a BY clause. Specify BYSORT=NO to disable automatic sorting. When BYSORT=NO, the server will perform the same as the Base SAS engine. It will require an explicit sort statement (SORT procedure) to sort data.

Examples

Example 1: Turn Off Implicit Sorts for the SAS Session

Specify to turn off implicit server sorts for the session.

```
libname mylib sasspds 'conversion_area'
   server=husky.5400
   user='siteusr1'
   prompt=yes
   bysort=no ;

data mylib.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=mylib.old_autos;
   by model;
run;
```

In this program, the PRINT procedure will return an error message because the table Mylib.Old_Autos is not sorted.

Example 2: Use Table Option to Enable Implicit Sorting

Turn off implicit server sorts with the LIBNAME statement option, but specify a server sort for the table Mylib.Old_Autos by using the BYSORT= table option.

```
proc print data=mylib.old_autos
 (bysort=yes);
```

```
  by model;
run;
```

CHNGPASS= LIBNAME Statement Option

Specifies whether to prompt a server user for a change of password.

Valid in: SPD Server LIBNAME Statement

Default: NO

Note: Option to change a server user's password.

Syntax

CHNGPASS=YES | NO

Required Arguments

YES
prompts for a change of the user password.

NO
suppresses a prompt for a change of the user password. This is the default.

Details

When ACL security is enabled, a server user sometimes has to change his password. The CHNGPASS= option is one way to make a password change. The server validates the old password then saves the password supplied at the prompt in the password database.

Example

Specify a prompt to change the password of the 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 statement 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. Then check with your LDAP server administrator to ensure that the LDAP database also records password changes.

See Also

"NEWPASSWORD= LIBNAME Statement Option" on page 166

DISCONNECT= LIBNAME Statement Option

Controls how user proxy resources are assigned for a server user.

Valid in: SPD Server LIBNAME Statement

Default: NO

Syntax

DISCONNECT=YES | NO

Required Arguments

YES
closes network connections between the SAS client and the server when server librefs are cleared.

NO
closes network connections between the SAS client and the server only when the SAS session ends. This is the default setting.

Details

Each server user in a SAS session requires a server user proxy process to handle client requests. By default, this proxy is opened when the user assigns the first LIBNAME statement in the SAS session. It remains open until the SAS session ends. Closing the network connection ends all server user proxy processes for that session.

The advantage of this behavior is that the processor overhead required to create a server user proxy is required only when a user issues the first LIBNAME statement of the session. The disadvantage is that the server user proxy does not terminate until the user's SAS session ends. For example, if a user does not log off at the end of the day and leaves a server session running overnight, the user proxy remains in force, occupying system resources that might be used by other jobs.

The DISCONNECT= LIBNAME statement option is provided to control how user proxy resources are created and terminated for a server user. When DISCONNECT=YES is specified in the LIBNAME statement, the network connections between the SAS client and the server user proxy are closed when the user's last server libref in the SAS session is cleared. The clearing process closes the network connection and the server user proxy but not the SAS session. If the user issues a subsequent server libref in that SAS session, a new server user proxy is started up.

The advantage of using DISCONNECT=YES is that user resources are freed as soon as the user's last libref is cleared. The disadvantage of using DISCONNECT=YES is that the user needs to issue a subsequent LIBNAME statement in that session. Each LIBNAME assignment will launch a new server user proxy.

The DISCONNECT=YES LIBNAME statement option must be used with the LIBNAME CLEAR statement to be effective.

The DISCONNECT= state of the user proxy is determined by the first LIBNAME statement a user issues in the SAS session.

Example

The following output shows the result of multiple LIBNAME assignments under the default setting of DISCONNECT=NO. 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.5400
   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/
```

If user Bob assigns another libref that specifies DISCONNECT=YES without first clearing the previous librefs, the new libref (Fruit) will reuse 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= LIBNAME Statement Option

Specifies the end row number in a user-defined range for processing.

Valid in: SPD Server LIBNAME Statement

Syntax

ENDOBS=*n*

Required Argument

n
the number of the end row.

Details

By default, the server processes the entire table unless the user specifies a range of rows with the STARTOBS= LIBNAME statement option or the ENDOBS= LIBNAME statement option. If the STARTOBS= LIBNAME statement option is used without the ENDOBS= LIBNAME statement option, the implied value of ENDOBS= is the end of the table. When both options are used together, the value of the ENDOBS= LIBNAME statement option must be greater than the STARTOBS= LIBNAME statement option.

Comparisons

In contrast to the Base SAS software options FIRSTOBS= and OBS=, the STARTOBS= LIBNAME option and ENDOBS= LIBNAME option can be used for WHERE clause processing in addition to table input operations.

Example

Specify for the server to process only row numbers 200-500 while the LIBNAME statement is active.

```
libname mydatalib sasspds 'conversion_area'
   server=husky.5105
   user='siteusr1'
   prompt=yes
   startobs=200
   endobs=500;
```

See Also

SPD Server macro variables:

- “SPDSEOBS Macro Variable” on page 219

SPD Server table options:

- “ENDOBS= Table Option” on page 257

HOST= LIBNAME Statement Option

Specifies an SPD Server machine by node name or IP address, and locates the name server using the SERVICE= value.

Valid in: SPD Server LIBNAME Statement

Note: Option to locate a server host.

Syntax

HOST=*host-name* <SERVICE=*service*>

Required Argument

host-name
the node name or IP address of the server machine.

Optional Argument

service
the name of the service or port number for the server's name server.

Details

This option provides the node name of the server host machine that you want to connect to and locates the port number of the server's name server in the SERVICE= option. If the SERVICE= LIBNAME option is omitted, the server checks the client's **`/etc/services`** file (or its equivalent file) for Spdsname, which is a reserved name for the server's name server.

Example

Specify the server machine Samson. Use the default named service, Spdsname, to obtain the port number of the name server.

```
libname mylib sasspds 'spdsdata'
   host='samson';
```

Specify the server machine Samson and provide the port number of the name server.

```
libname mylib sasspds 'spdsdata'
     host='samson'
     service='5400';
```

Use a Macro Variable to Specify the SPD Server Host. Assign the macro variable SPDSHOST= to the server host Samson so that the LIBNAME statement does not need to specify HOST= LIBNAME statement option to locate Samson.

```
%let spdshost=samson;
   libname mylib sasspds 'spdsdata'
      user='yourid'
      password='swami';
```

IP=YES LIBNAME Statement Option

Invokes SQL implicit pass-through processing for the SPD Server session.

Valid in: SPD Server LIBNAME Statement

Note: Option to specify SQL implicit pass-through.

Syntax

IP=YES

Required Argument

YES
invokes SQL implicit pass-through processing for a single user for the specified SPD Server connection.

Details

The IP=YES LIBNAME statement option specifies an SQL implicit pass-through connection for a single user to a specified domain and server during a given SPD Server session. This is an abbreviated specification which replaces the PASSTHRU= LIBNAME statement option. The IP=YES LIBNAME option draws on other information specified in the LIBNAME statement to establish a server connection. For more information, see "PASSTHRU= LIBNAME Statement Option" on page 167.

Example

```
libname BOAF sasspds 'BOAF'
   server=kaboom.5400
   user='rcnye'
   password='*******'
   IP=YES ;
```

LIBGEN= LIBNAME Statement Option

Configures the SPD Server connection to generate additional domain connections.

Valid in: SPD Server LIBNAME Statement

Default: NO

Applies to: Use of DS2 and FEDSQL languages with SPD Server; referencing more than one domain in explicit SQL pass-through

Syntax

LIBGEN=YES | NO

Required Arguments

NO
connects to the server with a single domain connection.

YES
generates additional domain connections. These additional connections have two purposes:

- enable access to SPD Server with the SAS DS2 procedure and the SAS FEDSQL procedure. The procedures enable you to submit SAS DS2 and SAS FedSQL language statements to SPD Server. This functionality is new with SPD Server 5.3.
- enable you to perform SQL joins across different server domains.

Details

To use the DS2 and FEDSQL procedures with SPD Server, you must have the third maintenance release of SAS 9.4 or later in addition to SPD Server 5.3.

PROC DS2 and PROC FEDSQL cannot be used to create tables in domains that have the attribute BACKUP=YES set in their domain definition. Users of the DS2 and FedSQL languages will get the following error if they try to create a table in a BACKUP=YES domain:

```
ERROR: Creation of aligned tables in BACKUP=YES domains not supported
```

BACKUP=YES on a domain does not prevent table creation with PROC SQL and the DATA step.

An SQL explicit pass-through connection typically specifies a single server domain. The LIBGEN= LIBNAME option enables you to easily reference a second server domain within an explicit pass-through connection. In order to use LIBGEN=, you must first assign LIBNAME statements that point to both domains in your SAS session, specifying LIBGEN=YES in both statements. Then connect to one domain using explicit pass-through. In the EXECUTE statement, identify each domain by using its previously assigned libref.

Examples

Example 1: Using LIBGEN=YES With New Language Procedures

The following example uses LIBGEN=YES to enable use of PROC FEDSQL and PROC DS2 with SPD Server.

```
libname mylib sasspds 'mydomain' host='host.company.com' service=5400
    user='anonymous' libgen=yes;

proc fedsql;
  ...FedSQL language statements ...;
quit;

proc ds2;
  ...DS2 language statements ...;
run;
quit;
```

Example 2: Using LIBGEN= To Support More Than One Domain

The following example uses the LIBGEN=YES LIBNAME option to perform a join between tables in different domains without having to issue an extra EXECUTE connection statement.

```
/* assign a libref to the first domain */
libname path1 sasspds 'domain1'
  server=boxer.5400
  libgen=yes
  ip=YES
  user='anonymous' ;

/* assign a libref to the second domain */
libname path2 sasspds 'domain2'
  server=boxer.5400
  libgen=yes
```

```
  ip=YES
  user='anonymous' ;

/* create a table in each domain */
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 ;
/* make an explicit connection to the first domain */
connect to sasspds (
  dbq='Path1'
  server=boxer.5140
  user='anonymous') ;

/* Use the assigned librefs */
/* to identify the tables in */
/* the EXECUTE statement   */

execute
 (create table table4 as
  select *
  from
   path1.table1 a,
   path2.table2 b
  where a.i = b.i)
by sasspds ;

disconnect from sasspds ;

quit ;
```

See Also

- “Using the SAS DS2 and FedSQL Languages with SPD Server” on page 16
- “Nesting SQL Pass-Through Access” on page 19
- *SAS 9.4 DS2 Language: Reference, Sixth Edition*
- *SAS 9.4 FedSQL Language: Reference, Fifth Edition*
- *SAS 9.4 Procedures Guide: Sixth Edition*

LOCKING= LIBNAME Statement Option

Enables record-level locking for the domain.

Valid in: SPD Server LIBNAME Statement

Default: NO

Restriction: The LOCKING= option cannot be used on domains that have dynamic locking enabled. Specifying LOCKING=YES on a domain that has dynamic locking enabled will not result in an error in the LIBNAME statement. However, subsequent queries and table operations will fail.

Syntax

LOCKING=YES | NO

Required Arguments

YES
enables record-level locking.

NO
disables record-level locking.

Details

By default, SPD Server uses member-level locking to provide data integrity. Member-level locking obtains an exclusive lock on a table on behalf of the user who is updating the table. When a member-level lock is held, only the owner of the lock can access the table. Attempts by others to open the table for Write access fail with a member lock failure error.

Record-level locking locks individual records in a table. The user updating the record has exclusive access to the record, while other users can access other records in the same table. Record-level locking allows multiple clients or parallel operations from the same client to have concurrent Write access to a table, while ensuring the integrity of the record updates. Record-level locking enforces SAS style record-level integrity across multiple clients. Clients are guaranteed that a row will not change during a multi-phased Read or Write operation on the specified row.

When LOCKING=YES is set in an SPD Server LIBNAME statement, all subsequent operations on the server domain will use record-level locking. Operations that affect metadata, such as creating or deleting indexes, renaming variables, and renaming tables require exclusive access to a table, 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. To perform these updates, there must be no other outstanding read or write locks on the table.

Record-level locking must be enabled in the server before a SAS client can use the CNTLEV=REC table option in their SAS program to access server tables.

Record-level locking can have a performance cost if it is enabled on multiple domains. When LOCKING=YES is set, the server connection is made with the single record-level locking proxy process. There is only one record-level locking proxy process per server. All server clients that use record-level locking connections are processed through this 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. In addition, the record-level locking proxy process is a single point of failure for all of these connections.

Record-level locking is not supported for dynamic cluster tables. The server returns an error when an attempt to update a dynamic cluster table while using record-level locking.

Example

The LOCKING= LIBNAME option is specified as follows:

```
libname testrl sasspds 'tmp'
      server=serverNode.port
      user='anonymous'
      locking=YES ;
```

NETCOMP= LIBNAME Statement Option

Sends compressed data in a server network packet.

Valid in: SPD Server LIBNAME Statement

Default: YES

Note: Option for a client and server running on the same UNIX machine.

Syntax

NETCOMP=YES | NO

Required Arguments

YES
sends compressed data in a server network packet.

NO
sends uncompressed data in a server network packet.

Details

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.

Example

This example specifies NETCOMP=NO to turn off compression of the data stream.

```
libname mylib sasspds 'test_area'
   netcomp=no;
```

NEWPASSWORD= LIBNAME Statement Option

Specifies a new password for an SPD Server user.

Valid in: SPD Server LIBNAME Statement

Alias: NEWPASSWD=

Note: Option to identify the server user.

Syntax

NEWPASSWORD='*new-password*'

Required Argument

'*new-password*'
the new password for the server user. The password, visible in a SAS program, is encrypted in the SAS log file.

Details

To change a password with the NEWPASSWORD= LIBNAME option, you specify both the old and new passwords in the LIBNAME statement. When ACL file security is enabled, the server validates the old or new password against its user ID table and changes the password if the validation succeeds.

Example

This example changes server user Receiver's password from WhizBang to Rambo.

```
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 statement option, the user password will not be changed. Follow the operating system procedures to change a user password. Then check with your LDAP server administrator to ensure that the LDAP database also records password changes.

PASSTHRU= LIBNAME Statement Option

Invokes SQL implicit pass-through processing for the SPD Server connection.

Valid in: SPD Server LIBNAME Statement

Interaction: PASSTHRU= provides the same functionality as IP=YES. Use IP=YES instead of PASSTHRU=. IP=YES is easier to use.

Note: Option to specify SQL implicit pass-through.

Syntax

PASSTHRU='dbq='*domain-name*' <*SPD Server-options*>
user='*UserID*' password='*password*' ' ;

Required Arguments

dbq='*domain-name*'
specifies the primary server domain for the SQL pass-through connection. The name that you specify is identical to the server domain name that you used when you made a SAS LIBNAME assignment to **SASSPDS**. Use single or double quotation marks around the specified value.

USER='*user ID*'
required on Windows, but not UNIX. Specifies a user ID in order to access the server's SQL processor. Use single or double quotation marks around the specified value.

PASSWORD='*password*'
required, or use PROMPT=YES, unless USER='anonymous'. Specifies a user ID password to access the server. This value is case sensitive.

Optional Argument

SPD Server-options
one or more server options.

Details

The PASSTHRU= option is the original way that implicit SQL pass-through was made available for SPD Server. This older specification is still supported for backward compatibility.

Example

The following LIBNAME statement shows how the PASSTHRU= option is used:

```
libname BOAF sasspds 'BOAF'
   server=kaboom.5400
   user='rcnye'
   password='*******'

   PASSTHRU='
   dbq="BOAF"
   server=kaboom.5400
   user="rcnye"
   password="*******"' ;
```

PASSWORD= LIBNAME Statement Option

Specifies the password of the server user.

Valid in: SPD Server LIBNAME Statement

Alias: PASSWD=

Note: Option to identify the SPD Server user.

Syntax

PASSWORD='*password*'

Required Argument

password
the case-sensitive password of a server user. The password, visible in a SAS program, is encrypted in the SAS log file.

Details

The password can originate from the SPD Server password database. This type of authentication is referred to as native authentication. Or the password can be an LDAP password, depending on how SPD Server is configured. The server administrator will let you know the password requirements.

When native authentication is configured, you must change your password the first time that you use SPD Server. For more information, see "Changing Server Passwords" on page 17.

Example

The PASSWORD= option is specified as follows:

```
libname mylib sasspds 'spdsdata'
   server=kaboom.5400
   user='spdsuser'
   password='whizbang';
```

PROMPT= LIBNAME Statement Option

Specifies to prompt the SPD Server user for a password.

Valid in: SPD Server LIBNAME Statement

Default: NO

Note: Option to identify the server client.

Syntax

PROMPT=YES | NO

Required Arguments

YES
prompts a user for a password.

NO
suppresses a prompt for a password.

Details

If ACL file security is enabled, the server validates the password against its user ID table.

Example

The PROMPT= LIBNAME option is specified as follows:

```
libname mylib sasspds 'spdsdata'
   user='bigwhig'
   prompt=yes;
```

SCHEMA= LIBNAME Statement Option

Specifies the server domain name.

Valid in: SPD Server LIBNAME Statement

Note: Option to identify the server domain name.

Syntax

SCHEMA="*domain-name*"

Required Argument

DOMAIN-NAME
name of the server domain.

Details

When accessing SPD Server from some applications, you can specify the server domain name in the SCHEMA= option. This enables you to use a named option to specify the domain name. This enables compatibility with the use of the SCHEMA= option in other applications. The following example shows the SCHEMA= option used in the LIBNAME statement:

```
libname currlib sasspds schema="employee" host="mach1"
    serv="12345" user="hrdep1" password="xxxxx"  aclspecial=yes;
```

SERVER= LIBNAME Statement Option

Specifies an SPD Server host machine by its host name and service name.

Valid in: SPD Server LIBNAME Statement

Note: Option to locate a server host.

Syntax

SERVER=*host-name.service-name*

Required Arguments

host-name
the node name of the server host machine.

service-name
the name of a service or the port number for the server's name server

Example

Specify the server host machine Samson and use the default named service Spdsname to obtain the port number of the name server.

```
libname mylib sasspds 'spdsdata'
     server=samson.spdsname;
```

Specify the server host machine Samson and give the port address of the name server.

```
libname mylib sasspds 'spdsdata'
   server=samson.5002;
```

SHARE= LIBNAME Statement Option

Enables enhanced sharing of user proxies.

Valid in: SPD Server LIBNAME Statement

Default: If the SHARE= LIBNAME statement option is not specified, the server enhanced user proxy sharing settings default to the configuration defined by the SHRUSRPRXY= option setting in the server parameter file.

Syntax

SHARE=YES | NO

Required Arguments

YES
enables enhanced sharing of user proxies.

NO
disables enhanced sharing of user proxies.

Details

Enhanced server proxy sharing enables librefs in the same SAS session with different user credentials to share a server user proxy. You use enhanced user proxy sharing to keep the number of concurrent SPDSBASE process resources from growing too large. A large number of concurrent SPDSBASE processes can create system resource allocation issues in some server environments.

STARTOBS= LIBNAME Statement Option

Specifies the start row number in a user-defined range for processing.

Valid in: SPD Server LIBNAME Statement

Syntax

STARTOBS=*n*

Required Argument

n
the number of the start row.

Details

By default, the server processes the entire table unless the user specifies a range of rows with the LIBNAME statement options STARTOBS= and ENDOBS=. If the ENDOBS= LIBNAME statement option is used without the STARTOBS= LIBNAME statement option, the implied value of STARTOBS=LIBNAME statement option is 1. When both options are used together, the value of the STARTOBS= LIBNAME statement option must be less than the value of the ENDOBS= LIBNAME statement option.

In contrast to the Base SAS software options FIRSTOBS= and OBS=, STARTOBS= and ENDOBS= can be used for WHERE clause processing in addition to table input operations.

Example

Specify for the server to process only row numbers 200–500 while the LIBNAME is active.

```
libname mydatalib sasspds 'conversion_area'
   server=husky.5105
   user='siteusr1'
   prompt=yes
   startobs=200
   endobs=500;
```

See Also

SPD Server macro variables:

- “SPDSSOBS Macro Variable” on page 235

SPD Server table options:

- “STARTOBS= Table Option” on page 267

TEMP= LIBNAME Statement Option

Creates a temporary server domain for this LIBNAME assignment.

Valid in: SPD Server LIBNAME Statement

Default: NO

Syntax

TEMP=YES | NO

Required Arguments

YES
creates a temporary server domain for the LIBNAME assignment.

NO
does not create a temporary server domain.

Details

Use this option to create a temporary server domain that exists for the duration of the LIBNAME assignment. A temporary server domain is similar to the SAS Work library. When you specify TEMP=YES in the LIBNAME statement, any data objects, tables, catalogs, or utility files that are created in the referenced domain are automatically deleted when you end the SAS session. The temporary domain is created as a subdirectory of the pathname defined for the server domain by administrators and is later deleted by the server.

If you specify USER as the libref in a LIBNAME statement that specifies TEMP=YES, then both table references that use a one-level name (MyTable) and table references that specify the USER libref (User.MyTable) are created in the temporary server domain.

If you want some but not all one-level table references to be processed by the server, use the USER= system option to make the temporary domain assignment instead of using the USER libref. For more information, see"Example 2: Specify Other Libref with TEMP=YES" on page 174.

Examples

Example 1: Specify Libref USER and TEMP=YES

```
libname user sasspds 'mydomain'
   server=kaboom.5191
   user='siteusr1'
   prompt=yes
   temp=yes ;
```

When the libref USER is specified with TEMP=YES, all one-level table references (MyTable) and table references qualified by the libref USER (User.MyTable) are processed by and temporarily stored on the server.

Example 2: Specify Other Libref with TEMP=YES

```
libname mylib sasspds 'mydomain'
   server=kaboom.5191
   user='siteusr1'
   prompt=yes
   temp=yes ;
```

When libref MyLib is specified with TEMP=YES, only table references that are qualified by the libref (MyLib.MyTable) are processed by and temporarily stored on the server. One-level table references are created as SAS tables in the Work library.

To make a particular one-level table reference to go to server temporary storage, set the USER= system option as follows:

```
option user=MyLib;
data MyTable;
 set OtherTable;
run;
options user='';
```

The OPTIONS statement enables USER access via libref MyLib. Table MyTable is temporarily created in domain MyDomain. Submitting the USER= system option without a value clears the USER library assignment.

TRUNCWARN= LIBNAME Statement Option

Suppresses hard failure on NLS transcoding overflow and character mapping errors.

Valid in: SPD Server LIBNAME Statement

Default: NO

Syntax

TRUNCWARN=YES | NO

Required Arguments

YES
can cause hard failure on NLS transcoding overflow and character mapping errors.

NO
suppresses hard failure on NLS transcoding overflow and character mapping errors.

Details

When you are using the TRUNCWARN=YES LIBNAME statement 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 table close time if overflow occurs, the data overflow is lost.

UNIXDOMAIN= LIBNAME Statement Option

Specifies to use UNIX domain sockets for data transfers between the client and SPD Server.

Valid in: SPD Server LIBNAME Statement

Default: YES

Note: Option for a client and server running on the same UNIX machine.

Syntax

UNIXDOMAIN=YES | NO

Required Arguments

NO
uses TCP/IP for data transfer between the client and SPD Server.

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

Details

A UNIX domain socket is an inter-process communication mechanism that provides for exchanging data between processes executing on the same host operating system. UNIX domain sockets can offer better performance than inter-process TCP/IP socket communication when both the client and SPD Server reside on the same host.

USER= LIBNAME Statement Option

Specifies an SPD Server user ID.

Valid in: SPD Server LIBNAME Statement

Note: This option is required to connect to the server when ACL file security is enabled. If only UNIX file security is configured for the server, it is not necessary.

See: "Understanding User Validation and Authorization" on page 14

Syntax

USER='*user-name*'

Required Argument

'***user-name***'
a user ID that is registered in the server's password database. This user ID is given to you by the server administrator.

Example

This example connects to the server with the user ID MySpdsId.

```
libname mylib sasspds 'spdsdata' host="husky" service="5600"
  user='myspdsid' prompt=yes;
```

Chapter 18

Explicit Pass-Through SQL Statements

SPD Server SQL Explicit Pass-Through Statements

This chapter provides reference information about the SQL statements necessary to establish an SQL explicit pass-through connection to the SPD Server SQL processor. For usage information, see "Connect to SPD Server with Explicit SQL Pass-Through" on page 17.

Dictionary

CONNECT Statement

Specifies the SAS engine that provides SQL explicit pass-through access.

Valid in: SPD Server

Syntax

CONNECT TO *engine-name* <AS *alias*> (*connection-argument(s)*);

Required Arguments

engine-name
specifies the name of the engine. There are two valid values:

SASSPDS specifies to obtain SQL pass-through to the server's SQL processor from PROC SQL in SAS. Most SQL pass-through connections will use this value.

SPDSENG specifies to access SQL pass-through from a secondary connection (that is, a connection that is held by the server's SQL processor).

Note: SPDSENG is the engine that you use to reference an SPD Server from within an existing SQL explicit pass-through connection. For more information about nesting connections, see "Nesting SQL Pass-Through Access" on page 19.

TIP If you would prefer not to use the SPDSENG engine to reference a server, you can use the LIBGEN=YES option. Libraries with the LIBGEN=YES option are automatically available in SQL environments. For more information, see "LIBGEN= LIBNAME Statement Option" on page 162.

connection-argument(s)

identify the SPD server domain and name server. The following *connection-args* arguments are for the SPD Server engines, SASSPDS and SPDSENG. Submit them as keyword=*value* pairs.

Note: Some *connection-arguments* are required and some are optional.

USER=*server-user-ID*

specifies a server user ID to access the server's SQL processor. Enclose the value in single or double quotation marks.

Note: USER= option is required on Windows. On UNIX, it is not necessary to specify USER= in a CONNECT statement because the server assumes the UNIX user ID.

PASSWORD=*password*
PASSWD=*password*

specifies the password associated with the server user ID. This value is case sensitive. You should not specify a password in a text file that another user can view. You should use this argument in a batch job that is protected by file system permissions, which prohibit other users from reading the text file.

Note: PASSWORD= option is required unless you use PROMPT=YES or unless USER='anonymous'.

HOST=*host-name*

specifies a node name or an IP address for the SPD Server host server. Enclose the string in single or double quotation marks. If you do not specify a value, the server uses the current value of the SAS macro variable SPDSHOST= to determine the node name.

Note: The HOST= option is optional.

SERVICE=*port-number*

specifies the network address (port number) for the name server. Enclose the value in single or double quotation marks. If you do not specify a port number for the name server, SPD Server determines the network address from the named service spdsname in the/etc/services file.

PROMPT=YES

specifies to issue a password prompt to access the server's SQL processor. The prompter is case sensitive.

Note: The PROMPT=YES option is required unless you use PASSWORD= or unless USER='anonymous'.

Optional Argument

AS ***alias***

specifies an alias for the connection. When you specify an alias to identify the connection, use a string that is not enclosed in quotation marks. Refer to this name in subsequent SQL explicit pass-through statements.

Note: For the alias, you must specify the connection that executes the statement.

The following two examples show how to use an alias:

```
execute(...) by alias

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

Example: Using the Explicit Pass-Through Facility

In this example, you issue a CONNECT statement to connect from a SAS session to the server SQL processor. After the connection is made, the first EXECUTE statement creates a table named EMPLOYEE_INFO with three columns: EMPLOYEE_NO, EMPLOYEE_NAME, and ANNUAL_SALARY. The second EXECUTE statement inserts a row into the table. The subsequent SELECT FROM CONNECTION TO statement retrieves all of the records from the new EMPLOYEE_INFO table. The DISCONNECT statement terminates the connection.

```
proc sql;
connect to sasspds
    (dbq='mydomain'
     host='workstation1'
     service='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', 10000)) by sasspds;

select * from connection to sasspds
(select * from employee_info);

disconnect from sasspds;
quit;
```

CONNECTION TO Statement

Enables you to send SELECT queries to the SPD Server SQL processor using explicit SQL pass-through.

Valid in: SPD Server

Syntax

CONNECTION TO [*engine-name* | *alias*] (*SQL-query*);

Required Arguments

engine-name* or *alias
specify SASSPDS to obtain a pass-through connection to the server's SQL processor from PROC SQL. Specify SPDSENG to obtain a secondary pass-through connection from the server's SQL processor to another server.

Note: SPDSENG is the engine that you use to reference an SPD Server from within an existing server SQL explicit pass-through connection. For more information about nesting connections, see "Nesting SQL Pass-Through Access" on page 19.

alias
specifies the alias that was used in the CONNECT statement.

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

Details

CONNECTION TO is an SQL explicit pass-through component that you can use in the FROM clause of a PROC SQL SELECT statement to connect to SPD Server. The CONNECTION TO component enables you to make SQL explicit pass-through queries for server 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.

SPD Server SQL does not support the full SELECT functionality of SAS SQL. For more information, see Chapter 8, "Understanding the SPD Server SQL Processor," on page 65.

For SELECT syntax, see *SAS SQL Procedure User's Guide*.

DISCONNECT Statement

Terminates the SQL explicit pass-through session with the server SQL processor.

Valid in: SPD Server

Syntax

DISCONNECT FROM [*engine-name* | *alias*];

Required Arguments

engine-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.

Details

When you no longer need the PROC SQL connection to the server, you must disconnect from the server SQL processor. You are automatically disconnected when you exit PROC SQL. However, you can explicitly disconnect by using the DISCONNECT statement

EXECUTE Statement

Submits SQL statements that do not return a result set directly to the server.

Valid in: SPD Server

Syntax

EXECUTE (SQL-statement) BY [*engine-name* | *alias*];

Required Arguments

(*SQL-statement*)
a valid SQL statement for the SPD Server SQL processor. This argument is required and must be enclosed within parentheses.

engine-name
specifies the SASSPDS engine. The *engine* value must be preceded by the keyword BY. You must specify either SASSPDS or the *alias* from your CONNECT statement.

alias
(optional) specifies an alias that can be used in the CONNECT statement. If you did not specify an *alias* in your CONNECT statement, then you must specify SASSPDS.

Details

Before you use the EXECUTE statement, you must establish a connection to the server by using the CONNECT statement. Use the EXECUTE statement to submit SAS SQL statements that do not return a result set to the SQL processor. You cannot submit a SELECT statement in the EXECUTE statement. Use the PROC SQL SELECT statement with the FROM CONNECTION statement to submit SELECT queries directly to the server's SQL processor. For information about the SAS SQL statements, see *SAS SQL Procedure User's Guide*.

SPD Server SQL functions a little bit differently than SAS SQL. See Chapter 8, "Understanding the SPD Server SQL Processor," on page 65 for information about SPD Server SQL requirements before using the EXECUTE statement.

You can also submit the SPD Server CREATE VIEW, COPY TABLE, LOAD TABLE, BEGIN| END ASYNCHRONOUS OPERATION, LIBREF, and RESET statements in the EXECUTE statement.

See Also

SQL Statements:

- "BEGIN ASYNC OPERATION Statement" on page 183
- "CREATE VIEW Statement" on page 188
- "CONNECT Statement" on page 177
- "CONNECTION TO Statement" on page 179
- "COPY TABLE Statement" on page 187
- "CREATE VIEW Statement" on page 188

Chapter 19

SPD Server SQL Statement Additions

SPD Server SQL Statement Additions

SPD Server SQL supports several SQL statements that are not part of the SAS SQL language. These additional statements enable data management functionality unique to SPD Server. The statements are available only through explicit SQL pass-through.

Note: The SAS SQL procedure provides a RESET statement. When the RESET statement is specified in the EXECUTE statement, the functionality of the SPD Server SQL statement supersedes the RESET statement in PROC SQL.

Dictionary

BEGIN ASYNC OPERATION Statement

Marks the beginning of a block of statements intended for asynchronous, parallel execution.

Valid in: SPD Server
Explicit SQL pass-through facility

Requirement: Must be used in conjunction with the “END ASYNC OPERATION Statement” on page 190.Optionally used with the “LIBREF Statement” on page 190.

Syntax

BEGIN ASYNC OPERATION

Without Arguments

```
execute(begin async operation) by sasspds;
```

Details

You can maximize the performance of certain SQL statements by specifying to execute them asynchronously, in parallel. SPD Server provides the BEGIN ASYNC OPERATION and END ASYNC OPERATION statements to delimit the block of statements that are intended for asynchronous, parallel execution.

Not all SQL statements are candidates for submission within the same ASYNC block. SPD Server initiates thread execution according to the order of the statements in the block. However, there is no way to guarantee that a statement will finish before another statement executes. Therefore, you should avoid submitting statements that depend on another statement to complete within the same block. Examples of statements that should not be executed within the same block are CREATE TABLE and CREATE INDEX. There is no guarantee that the CREATE TABLE statement will complete before the CREATE INDEX statement begins.

The block approach is useful for operations such as creating multiple tables in parallel and for creating multiple indexes on one or more existing tables in parallel.

If you plan to submit statements to more than one domain within a block, then you must re-create the connection made with the CONNECT TO statement within the block using the LIBREF statement. You must also issue a LIBREF statement to connect to the second domain within the block. The connection made with the CONNECT TO statement does not extend to the block. Conversely, any LIBREF statement that you issue inside the ASYNC block does not extend outside of the ASYNC block. To function correctly, a LIBREF statement that is specified inside a block must precede the first SQL statement that references it.For an example of how the LIBREF statement is used, see "Example 2: Using SQL Options in an ASYNC Block Statement" on page 185.

If you plan to specify an SQL RESET statement option for any statement in a block, the option must be set globally for all EXECUTE statements in the block. This is done by issuing the EXECUTE statement that resets the option before specifying the BEGIN ASYNC OPERATION statement.

Examples

Example 1: Creating Tables and Indexes in Parallel

This example shows how to create multiple tables and multiple indexes in parallel. The CREATE TABLE statements and the CREATE INDEX statements are submitted within separate asynchronous blocks. The END ASYNC statement in the first block serves as a synchronization point to ensure that all the tables are created before the second ASYNC statement block begins.

```
proc sql;
            connect to sasspds
              (dbq="path1"
               server=host.port
               user='siteusr1'
               password='mypasswd');

            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 indexes in a 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;
```

Example 2: Using SQL Options in an ASYNC Block Statement

This example shows how to submit SQL planner options in an asynchronous block. You must set the options globally for all EXECUTE statements in the ASYNC block. The options must be set before the BEGIN ASYNC OPERATION statement.

This code sample also shows how to use the LIBREF statement. When referencing more than one domain in an ASYNC block, you must specify a LIBREF statement for each of the domains in the asynchronous block. Note that the LIBGEN= option is set in the LIBNAME statement as well.

```
libname path1 sasspds ... libgen=yes;
    libname path2 sasspds ... libgen=yes;

proc sql;
    connect to sasspds
       (dbq='path1'
        host='hostname'
        service='spdsname'
        user='siteusr1')
        password='mypasswd';

    execute(reset noexec _method)
      by sasspds;

    execute(begin async operation)
      by sasspds;

    execute(libref path1
      engopt='dbq="path1"
              host='hostname'
              service='spdsname'
              user='siteusr1'
              password='mypasswd')
              by sasspds;

     execute(libref path2
       engopt='dbq="path2"
               host='hostname'
               service='spdsname'
               user='siteusr1'
               password='mypasswd')
               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;
```

See Also

Statements:

- "END ASYNC OPERATION Statement" on page 190
- "LIBREF Statement" on page 190

COPY TABLE Statement

Copies an SPD Server table.

Valid in: SPD Server

Explicit SQL pass-through facility

Requirements: When tables are copied between domains, the source and destination domains must have the same backup setting. That is, both domains must have either BACKUP=YES or BACKUP=NO in their definition. When domains have different backup settings, you must use PROC COPY to copy a table between the domains.

The COPY TABLE statement requires local direct access to the source and destination tables from the machine that the server is running on. SPD Server does not support the COPY TABLE statement for use with tables in a Hadoop domain.

Syntax

COPY TABLE *new-table-name* FROM *old-table-name* [WITHOUT INDEXES]
[ORDER BY *column-name* [ASC | DESC] [',' *column-name* [ASC | DESC]]] ';'

Required Arguments

new-table-name
specifies the name of the new server table.

old-table-name
specifies the name of the source server table.

Optional Arguments

ORDER BY *column-name* [ASC |DESC][, *column-name* [ASC |DESC]]
optional: sorts the data in the new table by one or more columns, setting the data in the columns in ascending or descending order.

WITHOUT INDEXES
optional: suppresses creation of indexes.

Details

Use COPY TABLE to copy an existing SPD Server table from one server domain to another. You can copy the table with or without indexes. COPY TABLE offers the same parallel table and index I/O and overlapped input as the LOAD TABLE statement.

(Optional) You can optionally specify a new sort order.

Comparisons

Use COPY TABLE when you want to duplicate an SPD Server table in its entirety. Use LOAD TABLE when you want to create a new table that contains a subset of the columns or data from the source SPD Server table.

Examples

Example 1: Copy a Table with and without Indexes

The following example creates two new tables: T_NEW and 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;
```

Example 2: Copy a Table and Order Its Columns

COPY TABLE does not support all of the options of PROC SORT. However, you can achieve substantial performance gains when you create ordered tables by using the COPY TABLE statement with an ORDER BY clause when appropriate. This 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 TABLE ORDER BY follows the typical SQL ORDER BY clause, but the column identifiers that you can specify are restricted. You can specify only actual table columns when you use the COPY TABLE ORDER BY clause.

```
execute(copy table t_new
from t_old
order by x, y desc, z asc)
by sasspds;
```

CREATE VIEW Statement

Creates a view of SPD Server tables from a query expression. The view can be materialized in a table.

Valid in: SPD Server
Explicit SQL pass-through facility

Restriction: An SPD Server view can reference only SPD Server tables.

Syntax

CREATE [MATERIALIZED] VIEW *view-name* AS SELECT *query-expression*';'

Required Arguments

view-name
specifies a name for the view.

query-expression
defines the columns in the view. The columns can originate from one or more SPD Server tables.

Optional Argument

MATERIALIZED
specifies to copy the contents of the view into a temporary table.

Details

When you create an SQL view of SPD Server tables, a view file is created in the specified domain with the name *view-name*.view.0.0.0.spds9. The view creator is the only one who has access to the view, until an ACL is created that grants other users access. Then, users who are using explicit SQL pass-through to access the view can use the view as they would use a table in SQL queries. Users who access the view through implicit SQL pass-through or by using a SAS DATA step must have direct access to the component tables that are referenced in the view in addition to having access to the view in order to use it.

Including the keyword MATERIALIZED in the CREATE VIEW statement specifies to create the view as a materialized view. When you create a materialized view, an additional SPD Server table is created in the same domain as the standard SQL view file. This table contains a copy of the data that was available from the view when the view was created. The materialization process can add substantial time to the execution of a CREATE VIEW statement. 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.

As long as the data from the component tables does not change, the materialized view returns the results from the temporary table when the view is referenced in an SQL statement. When any of the component tables that make up the view are modified, the materialized view recomputes the results the next time the view is referenced and refreshes the temporary table with the new results. The temporary results table for a materialized view exists for as long as the view exists. When the owner deletes or drops a view, the temporary results table is also deleted. It is not necessary to specify the MATERIALIZED keyword in the DROP VIEW statement.

A materialized view table is accessed with SQL statements. A materialized view table cannot be accessed by using PROC DATASETS or other SAS procedures.

For a regular SPD Server SQL view, the results are computed each time the view is referenced in a subsequent 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 regular view. If the results table produced by the view is relatively small in comparison with the input tables, the execution time for queries that use a materialized view might be a few seconds versus several minutes for a standard view.

If a view is being referenced at least twice before any updates occur, then the materialized view can provide superior performance. If the input tables are frequently updated in comparison to how often the view is referenced, a standard view is probably more efficient.

Examples

Example 1: Creating a Regular SPD Server SQL View

```
connect to sasspds(dbq='temp' user='Stan');
execute(create view WinterSpring as
select * from SpringSales, WinterSales
where SpringSales.id = WinterSales.id) by sasspds;
```

Example 2: Creating a Materialized SPD Server SQL View

```
connect to sasspds(dbq='temp' user='Stan');
execute(create materialized View WinterSpringTable as
select * from SpringSales, WinterSales
where SpringSales.id = WinterSales.id) by sasspds;
```

END ASYNC OPERATION Statement

Marks the end of a block of statements intended for asynchronous, parallel execution.

Valid in: SPD Server
Explicit SQL pass-through facility

Requirement: Must be used in conjunction with the "BEGIN ASYNC OPERATION Statement" on page 183.

Syntax

END ASYNC OPERATION

Without Arguments

```
execute (end async operation) by sasspds;
```

Details

You can maximize the performance of certain SQL statements by specifying to execute them asynchronously, in parallel. SPD Server provides the BEGIN ASYNC OPERATION and END ASYNC OPERATION statements to delimit the block of statements that are intended for asynchronous, parallel execution. The BEGIN ASYNC OPERATION statement marks the beginning of the statement block. The END ASYNC OPERATION statement marks the end of the statement block.

For examples of how the BEGIN ASYNC OPERATION and END ASYNC OPERATION statements are used, see "Creating Asynchronous Operation Blocks" in X.

LIBREF Statement

Creates a connection to an SPD Server domain within an EXECUTE statement.

Valid in: SPD Server
Explicit SQL pass-through facility

Requirement: The domain specified in the LIBREF statement must be on the same network as the domain specified in the CONNECT TO statement.

Syntax

LIBREF *libref-name* ENGOPTS='*connection-string*'

Arguments

libref-name
specifies a logical name for the connection. This name can be used a domain qualifier in subsequent explicit pass-through requests.

ENGOPTS='*connection-string*'
specifies SPD Server connection parameters:

DBQ=*domain*
specifies the name of the SPD Server domain to which you want to connect. The domain must have been previously defined in a libnames.parm file.

SERVER=*host.port*
specifies the name of the SPD Server Name Server host computer and port number.

USER=*userID*
specifies an SPD Server user ID.

PASSWORD=*password*
specifies the password associated with the user ID. The PASSWORD parameter is not required when the ANONYMOUS user ID is used.

Details

The statements submitted within an SQL EXECUTE statement apply to the SPD Server domain specified in the CONNECT TO statement. The LIBREF statement is provided to enable you to reference another domain within the SQL explicit pass-through session.

The domain specified in the LIBREF statement must be on the same network as the domain specified in the CONNECT TO statement. For information to establish a connection to a domain that is in a different network, see "Nesting SQL Pass-Through Access" on page 19.

The LIBREF statement is often used in statement blocks that are submitted to the SQL processor within the BEGIN ASYNC OPERATION and END ASYNC OPERATION statements. When used within a block, a LIBREF statement must be issued for both the CONNECT TO domain and the new domain, so that you can identify tables from the domains using two-part names. For an example of how the LIBREF= statement is used, see "Example 2: Using SQL Options in an ASYNC Block Statement" on page 185.

LOAD TABLE Statement

Creates a new SPD Server table from an existing SPD Server table by using a SELECT clause.

Valid in: SPD Server
Explicit SQL pass-through facility

Requirements: When data is loaded between domains, the source and destination domains must have the same backup setting. That is, both domains must have BACKUP=YES or BACKUP=NO in their definition. When domains have different backup settings, you must use CREATE TABLE AS to create a table from an existing server table.

The LOAD TABLE statement requires local direct access to the source and destination tables from the machine that the server is running on. SPD Server does not support the LOAD TABLE statement for use with tables in a Hadoop domain.

Syntax

LOAD TABLE *new-table-name* [WITH *index-name* ON (*column-name*)
[',' WITH *index-name* ON (*column-name*)]]
AS SELECT *select-list* FROM *old-table-name*
[WHERE *sql-expression*]';'

Required Arguments

new-table-name
specifies the name of the new server table.

old-table-name
specifies the name of the source server table.

select-list
defines the columns for the new table. Valid values are one or more column names or an * (asterisk), which indicates all columns from the table. All characteristics of the columns in the SELECT list are preserved and become permanent attributes of the new table's column definitions.

Optional Arguments

WHERE *sql-expression*
specifies an sql-expression that selects a subset of the data from the old table for the new table.

WITH *index-name* ON (*column-name*) [',' WITH *index-name* ON (*column-name*)|
creates indexes on one or more columns in the new table.

Details

Use the LOAD TABLE statement to create a table from an existing table with one or more indexes using a single statement. The SELECT statement enables you to use a subset of the columns from the source table in the new table. The WHERE statement enables you to subset the data. In general, the LOAD TABLE statement is faster than a corresponding CREATE TABLE and CREATE INDEX statement pair, because it builds the table and one or more associated indexes asynchronously by using parallel processing.

Comparisons

Use LOAD TABLE when you want to create a new table that contains a subset of the columns or data from an existing SPD Server table. Use COPY TABLE when you want to duplicate the source table in its entirety.

Examples

Example 1: Create a New Table with Multiple Indexes

This example creates a server table named CarLoad that contains a subset of the data from an SPD Server table named Cars. The creation of table CarLoad and its indexes occurs in parallel.

```
execute(
load table carload with
index origin
on (origin),
index mpg
on (mpg)
as select *
from cars)
by sasspds;
```

Example 2: Creating Multiple Server Tables from a Single Server Table

In this example, multiple EXECUTE statements are issued to create a table for individual U.S. states from a global table called State that contains many states. The first EXECUTE statement uses LOAD TABLE to create table State_AL (Alabama), and creates an index on the County column. The structure of the table State_AL and the data in the table both come from the global table State. The data in State_AL is the subset of all records from the State table in which the column value equals 'AL'. The LOAD TABLE statement creates a table for all states (Alabama through Wyoming). The table for each state is indexed by county and mirrors 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;
```

Chapter 20

SPD Server Functions, Formats, and Informats

Functions

SPD Server supports the use of SAS functions in the DATA step and SAS procedures. Most functions also work in SQL pass-through statements. See *SAS Functions and CALL Routines: Reference* for information about the functions.

Note that DATE, INT, LEFT, RIGHT, LENGTH, and TRIM are reserved keywords. Therefore, they must be preceded by a backslash in server SQL queries:

```
select \date() from t
```

Introduction to Formats and Informats

SPD Server supports some of the more commonly used SAS formats and informats. Use these in your SAS DATA step code and in your SQL code when you want the server to associate a table column with a specific format.

A general reminder about formats: A format is applied to column values when they are written out. Informats are applied as the column values are being read.

For more information about the supported formats, see *SAS Formats and Informats: Reference*.

Formats

List of Formats

Format	Description
$	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	SPD Server 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.

Format	Description
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 an '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).

Format	Description
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 an '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 that 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 a server table named Sales, which has a single numeric column Salesite. The Salesite column stores a value that represents the total sales for a given site. Using SQL explicit pass-through, create a new table 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;

disconnect from sasspds;
```

```
quit;
```

User-Defined Formats

To create and access user-defined formats in SPD Server, you must do the following:

- The user-defined formats must be created on the architecture where they will be used. For example, if the format is to be used on a Windows server, the format must be created on a Windows machine.
- The user-defined formats must be created in a domain called **formats**.
- You must make an SPD Server LIBNAME assignment to the domain called **formats**.
- The LIBNAME assignment **cannot** be a temporary assignment that uses the TEMP=YES LIBNAME option. All user-defined formats must be in the same physical location that is defined by the formats domain.
- The LIBNAME assignment must use the LOCKING=YES setting. The LOCKING=YES setting enables SPD Server to synchronize concurrent read calls to the user-defined formats.
- You must set the FMTSEARCH= system option in the SAS session so that SAS can also find the formats to verify them as in this example:

  ```
  options fmtsearch=(formats);
  ```

  SPD Server does not require that your data and your user-defined formats reside in the same domain. The server will always look in the domain that is named **formats** when the operating system encounters any call for user-defined formats.

The following example code shows how user-defined formats can be referenced:

- in parallel GROUP BY statements
- in a WHERE clause within a PROC PRINT step
- in a WHERE clause referenced in explicit SQL

The example includes the creation of the user-defined formats and a test table. The example also provides changes to configuration files (spdsserv.parm and libnames.parm) that normally would be made by your SPD Server administrator. For more information about configuring spdsserv.parm files, see *SAS Scalable Performance Data Server: Administrator's Guide*.

The example uses the following spdsserv.parm file:

```
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=5400;
```

The example uses the following libnames.parm file:

```
LIBNAME=tmp pathname=c:\temp;
LIBNAME=formats pathname=c:\data\formats;
```

Here is the complete example code with comments:

```
%let domain=tmp;
%let host=d8488;
%let serv=5400;

/* locking=YES must be specified when using */
/* options fmtsearch=(formats); */

libname &domain sasspds "&domain"
  host="&host"
  serv="&serv"
  user='anonymous'
  password='mypwd'
  IP=YES;

libname formats sasspds 'formats'
  host="&host"
  serv="&serv"
  user='anonymous'
  locking=YES;

options fmtsearch=(formats);

proc datasets nolist
  lib=formats
  memtype=catalog;

delete formats;
quit;

/* Create AGEGRP and $GENDER formats. */

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 and $GENDER formats.       */
```

```
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 Standard 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;

title 'Format in WHERE clause example';
proc print data=temp;
run;

/* This explicit SQL executes a WHERE clause that */
/* references a user-defined format.              */

title 'Explicit SQL with a User-Defined Format in a WHERE Clause';

proc sql;
connect to sasspds
  (dbq="&domain"
   serv="&serv"
   host="&host"
   user="anonymous");

select * from connection to sasspds
  (select * from fmttest where put(age, AGEGRP.) eq 'Child');
quit;
```

Informats

Informat	Description
$	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.
$VARYING	Reads varying length values.
$SASNAME	
BEST	SPD Server 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).

Informat	Description
E	Reads scientific notation.
F	Reads scientific notation.
HEX	Converts hexadecimal positive binary values to fixed-point 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 record 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 day 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 21

SPD Server Macro Variables

Overview of SPD Server Macro Variables

Macro variables operate similarly to LIBNAME options and table options. But they have an advantage because they apply globally. That is, their value remains constant until explicitly changed. These variables can be used by the SPD Server SAS client to service the behavior of the client and server.

To set a macro variable to YES, submit the following statement:

```
%let macro-variable-name=YES;
```

CAUTION:

Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

After you have set the value of a macro variable, you can verify its value by submitting a PUT statement:

```
%put &=macro-variable-name;
```

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 server macro variables, use the PROC SPDO SPDSMACS statement. See "SPDSMAC Statement" in *SAS Scalable Performance Data Server: Administrator's Guide*.

SPDSUSDS Reserved Macro Variable

When the SPDSUSAV macro variable or UNIQUESAVE= table option is set to YES and an Append or Insert operation is performed on a table that has unique indexes, SPD Server creates a hidden table to store any rows that are rejected because they have duplicate key values. The SPDSUSDS reserved macro variable references this hidden table.

The following example shows how SPDSUSDS is used to access the hidden table.

```
data employee.names1 (index=(id/unique ));
   input name $ id ;
   list;
   datalines;
Jill 4344
Jack 5589
Jim  8888
Sam  3334
run;

data employee.names2;
   input name $ id;
   list;
```

```
   datalines;
Jack  8443
Ann   3334
Sam   8756
Susan 5321
run;

%let SPDSUSAV=YES;
proc append base=employee.names1 data=employee.names2;
run;
proc print data=&spdsusds;
run;
```

After the update, the log shows the following:

```
NOTE: Appending EMPLOYEE.NAMES2 to EMPLOYEE.NAMES1.
NOTE: There were 4 observations read from the data set EMPLOYEE.NAMES2.
NOTE: 3 observations added.
NOTE: The data set EMPLOYEE.NAMES1 has 7 observations and 2 variables.
WARNING: Duplicate values not allowed on index id for file EMPLOYEE.NAMES1.
(Occurred 1 times.)
NOTE: Duplicate records have been stored in file EMPLOYEE._7296DDD.
```

The &SPDSUSDS in the PROC PRINT statement refers to the file in the last NOTE.

Functional List of SPD Server Macro Variables

The following table lists the macro variables by their purpose.

Table 21.1 *Macro Variables by Purpose*

Purpose	Macro Variable	Description
Affect disk space	"SPDSCMPF Macro Variable" on page 216	Specifies the amount of growth space, sized in bytes, to be added to a compressed data block.
	"SPDSDCMP Macro Variable" on page 217	Compresses SPD Server tables that are stored on disk.
	"SPDSIASY Macro Variable" on page 224	Specifies whether to create indexes in parallel when creating multiple indexes on an SPD Server table.
	"SPDSSIZE Macro Variable" on page 233	Specifies the size of an SPD Server table partition.
Affect sorts	"SPDSBNEQ Macro Variable" on page 213	Specifies the output order of table rows that have identical values in the BY column.

Purpose	Macro Variable	Description
	"SPDSBSRT Macro Variable" on page 213	Configures SPD Server's sorting behavior when it encounters a BY-clause and there is no index available.
	"SPDSNBIX Macro Variable" on page 227	Suppresses index use for BY sorts.
	"SPDSSTAG Macro Variable" on page 236	Specifies whether to use non-tagged or tagged sorting for PROC SORT or BY processing.
Affect WHERE clause evaluations	"SPDSTCNT Macro Variable" on page 237	Specifies the number of threads that you want to use during WHERE clause evaluations.
	"SPDSEV1T Macro Variable" on page 219	Specifies whether data returned from SPD Server WHERE clause evaluations should be in strict row order.
	"SPDSEV2T Macro Variable" on page 221	Specifies whether the data returned from WHERE clause evaluations should be in strict row order.
	"SPDSWDEB Macro Variable" on page 241	Specifies whether the WHERE clause planner WHINIT, when evaluating a WHERE expression, should display a summary of the execution plan.
	"SPDSIRAT Macro Variable" on page 226	Specifies whether to perform segment candidate pre-evaluation when performing WHERE clause processing with indexes.
	"SPDSNIDX Macro Variable" on page 228	Specifies whether to use the table's indexes when processing WHERE clauses.
	"SPDSWCST Macro Variable" on page 240	Specifies whether to use dynamic WHERE clause costing.
	"SPDSWSEQ Macro Variable" on page 241	Overrides WHERE clause costing and forces a global EVAL3 or EVAL4 strategy.
Client and server on same UNIX machine	"SPDSCOMP Macro Variable" on page 216	Specifies to compress the data when sending a data packet through the network.
Enhance performance	"SPDSCLJX Macro Variable" on page 215	Affects the SAS SQL planner when joining a SAS table with an indexed SPD Server table.

Purpose	Macro Variable	Description
	"SPDSNETP Macro Variable" on page 227	Sizes buffers in server memory for the network data packet.
Handle duplicates	"SPDSAUNQ Macro Variable" on page 212	Specifies whether to cancel an append or insert to a table. Use this macro variable if the table has a unique index and the new rows would violate the index uniqueness.
	"SPDSSADD Macro Variable" on page 233	Specifies whether SPD Server appends tables by transferring a single row at a time synchronously, or by transferring multiple rows asynchronously (block row appends).
	"SPDSUSAV Macro Variable" on page 237	Specifies whether to save rows with nonunique (rejected) keys to a separate SAS table.
Logging	"SPDSIPDB Macro Variable" on page 225	Specifies whether to include SQL implicit pass-through code or error messages in the SPD Server log.
Security	"SPDSRSSL Macro Variable" on page 232	Specifies to make a secure sockets connection to SPD Server.
Miscellaneous function	"SPDSEINT Macro Variable" on page 218	Specifies how SPD Server responds to network disconnects during SQL explicit pass-through EXECUTE() statements.
	"SPDSEOBS Macro Variable" on page 219	Specifies the number of the last row for a user-defined range that you want to process in a table.
	"SPDSFSAV Macro Variable" on page 222	Specifies whether you want to retain table data if the SPD Server table creation process terminates abnormally.
	"SPDSHOST Macro Variable" on page 223	Assigns an SPD Server host name.
	"SPDSSQLR Macro Variable" on page 236	Specifies an SQL reset option in implicit pass-through.
	"SPDSSOBS Macro Variable" on page 235	Specifies the number of the starting row in a user-defined range of a table
	"SPDSVERB Macro Variable" on page 239	Specifies that PROC CONTENTS output include details about compressed blocks, data partition size, and indexes.

Dictionary

SPDSAUNQ Macro Variable

Specifies whether to cancel an append or insert to a table. Use this macro variable if the table has a unique index and the new rows would violate the index uniqueness.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Tip: Use the UNIQUESAVE table option (or SPDSUSAV macro variable) to save rejected rows when SPDSAUNC=NO.

Syntax

SPDSAUNQ=YES | NO

Required Arguments

YES
cancels the Append or Insert operation if any duplicate values are found.

NO
performs the Append or Insert operation enforcing uniqueness at the expense of appending unique indexes in row order, one row at a time. Only rows with duplicate values are discarded.

Details

Set SPDSAUNQ=YES to improve append or insert performance to a table with unique indexes. If uniqueness is not maintained, the Append or Insert is canceled and the table is returned to its state before the additions. At this point, you have two choices: remove the duplicates from the table being added and attempt the operation again, or repeat the operation with SPDSAUNQ set to NO. When SPDSAUNQ=NO, set the SPDSUSAV macro variable (or UNIQUESAVE= table option) to YES to save rejected rows.

See Also

SPD Server macro variables:

- “SPDSUSAV Macro Variable” on page 237
- “SPDSUSDS Reserved Macro Variable” on page 208

SPD Server table options:

- “UNIQUESAVE= Table Option” on page 273

SPDSBNEQ Macro Variable

Specifies the output order of table rows that have identical values in the BY column.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Interaction: Corresponding table option is BYNOEQUALS=.

Syntax

SPDSBNEQ=YES | NO

Required Arguments

YES
generates rows with identical values in a BY clause in random order.

NO
generates rows with identical values in a BY clause using the relative table position of the rows from the input table.

Details

SPDSBNEQ=NO configures SPD Server to imitate the Base SAS engine behavior. If strict compatibility is not required, assign SPDSBNEQ=YES. Random output enables the server to create indexes and append to tables faster.

Example

Configure the server so that the table rows are generated as quickly as possible when processing rows that have identical values in the BY column.

```
%let SPDSBNEQ=YES;
```

See Also

SPD Server table options:

- "BYNOEQUALS= Table Option" on page 246

SPDSBSRT Macro Variable

Configures SPD Server's sorting behavior when it encounters a BY clause and there is no index available.

Valid in: SPD Server

Default: YES

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Interaction: Corresponding table option is BYSORT=.

Syntax

SPDSBSRT=YES | NO

Required 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.

Details

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.

Example

At the start of a session to run old converted 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 the server to turn off implicit sorts. Put the macro variable assignment in your autoexec.sas file so that the server retains the configuration for all job sessions.

```
%let SPDSBSRT=NO;
```

During this server session, you decide to run a new program that has no PROC SORT statements. Instead, the new program takes advantage of the server implicit sorts.

```
libname inventory sasspds "conversion_area" server=samson.5105
user="siteusr1" password="secret";

data inventory.old_autos;
   infile datalines delimiter=',';
   input
     year $
     manufacturer $
     model $
     body_style $
     engine_liters
     transmission_type $
     exterior_color $
     options
     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
;
run;

title Old Autos Table with SPDSBSRT=NO;
proc print data=inventory.old_autos;
by model;
run;
```

When the code executes, the PRINT procedure returns the following error message because SAS expected Inventory.OldAutos to be sorted before it would generate print output.

```
ERROR: Data set TEMPDATA.OLD_AUTOS is not sorted in ascending sequence.
The current BY group has model = Skylark and the next BY group has model =
Fiesta.
```

Since there is no PROC SORT statement—and implicit sorts are still turned off, the sort does not occur.

Keep implicit sorts turned off for the session, but specify an implicit sort for the table Inventory.Old_Autos by using the BYSORT= table option:

```
proc print data=inventory.old_autos(bysort=yes);
 by model;
 run;
```

See Also

SPD Server table options:

- "BYSORT= Table Option" on page 247

SPDSCLJX Macro Variable

Affects the SAS SQL planner when a SAS table is being joined with an indexed SPD Server table.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Syntax

SPDSCLJX=YES | NO

Required Arguments

YES

The SAS SQL planner includes the index join as a possible method for joining a Base SAS table with an SPD Server cluster table.

NO
The SAS SQL planner disallows the index join as a possible method for joining a Base SAS table with an SPD Server cluster table.

Details

SPDSCLJX=YES can potentially improve processor performance during heterogeneous joins between an SPD Server cluster table and Base SAS table. Index joins tend to be most beneficial when there are relatively few rows in the SAS table.

Note: The SPDSCLJX macro variable has no effect on implicit or explicit join queries to SPD Server that involve a cluster table.

SPDSCMPF Macro Variable

Specifies the amount of growth space, sized in bytes, to be added to a compressed data block.

Valid in: SPD Server

Default: 0 bytes

Syntax

SPDSCMPF=*n*

Required Argument

n
the number of bytes to add.

Details

Updating rows in compressed tables can increase the size of a given table block. Additional space is required in order 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.

SPDSCOMP Macro Variable

Specifies whether to compress the data when sending a data packet through the network.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Syntax

SPDSCOMP=YES | NO

Required Arguments

YES
compresses the data when sending a data packet through the network.

NO
does not compress the data when sending a data packet through the network.

SPDSDCMP Macro Variable

Specifies to compress SPD Server tables that are stored on disk.

Valid in: SPD Server

Default: NO

Restrictions: A server table cannot be encrypted if it is compressed.

Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Interactions: The corresponding table option is COMPRESS=. If you specify values for both the COMPRESS= table option and the SPDSDCMP macro variable, the SPDSDCMP setting overrides the COMPRESS=setting.

Used in conjunction with the IOBLOCKSIZE= table option.

Syntax

SPDSDCMP=YES | CHAR | BINARY

Required Arguments

YES | CHAR
specifies that the rows in a newly created table are compressed by SAS using run-length encoding (RLE). RLE compresses rows by reducing repeated consecutive characters (including blanks) to 2-byte or 3-byte representations. Use the YES or CHAR argument to enable RLE compression for character data. The YES and CHAR arguments are functionally identical and interchangeable.

BINARY
specifies that the rows in a newly created table are compressed by SAS using Ross Data Compression (RDC). RDC combines run-length encoding and sliding-window compression to compress the file. Use the BINARY argument to compress binary and numeric data. This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data (numeric variables).

Details

When you set the SPDSDCMP macro variable to **`YES`**, the 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 more information, see “IOBLOCKSIZE= Table Option” on page 259.

Note: Once a compressed table is created, you cannot change its block size. To resize the block, you must issue PROC COPY to copy the table to a new table, setting IOBLOCKSIZE= to the new block size for the output table.

Example

You should conserve disk space before you create a huge table. You can use SPDSDCMP to compress character and numeric data at the beginning of your job.

```
%let SPDSDCMP=BINARY;
```

See Also

SPD Server table options:

- "COMPRESS= Table Option" on page 250
- "IOBLOCKSIZE= Table Option" on page 259

SPDSEINT Macro Variable

Specifies how SPD Server responds to network disconnects during SQL explicit pass-through EXECUTE() statements.

Valid in: SPD Server

Default: YES

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Syntax

SPDSEINT=YES | NO

Required Arguments

YES
interrupts SQL processing by default when a network failure occurs.

NO
configures the server's SQL processor to continue processing until completion regardless of network disconnects.

Details

The server's SQL processor interrupts SQL processing by default when a network failure occurs. The interruption prematurely terminates the EXECUTE() statement. Setting SPDSEINT=NO configures the server's SQL processor to continue processing until completion regardless of network disconnects.

CAUTION:
Use the macro variable setting SPDSEINT=NO carefully! A runaway EXECUTE() statement requires a privileged system user on the server machine to kill the server SQL proxy process. This is the only way to stop the processing.

SPDSEOBS Macro Variable

Specifies the number of the last row of a user-defined range that you want to process in a table.

Valid in: SPD Server

Default: The default setting of 0 processes the entire table.

Interaction: Corresponding table option is ENDOBS=.

Syntax

SPDSEOBS=*n*

Required Argument

n
the number of the end row.

Details

The 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, the 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 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;
```

See Also

SPD Server table options:

- "ENDOBS= Table Option" on page 257

SPDSEV1T Macro Variable

Specifies whether data returned from an SPD Server WHERE clause evaluations should be in strict row order.

Valid in: SPD Server

Default: 1

Interactions: Use in conjunction with the indexed WHERE clause evaluation strategy.

This macro variable works in conjunction with the SPD Server WHERE clause planner WHINIT.

Note: The SPDSEV1T evaluation strategy is indexed.

Syntax

SPDSEV1T=0 | 1 | 2

Required Arguments

0

returns data in row order.

TIP 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.

1

might not return the data in row order. The 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. Might not return data in row order.

TIP Use **2** only when you know row order is not important to the result.

Details

The macro variables SPDSEV1T and SPDSEV2T work in conjunction with the 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 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, the server returns data in row order whetherthe SPDSEV1T or the SPDSEV2T strategy is selected.

You use a SAS procedure to retrieve rows from a sorted table. Some SAS procedures can use the sort order information to optimize how to receive and process the rows. For example, you use PROC SQL to perform table joins on a sorted table that uses WHERE predicates to filter table rows. PROC SQL uses the sort order information to optimize the join strategy. If you use the default values of SPDSEV1T and SPDSEV2T in these instances, the SAS procedures receives the table rows in sorted order.

If the SAS procedure that you submit does *not* use 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 procedure that you are submitting does not take advantage of a sorted table, you can change the setting for SPDSEV1T or SPDSEV2T to 2. This change allows 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 procedure that uses that information.

```
%let SPDSEV1T=0;
```

Note: The SPDSEV1T and SPDSEV2T usage that is discussed here does not apply to SQL statements that are executed via the server SQL explicit pass-through facility.

See Also

Concepts:

- Chapter 16, "WHERE Clause Planner," on page 127

SPDSEV2T Macro Variable

Specifies whether the data returned from WHERE clause evaluations should be in strict row order.

Valid in: SPD Server

Default: 1

Interactions: Use in conjunction with the indexed WHERE clause evaluation strategy.

This macro variable works in conjunction with the SPD Server WHERE clause planner WHINIT.

Note: The SPDSEV2T evaluation strategy is non-indexed.

Syntax

SPDSEV2T=0 | 1 | 2

Required Arguments

0

returns data in row order.

TIP 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.

1

might not return the data in row order. The 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. Might not return data in row order.

TIP Use **2** only when you know that row order is not important to the result.

Details

Use the SPDSEV2T macro variable to specify whether the data returned from WHERE clause evaluations should be in strict row order.

The macro variables SPDSEV1T and SPDSEV2T work in conjunction with the 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 completely understand the 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, the server returns data in row order whether the SPDSEV1T or the SPDSEV2T strategy is selected.

You use a SAS procedure to retrieve rows from a sorted table. Some SAS procedures can use the sort order information to optimize how to receive and process the rows. For example, you use PROC SQL to perform table joins on a sorted table that uses WHERE predicates to filter table rows. PROC SQL uses the sort order information to optimize the join strategy. If you use the default values of SPDSEV1T and SPDSEV2T in these instances, the SAS procedure receives the table rows in sorted order.

If the SAS procedure that you submit does *not* use 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 procedure that you are submitting does not take advantage of a sorted table, you can change the setting for SPDSEV1T or SPDSEV2T to 2. This change allows 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 procedure 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 server SQL explicit pass-through facility.

See Also

Concepts:

- Chapter 16, “WHERE Clause Planner,” on page 127

SPDSFSAV Macro Variable

Specifies whether you want to retain table data if the SPD Server table creation process terminates abnormally.

Valid in: SPD Server

Default: NO. Normally, SAS closes and deletes tables that are not properly created.

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Syntax

SPDSFSAV=YES | NO

Required Arguments

YES
enables FORCESAVE mode and saves the table.

NO
Default server actions delete partially completed tables.

Details

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 an error condition might be signaled. If SAS encounters such an error condition, it deletes the partially completed table.

In the 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 the 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;
```

SPDSHOST Macro Variable

Assigns an SPD Server host name.

Valid in: SPD Server

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Syntax

SPDSHOST=*HOST-NAME*

Required Argument

HOST-NAME
the node name of the SPD Server machine.

Details

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

Example

The first statement assigns the host Samson to the macro variable SPDSHOST.

```
%let spdshost=SAMSON;

libname myref sasspds 'mylib'
user='siteusr1'
password='mypasswd';
```

SPDSIASY Macro Variable

Specifies whether to create indexes in parallel when creating multiple indexes on an SPD Server table.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Interaction: Corresponding table option is ASYNCINDEX=.

Syntax

SPDSIASY=YES | NO

Required Arguments

YES
creates the indexes in parallel.

NO
creates one index at a time.

Details

You use the macro variable SPDSIASY to choose between parallel and sequential index creation on the 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 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 otherwise 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 work storage space needed for index key sorting.

When managing disk space on your server, remember that grouping INDEX CREATE statements can minimize the number of table scans that the 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

You 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 the 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, the 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.

See Also

SPD Server table options:

- "ASYNCINDEX= Table Option" on page 245

SPDSIPDB Macro Variable

Specifies whether to include SQL implicit pass-through code or error messages in the SPD Server log.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Syntax

SPDSIPDB=YES | NO

Required Arguments

YES
SQL implicit pass-through code or error messages are included in the SPD Server log.

NO
SQL implicit pass-through code or error messages are not included in the SPD Server log.

SPDSIRAT Macro Variable

Specifies whether to perform segment candidate pre-evaluation when performing WHERE clause processing with indexes.

Valid in: SPD Server

Interaction: Corresponding server parameter option is MAXSEGRATIO=.

Syntax

SPDSIRAT=*0...100*

Details

When using 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 are 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 contain candidates by the number of total segments is compared to a cutoff point. 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. The 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 server parameter MAXSEGRATIO to set the default cutoff value. The default MAXSEGRATIO should provide good performance. Certain specific query situations might justify modifying your SPDSIRAT value. When you modify your SPDSIRAT value, the new value 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;
```

See Also

Concepts:

- Chapter 16, "WHERE Clause Planner," on page 127

SPDSNBIX Macro Variable

Specifies to suppress index use during a BY sort.

Valid in: SPD Server

Default: NO

Restrictions: The SPDSNBIX macro variable cannot enable index use for BY sorts. It can suppress index use when index use for BY sorts is enabled on the server. The corresponding server parameter is BYINDEX.

Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Syntax

SPDSNBIX=YES | NO

Required Arguments

YES
suppresses 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
uses indexes for BY sorts, if index use for BY sorts is enabled on the server.

Example

```
%let SPDSNBIX=YES;
```

SPDSNETP Macro Variable

Sizes buffers in server memory for the network data packet.

Valid in: SPD Server

Default: 32K

Interaction: Corresponding table option is NETPACKSIZE=.

Syntax

SPDSNETP=*size-of-packet*

Required Argument

size-of-packet
the size (integer) in bytes of the network packet.

Details

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. For more information, see "NETPACKSIZE= Table Option" on page 262.

Example

Despite recent upgrades to your network connections, you are experiencing significant pauses when the 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;
```

See Also

SPD Server table options:

- "NETPACKSIZE= Table Option" on page 262

SPDSNIDX Macro Variable

Specifies whether to use the table's indexes when processing WHERE clauses. SPDSNIDX can also be used to disable index use for BY order determination.

Valid in: SPD Server

Default: NO

Restrictions: SPDSNIDX affects index usage for BY ordering only if index usage for BY sorts is enabled on the server. The corresponding server parameter is BYINDEX.

Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Interaction: Corresponding table option is NOINDEX=.

Syntax

SPDSNIDX=YES | NO

Required Arguments

YES
ignores indexes when processing WHERE clauses.

NO
uses indexes when processing WHERE clauses.

Details

Set SPDSNIDX=YES to test the effect of indexes on performance or for specific processing. Do not use YES routinely for normal processing.

Example

You havecreated an index for the Type column of table TempData.AudiCars but decide to test whether it is necessary for your processing. You set SPDSNIDX=YES and a PROC PRINT statement, then set SPDSNIDX=NO and a PROC PRINT statement, so that you can compare processing. You also set the SPDSWDEB macro variable.

Set SPDSNIDX to YES to ignore index:

```
libname tempdata sasspds "conversion_area" server=husky.5105
    user="siteusr1" password="secret";

proc sql;
drop table tempdata.audicars;

create table tempdata.audicars as
select * from sashelp.cars
where make="Audi";

create index type on tempdata.audicars(type);

quit;

/*Turn on the macro variable SPDSWDEB */
/* to show that the index is not used */
/* during the table processing. */
%let spdswdeb=YES;

/* Set SPDSNIDX to YES to ignore index */
%let spdsnidx=YES;

title "Sedans manufactured by Audi";
proc print data=tempdata.audicars;
 where type="Sedan";
run;

/* Set SPDSNIDX to NO to ignore index */
%let spdsnidx=NO;

title "Sedans manufactured by Audi";
proc print data=tempdata.audicars;
 where type="Sedan";
run;
```

The following information was written to the log for the PROC PRINT request that uses SPDSNIDX=YES:

```
207  /*Turn on the macro variable SPDSWDEB */
208  /* to show that the index is not used */
209  /* during the table processing. */
210  %let spdswdeb=YES;
211
212  /* Set SPDSNIDX to YES to ignore index */
213  %let spdsnidx=YES;
214
215  title "Sedans manufactured by Audi";
216  proc print data=tempdata.audicars;
217   where type="Sedan";


whinit: WHERE (type='Sedan')
whinit: wh-tree presented
           /-NAME = [type]
 --CEQ----|
           \-LITC = ['Sedan']
whinit returns: ALL EVAL2

218  run;



whinit: WHERE (type='Sedan')
whinit: wh-tree presented
           /-NAME = [type]
 --CEQ----|
           \-LITC = ['Sedan']
whinit returns: ALL EVAL2



whinit: WHERE (type='Sedan')
whinit: wh-tree presented
           /-NAME = [type]
 --CEQ----|
           \-LITC = ['Sedan']
whinit returns: ALL EVAL2
NOTE: There were 13 observations read from the data set TEMPDATA.AUDICARS.
      WHERE type='Sedan';
```

The following information was written to the log for the PROC PRINT request that uses SPDSNIDX=NO:

```
220  %let spdsnidx=NO;
221
222  title "Sedans manufactured by Audi";
223  proc print data=tempdata.audicars;
224   where type="Sedan";

whinit: WHERE (type='Sedan')
whinit: wh-tree presented
           /-NAME = [type]
 --CEQ----|
           \-LITC = ['Sedan']
whinit: wh-tree after split
 --<empty>
whinit: INDEX Type uses 100% of segs (WITHIN maxsegratio 100%)
whinit: INDEX tree after split
            /-NAME = [type] <1>INDEX Type (type)
  --CEQ----|
            \-LITC = ['Sedan']
whinit costing: 1 segs with est 1% yield reduces whthreads from 32 to 1
whinit returns: ALL EVAL1(w/SEGLIST)
225  run;

whinit: WHERE (type='Sedan')
whinit: wh-tree presented
           /-NAME = [type]
  --CEQ----|
            \-LITC = ['Sedan']
whinit: wh-tree after split
 --<empty>
whinit: INDEX Type uses 100% of segs (WITHIN maxsegratio 100%)
whinit: INDEX tree after split
            /-NAME = [type] <1>INDEX Type (type)
  --CEQ----|
            \-LITC = ['Sedan']
whinit costing: 1 segs with est 1% yield reduces whthreads from 32 to 1
whinit returns: ALL EVAL1(w/SEGLIST)

whinit: WHERE (type='Sedan')
whinit: wh-tree presented
            /-NAME = [type]
  --CEQ----|
            \-LITC = ['Sedan']
whinit: wh-tree after split
--<empty>
whinit: INDEX Type uses 100% of segs (WITHIN maxsegratio 100%)
whinit: INDEX tree after split
           /-NAME = [type] <1>INDEX Type (type)
 --CEQ----|
           \-LITC = ['Sedan']
whinit costing: 1 segs with est 1% yield reduces whthreads from 32 to 1
whinit returns: ALL EVAL1(w/SEGLIST)
NOTE: There were 13 observations read from the data set TEMPDATA.AUDICARS.
      WHERE type='Sedan';
```

The **`whinit returns`** message shows ALL EVAL1 when the index is used, and ALL EVAL2 when a sequential pass is used.

See Also

SPD Server table options:

- "NOINDEX= Table Option" on page 263

Concepts:

- Chapter 16, "WHERE Clause Planner," on page 127

SPDSRSSL Macro Variable

Specifies to make a secure sockets connection to SPD Server.

Valid in: AUTOEXEC file, SAS session

Default: NO

Requirements: SPD Server must be configured with a compatible SSLSECURE= value or the client connection will fail.

The SPDSRSSL macro variable must be specified before the SASSPDS LIBNAME statement in order for any changes to the default value to be applied.

Syntax

SPDSRSSL=YES | NO

Required Arguments

YES
specifies to make a secure sockets connection to SPD Server.

NO
specifies that the SPD Server connection is not secured.

Details

Beginning with SPD Server 5.3, SPD Server supports secure sockets connections by using Transport Layer Security (TLS). TLS and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols that are designed to provide over-the-wire communication security. TLS and SSL provide network data privacy, data integrity, and authentication.

SPD Server is configured to use TLS by default (SSLSECURE=YES). When SSLSECURE=YES is set on the server, SPDSRSSL must also be set to YES or SAS client connections will fail. When SSLSECURE=YES, the server also will not accept client connections from ODBC and JDBC clients. SPD Server does not support secure sockets connection for ODBC and JDBC clients in the initial release. However, an SPD Server 5.3 SAS client that specifies SPDSRSSL=YES is not prevented from connecting to an SPD Server 5.2 (or earlier) server.

Administrators can modify the default server setting to make TLS optional (SSLSECURE=PREFERRED) or to not use TLS at all (SSLSECURE=NO). When SSLSECURE=PREFERRED, the server will accept connections from clients that specify SPDSRSSL=YES, from clients that specify SPDSRSSL=NO, and from clients for which TLS is not supported. Client/server communication with clients that specify SPDSRSSL=YES is secured. Other connections are not secure. SSLSECURE=PREFERRED is recommended for servers that will support ODBC and JDBC clients in addition to SAS clients.

When SSLSECURE=NO, SPDSRSSL must also be set to NO, or SAS client connections will fail.

SPDSSADD Macro Variable

Specifies whether SPD Server appends tables by transferring a single row at a time synchronously, or by transferring multiple rows asynchronously (block row appends).

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Interaction: Corresponding table option is SYNCADD=.

Tip: Use the UNIQUESAVE table option (or SPDSUSAV macro variable) to save rejected rows when SPDSSADD is set to NO.

Syntax

SPDSSADD=YES | NO

Required Arguments

YES
applies a single row at a time during an Append operation.

NO
appends multiple rows at a time.

Details

SPDSSADD=YES slows performance.

See Also

SPD Server macro variables:

- "SPDSUSAV Macro Variable" on page 237

SPD Server table options:

- "SYNCADD= Table Option" on page 269

SPDSSIZE Macro Variable

Specifies the size of an SPD Server table partition.

Valid in: SPD Server

Default: 16 MB for domains that are not Hadoop domains, 128 MB for Hadoop domains

Restriction: The SPDSSIZE specification is limited by MINPARTSIZE=, a server parameter that is maintained by the server administrator. Ask your administrator what the

MINPARTSIZE= setting is for your site. If you use SPDSSIZE, its value must be greater than MINPARTSIZE= to have any effect.

Interaction: Corresponding table option is PARTSIZE=.

Syntax

SPDSSIZE=*n*

Required Argument

n
the size of the partition. The number is assumed to be in megabytes.

Details

The SPDSSIZE macro variable determines the partition size of any tables that are created during a server session until SPDSSIZE is specified again with a different value or the value is overridden by the PARTSIZE= table option. To take effect, the SPDSSIZE value must be greater than the value declared for MINPARTSIZE= in the server parameter file.

Use the SPDSSIZE macro variable 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 the 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 that are required to store the rows of the table. The scalability achieved with SPDSSIZE ultimately depends on how the SPD Server administrator structured the DATAPATH= option for the domain. The configuration of the DATAPATH= file systems across striped volumes is important. Each individual volume's striping configuration should be spread across multiple disk controllers and SCSI channels in the disk storage array. The configuration goal, at the hardware level, is to maximize parallelism when performing data retrieval.

The default SPDSSIZE values represent the absolute minimum recommended MINPARTSIZE= settings for each environment. They are intended to ensure that an over-zealous user cannot arbitrarily create small partitions, thereby generating an excessive number of physical files. Many sites specify a higher MINPARTSIZE= value in their server parameter file.

Note: The partition size for a table cannot be changed after a table is created. If you must change the partition size, use PROC COPY to duplicate the table and specify a different SPDSSIZE setting (or PARTSIZE= value) on the output table.

Example

To set a partition size of 50 MB with SPDSSIZE, specify the following. If 50 MB is greater than the MINPARTSIZE= setting and you are setting the option for a table that is not in a Hadoop domain, the value will be applied. Otherwise, SPDSSIZE will have no effect.

```
%let SPDSSIZE=50;
```

See Also

SPD Server table options:

- "PARTSIZE= Table Option" on page 265

SPDSSOBS Macro Variable

Specifies the number of the starting row in a user-defined range of a table.

Valid in: SPD Server

Default: The default setting of 0 processes the entire table.

Interaction: Corresponding table option is STARTOBS=.

Syntax

SPDSSOBS=*n*

Required Argument

n
the number of the start row.

Details

By default, the 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 specify the beginning of the range using SPDSSOBS, the server sets SPDSSOBS to 1. 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;
```

See Also

SPD Server table options:

- "STARTOBS= Table Option" on page 267

SPDSSQLR Macro Variable

Specifies an SQL reset option using implicit pass-through.

Valid in: SPD Server

Syntax

SPDSSQLR=*reset-option*

Required Argument

reset-option
an SQL reset option.

Details

For more information and examples, see "Specify SQL Options for SQL Implicit Pass-Through Code" on page 78.

SPDSSTAG Macro Variable

Specifies whether to use non-tagged or tagged sorting for PROC SORT or BY processing.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Syntax

SPDSSTAG=YES | NO

Required Arguments

YES
performs tagged sorting.

NO
performs non-tagged sorting.

Details

During a non-tagged sort, the server attaches the entire table column to the key field or fields 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 low on disk space and do not know whether you have enough disk overhead to accommodate the extra sort space required to support a non-tagged sort operation.

Configure the server to perform a tagged sort.

```
%let SPDSSTAG=YES;
```

SPDSTCNT Macro Variable

Specifies the number of threads that you want to use during WHERE clause evaluations.

Valid in: SPD Server

Default: The value of MAXWHTHREADS configured by SPD Server parameters.

Interactions: Corresponding table option is THREADNUM=.
Use in conjunction with server parameter option MAXWHTHREADS.

Syntax

SPDSTCNT=*n*

Required Argument

n
the number of threads.

Details

See "THREADNUM= Table Option" on page 272 for a description and an explanation of how SPDSTCNT interacts with the SPD Server parameter MAXWHTHREADS.

See Also

SPD Server table options:

- "THREADNUM= Table Option" on page 272

Concepts:

- Chapter 16, "WHERE Clause Planner," on page 127

SPDSUSAV Macro Variable

Specifies whether to save rows with non-unique (rejected) keys to a separate SAS table.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Interactions: Corresponding table option is UNIQUESAVE=.

Use in conjunction with the SPDSUSDS reserved macro variable.

Note: This macro variable has no effect when the SYNCADD= table option is set to YES.

Syntax

SPDSUSAV=YES | NO | REP

Required Arguments

YES
writes rejected rows to a separate, system-created table. This table can be accessed by a reference to the macro variable SPDSUSDS.

NO
ignores duplicate rows rejected by an Append or Insert operation.

REP
replaces the current row in the master table with the duplicate row from the Insert or Append operation, instead of saving the rows to a separate table. This setting is useful when updating a master table from a transaction table, where the two tables share identical column structures.

Details

When the SPDSAUNQ macro variable is set to NO (the default value), rows with duplicate index values are rejected unless you specify UNIQUESAVE=YES (or set the SPDSUSAV macro variable to YES). By using UNIQUESAVE=YES, you can save rejected values to a hidden system table. When UNIQUESAVE=YES, a NOTE on the log identifies the name of the table. To access that table you can either cut-and-paste from the log, or refer to that table by using the reserved macro variable SPDSUSDS.

Example

Append several tables to the EMPLOYEE table, using employee number as a unique key. The appended tables should not have rows with duplicate employee numbers.

At the beginning of the job, configure SPD Server to write any rejected (identical) employee number rows 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;
```

See Also

SPD Server macro variables:

- "SPDSUSDS Reserved Macro Variable" on page 208

SPD Server table options:

- "SYNCADD= Table Option" on page 269
- "UNIQUESAVE= Table Option" on page 273

SPDSVERB Macro Variable

Specifies that PROC CONTENTS output include details about compressed blocks, data partition size, and indexes.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Interaction: Corresponding table option is VERBOSE=.

Syntax

SPDSVERB=YES | NO

Required Arguments

YES
requests detail information about compressed blocks, data partition size, and indexes.

NO
suppresses detail information about compressed blocks, data partition size, and indexes.

Example

You need information about associated indexes for the server table MyLib.Supply. Configure the server for verbose details at the start of your session so that 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 MyLib.Supply table:

```
proc contents data=mylib.supply;
run;
```

See Also

SPD Server table options:

- "VERBOSE= Table Option" on page 277

SPDSWCST Macro Variable

Specifies whether to use dynamic WHERE clause costing.

Valid in: SPD Server

Default: YES

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Interaction: Corresponding server parameter option is WHERECOSTING.

Syntax

SPDSWCST=YES | NO

Required Arguments

YES
use dynamic WHERE clause costing.

NO
do not use dynamic WHERE clause costing.

Details

The SPDSWCST macro variable enables you to turn dynamic WHERE clause costing off if the server is configured with the WHERECOSTING server parameter. If the server is configured with the NOWHERECOSTING server parameter, any declaration of values using SPDSWCST is ignored.

WHERE clause costing causes WHINIT to use a heuristic method to determine (among other things) which indexes to use to evaluate the WHERE clause. When WHERE costing is disabled, WHINIT uses a rules-based evaluation that includes all available indexes to evaluate the WHERE clause.

SPDSWCST=NO affects all server connections.

Example

```
%let SPDSWCST=NO;
```

See Also

Concepts:

- Chapter 16, "WHERE Clause Planner," on page 127

SPDSWDEB Macro Variable

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.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Syntax

SPDSWDEB=YES | NO

Required Arguments

YES
displays WHINIT's planning output.

NO
suppresses WHINIT's planning output.

See Also

Concepts:

- Chapter 16, "WHERE Clause Planner," on page 127

SPDSWSEQ Macro Variable

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.

Valid in: SPD Server

Default: NO

Restriction: Assignments for macro variables with character input (for example, YES | NO | BINARY arguments) must be entered in uppercase (capitalized).

Syntax

SPDSWSEQ=YES | NO

Required Arguments

YES
overrides WHERE clause costing and forces a global EVAL3 or EVAL4 strategy.

NO
does not override WHERE clause costing.

Example

```
%let SPDSWSEQ=YES;
```

See Also

Concepts:

- Chapter 16, “WHERE Clause Planner,” on page 127

Chapter 22

SPD Server Table Options

Overview of SPD Server Table Options

This chapter describes table options supported by SPD Server. The table options are intended for use in the DATA step and SAS procedures. Most table options also work in SQL pass-through statements.

To specify a table option in the DATA step or SAS procedure, place the table option in parentheses after the table name. The option value then specifies processing that applies only to that table.

To specify a table option in an SQL pass-through request, place the table option in brackets after the table name. The option value then specifies processing that applies only to that table.

When a table option is used subsequent to a LIBNAME statement option or macro variable, the value of the table option takes precedence.

Functional List of SPD Server Table Options

The following table lists the table options by their purpose.

Purpose	Table Option	Description
Affect disk space	"COMPRESS= Table Option" on page 250	Compresses SPD Server tables on disk.
	"PARTSIZE= Table Option" on page 265	Specifies the size of an SPD Server table partition.
Affect sorts	"BYNOEQUALS= Table Option" on page 246	Specifies the output order of table rows with identical values for the BY column.
	"BYSORT= Table Option" on page 247	Specifies to perform an implicit automatic sort when SPD Server encounters a BY clause for a given table.
Affect WHERE clause evaluations	"MINMAXVARLIST= Table Option" on page 260	Creates a list that documents the minimum and maximum values of specified variables.
	"THREADNUM= Table Option" on page 272	Specifies the number of threads to be used for WHERE clause evaluations.
	"WHERENOINDEX= Table Option" on page 278	Specifies a list of indexes to exclude when making WHERE clause evaluations.
Enhance performance	"ASYNCINDEX= Table Option" on page 245	Specifies when creating multiple indexes on an SPD Server table whether to create the indexes in parallel.
	"IOBLOCKSIZE= Table Option" on page 259	Specifies the number of rows in a block to be stored in or read from an SPD Server table.
	"NETPACKSIZE= Table Option" on page 262	Specifies the size of the SPD Server network data packet.
Handle duplicates	"SEGSIZE= Table Option" on page 267	Specifies the size of the segment for an index file associated with an SPD Server table.
	"SYNCADD= Table Option" on page 269	Specifies to process one row at a time or multiple rows at a time when adding rows.

Purpose	Table Option	Description
	"UNIQUESAVE= Table Option" on page 273	Specifies to save rows with non-unique key values (the rejected rows) to a separate table when appending data to tables with unique indexes.
Security	"ENCRYPT= Table Option" on page 252	Encrypts SPD Server tables on disk.
	"ENCRYPTKEY= Table Option" on page 255	When you use the ENCRYPT=AES option setting to specify AES-256 encryption, you must use the ENCRYPTKEY= option. This option sets a text string value that will enable the RSA 256-bit encryption key to encode data and indexes at rest on the server disk.
Test performance	"NOINDEX= Table Option" on page 263	Specifies whether to use the table's indexes when processing WHERE clauses.
Miscellaneous function	"ENDOBS= Table Option" on page 257	Specifies the end row number in a user-defined range for the processing of a given table.
	"STARTOBS= Table Option" on page 267	Specifies the start row number in a user-defined range for the processing of a given table.
	"VERBOSE= Table Option" on page 277	Specifies whether the CONTENTS procedure output includes details about compressed blocks, data partition size, and indexes.

Dictionary

ASYNCINDEX= Table Option

When you are creating multiple indexes on an SPD Server table, specifies whether to create the indexes in parallel.

Valid in: SPD Server

Default: No

Interaction: Corresponding macro variable is SPDSIASY.

Syntax

ASYNCINDEX=YES | NO

Required Arguments

YES
creates the indexes in parallel.

NO
creates a single index at a time.

Details

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 trade-offs of creating multiple indexes in parallel, see "SPDSIASY Macro Variable" on page 224.

Example

When the disk work space required for parallel index creation is available, specify ASYNCINDEX=YES for the 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;
```

See Also

SPD Server macro variables:

- "SPDSIASY Macro Variable" on page 224

BYNOEQUALS= Table Option

Specifies the output order of table rows with identical values for the BY column.

Valid in: SPD Server

Default: No

Interaction: Corresponding macro variable is SPDSBNEQ.

Syntax

BYNOEQUALS=YES | NO

Required Arguments

YES
does not guarantee the output order of table rows with identical values in a BY clause.

NO
guarantees that the output order of table rows with identical values in a BY clause is the relative table position of the rows from the input table. This value is the default.

Example

Specify for the server in the ensuing BY-column operation to randomly output rows with identical values in the key column.

```
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;
```

See Also

SPD Server macro variables:

- "SPDSBNEQ Macro Variable" on page 213

BYSORT= Table Option

Specifies to perform an implicit automatic sort when SPD Server encounters a BY clause for a given table.

Valid in: SPD Server

Default: YES

Interaction: Corresponding macro variable is SPDSBSRT.

Syntax

BYSORT=YES | NO

Required 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.

Details

The default is YES. The NO argument means that the table must have been previously sorted by the requested BY columns. The NO argument allows data to retain its precise order in the table. A YES argument groups the data correctly but possibly in a different order from the order in the table.

Examples

Example 1: Group Formatting with the BYSORT= Table Option

The following example uses group formatting with BYSORT= table option.

```
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 22.1 *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 the BYSORT= Table Option

The following example uses group formatting without the BYSORT= table option.

```
PROC PRINT data=sport.racquets (bysort=no);
   var raqname instock;
   by price;
   format price dollars.;
title 'Solo Brand Racquets by Price Level';
run;
```

Output 22.2 *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
```

See Also

Macro variables:

- "SPDSBSRT Macro Variable" on page 213

COMPRESS= Table Option

Compresses SPD Server tables on disk.

Valid in: SPD Server

Default: No

Restriction: A server table cannot be encrypted if it is compressed.

Interactions: Corresponding macro variable is SPDSDCMP. If you specify values for both the COMPRESS= table option and the SPDSDCMP macro variable, the SPDSDCMP setting overrides the COMPRESS= setting.

Use COMPRESS= in conjunction with the IOBLOCKSIZE= table option.

Syntax

COMPRESS=NO | YES | CHAR | BINARY

Required Arguments

CHAR|YES
specifies that data in a newly created table be compressed by SAS using run-length encoding (RLE). RLE compresses data by reducing repeated consecutive characters (including blanks) to 2-byte or 3-byte representations. Use the YES or CHAR argument to enable RLE compression for character data. The two arguments are functionally identical and interchangeable.

BINARY
specifies that the data in a newly created table be compressed by SAS using Ross Data Compression (RDC). RDC combines run-length encoding and sliding-window

compression to compress the file. Use the BINARY argument to compress binary and numeric data. This method is highly effective for compressing medium to large (several hundred bytes or larger) blocks of binary data.

NO
specifies that the data in the table is not to be compressed.

Details

When COMPRESS=YES is specified, the 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.

When COMPRESS=BINARY is specified, both numeric data and character data are compressed.

Note: Once a compressed table is created, you cannot change its block size. To resize the block, you must use PROC COPY to create a new table from this table, setting IOBLOCKSIZE= to the block size desired for the output table.

Examples

Example 1: Using the COMPRESS=YES Table Option

```
data mylib.CharRepeats(COMPRESS=YES);
   length ca $ 200;
   do i=1 to 100000;
      ca='aaaaaaaaaaaaaaaaaaaaa';
      cb='bbbbbbbbbbbbbbbbbbbbb';
      cc='ccccccccccccccccccccc';
      output;
   end;
run;
```

The following message is written to the log:

```
NOTE: Compressing table MYLIB.CHARREPEATS decreased size by 93.34 percent.
```

Example 2: Using the COMPRESS=BINARY Table Option

```
data mylib.StringRepeats(COMPRESS=BINARY);
   length cabcd $ 200;
   do i=1 to 1000000;
      cabcd='abcdabcdabcdabcdabcdabcdabcd';
      cefgh='efghefghefghefghefghefghefgh';
      cijkl='ijklijklijklijklijklijklijkl';
      output;
   end;
run;
```

The following message is written to the SAS log:

```
NOTE: Compressing table MYLIB.STRINGREPEATS decreased size by 97.85 percent.
```

See Also

SPD Server macro variables:

- "SPDSDCMP Macro Variable" on page 217

SPD Server table options:

- "IOBLOCKSIZE= Table Option" on page 259

ENCRYPT= Table Option

Encrypts SPD Server tables on disk.

Valid in: SPD Server

Default: NO

Restriction: A server table cannot be compressed if it is encrypted.

Interaction: Use ENCRYPT= in conjunction with the IOBLOCKSIZE= table option.

Syntax

ENCRYPT=YES | NO | AES

Required Arguments

YES
encrypts the table using SAS proprietary encryption. The encryption method uses passwords. You must specify the READ= option or the PW= data set option at the same time that you specify an ENCRYPT=YES option setting. For more information about the data set options, see *SAS Data Set Options: Reference*.

NO
no table encryption is performed. NO is the default setting for the ENCRYPT= option.

AES
specifies AES-256 encryption of data. You must also supply a value for the ENCRYPTKEY= parameter if you choose AES-256 encryption.

Details

Encrypting a server table provides security from users that have system access to dump raw server tables. The section about security in the *SAS Scalable Performance Data Server: Administrator's Guide* contains more information about how to control system access to the server tables.

When the ENCRYPT= table option is set to YES, the 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.

Usage Notes

1. Depending on your query patterns, increasing or decreasing the block size can affect performance.
2. When ENCRYPT=YES, the server encrypts only table row data. Table indexes and metadata are not encrypted.
3. When ENCRYPT=AES, both data and index files are encrypted.
4. To access an encrypted table, the user must have appropriate ACL permissions to the table and must provide either the password or the encryption key. The password is specified with the READ= option or the PW= data set option. The encryption key is specified with the ENCRYPTKEY= table option.

Example

This code creates encrypted server table EncTable from non-encrypted server table RegTable using implicit pass-through. It specifies the ENCRYPT= table option with the password “Secret”.

```
libname tempdata sasspds 'public' server=lax94d01.14545
user='anonymous' ip=yes;

data tempdata.regtable;x=1;run;

option dbidirectexec=yes;

proc sql;
  create table tempdata.enctable(encrypt=yes pw=secret) as
  select * from tempdata.regtable;
quit;
```

To use server table EncTable, specify the password as follows:

```
proc contents data=tempdata.enctable(pw=secret); run;
```

In the output, note that the Encrypted attribute for the table is set to YES.

The SAS System

The CONTENTS Procedure

Data Set Name	TEMPDATA.ENCTABLE	**Observations**	1
Member Type	DATA	**Variables**	1
Engine	SASSPDS	**Indexes**	0
Created	03/24/2016 12:44:14	**Observation Length**	8
Last Modified	03/24/2016 12:44:14	**Deleted Observations**	0
Protection	READ/WRITE/ALTER	**Compressed**	NO
Data Set Type		**Sorted**	NO
Encrypted	YES		
Label			
Data Representation	Default		
Encoding	latin1 Western (ISO)		

Engine/Host Dependent Information	
Blocking Factor (obs/block)	4094
ACL Entry	NO
ACL User Access(R,W,A,C)	(Y,Y,Y,Y)
ACL UserName	CORETEST
ACL OwnerName	CORETEST
ACL GroupName	CORE
Data set is Ranged	NO
Data set is a Cluster	NO

Alphabetic List of Variables and Attributes

#	Variable	Type	Len
1	x	Num	8

See Also

SPD Server table options:

- “ENCRYPTKEY= Table Option” on page 255
- “IOBLOCKSIZE= Table Option” on page 259

ENCRYPTKEY= Table Option

Specifies a key value for AES encryption.

Valid in: SPD Server

Requirement: ENCRYPTKEY= must be specified when ENCRYPT=AES.

Syntax

ENCRYPTKEY=*key-value*

Required Argument

key-value

assigns an encrypt key value. To create an ENCRYPTKEY= key value with or without quotation marks, follow these rules:

No quotation marks:

- use alphanumeric characters and underscores only
- can be up to 64 bytes long
- use uppercase and lowercase letters
- must start with a letter
- cannot include blank spaces
- is not case sensitive

Examples:

```
%let mykey=abcdefghi12;
encryptkey=&mykey
encryptkey=key_value
encryptkey=key_value1
```

Single quotation marks:

- use alphanumeric, special, and DBCS characters
- can be up to 64 bytes long
- use uppercase and lowercase letters
- can include blank spaces, but cannot contain all blanks
- is case sensitive

Examples:

```
encryptkey='key_value'
encryptkey='1234*#mykey'
```

Double quotation marks:

- use alphanumeric, special, and DBCS characters
- can be up to 64 bytes long
- use uppercase and lowercase letters
- can include blank spaces, but cannot contain all blanks
- is case sensitive

Examples:

```
encryptkey="key_value"
encryptkey="1234*#mykey"
%let mykey=Abcdefghi12;
encryptkey="&mykey"
```

Interaction You cannot change the key value on an AES-encrypted table without re-creating the table.

Note When the ENCRYPTKEY= key value uses DBCS characters, the 64-byte limit applies to the character string after it has been transcoded to UTF-8 encoding. You can use the following DATA step to calculate the length in bytes of a key value in DBCS:

```
data _null_;
    key=length(unicodec('key-value','UTF8'));
    put 'key length=' key;
run;
```

Details

When you use the ENCRYPT=AES option setting to specify AES-256 encryption, you must use the ENCRYPTKEY= option to specify a text string value. This value enables the RSA 256-bit encryption key to encode data and index files.

CAUTION:

Record all ENCRYPTKEY= values when you are using ENCRYPT=AES. If you forget to record the ENCRYPTKEY= value, you lose your data. SAS cannot assist you in recovering the ENCRYPTKEY= value.

The ENCRYPTKEY= table option does not protect the file from deletion or replacement.

You must specify the ENCRYPTKEY=value to read or copy the file.

You can use a macro variable as the ENCRYPTKEY= key value. The following code defines a macro variable:

```
%let secret=Abcdefghi12;
```

The following code uses the macro variable as the ENCRYPTKEY= value:

```
data tempdata.aestable(encrypt=aes encryptkey="&secret");
```

Example

This example sets the ENCRYPT=AES option and an encryption key using PROC SQL:

```
libname tempdata sasspds "test" host="host.company.com" service="8561"
    user="siteusr1" prompt=yes;

data tempdata.regtable;x=1;run;

option dbidirectexec=yes;

proc sql;
create table tempdata.aestable(encrypt=aes encryptkey="1234*#mykey") as
select * from tempdata.regtable;
quit;
```

To use the table, specify the ENCRYPTKEY= value as follows:

```
proc contents data=tempdata.aestable(encryptkey="1234*#mykey"); run;
```

See Also

SPD Server table options:

- "ENCRYPT= Table Option" on page 252

ENDOBS= Table Option

Specifies the end row number in a user-defined range for the processing of a given table.

Valid in: SPD Server

Syntax

ENDOBS=*n*

Required Argument

n
specifies the number of the end row.

Details

By default, the server processes the entire table unless the user specifies a range of rows with the STARTOBS= option and the ENDOBS= option. 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. When ENDOBS= is used in a WHERE expression, the ENDOBS= value represents the last observation to process, rather than the number of observations to return. The following examples show the difference.

Examples

Example 1: Use ENDOBS= with STARTOBS=

Create server table TempData.Old_Autos and print only rows 2-4.

```
libname tempdata sasspds 'conversion_area'
   server=husky.5105
   user='siteusr1'
   prompt=yes;

proc sql;
create table tempdata.old_autos(
   Year char(4),
   Manufacturer char(12),
   Model char(10),
```

```
   Body_style char(5),
   Engine_liters num,
   Trans_type char(1),
   Ext_color char(10),
   Options char(10),
   Mileage num,
   Condition num);

insert into tempdata.old_autos values('1971', 'Buick', 'Skylark', 'conv', 5.8,'A',
'yellow', '0000001', 143000, 2);
insert into tempdata.old_autos values('1982', 'Ford', 'Fiesta', 'hatch', 1.2, 'M',
'silver', '0000001', 70000, 3);
insert into tempdata.old_autos values('1975', 'Lancia', 'Beta', '2door', 1.8, 'M',
'drk blue', '0000010', 80000, 4);
insert into tempdata.old_autos values('1966', 'Oldsmobile','Toronado', '2door', 7.0,
'A', 'black', '11000010', 210000, 3);
insert into tempdata.old_autos values('1969', 'Ford', 'Mustang', 'sptrf', 7.1, 'M',
'red', '0000111', 125000, 3);
quit;

proc print data=tempdata.old_autos (startobs=2 endobs=4);
run;
```

The SAS System

Obs	Year	Manufacturer	Model	Body_style	Engine_liters	Trans_type	Ext_color	Options	Mileage	Condition
2	1982	Ford	Fiesta	hatch	1.2	M	silver	0000001	70000	3
3	1975	Lancia	Beta	2door	1.8	M	drk blue	0000010	80000	4
4	1966	Oldsmobile	Toronado	2door	7.0	A	black	11000010	210000	3

Example 2: Use ENDOBS= with a WHERE Clause

Print rows from table TempData.Old_Autos where Trans_type="M" and ENDOBS=2. ENDOBS= specifies to stop processing at row 2 .

```
proc print data=tempdata.old_autos(endobs=2);
where trans_type="M";
run;
```

The SAS System

Obs	Year	Manufacturer	Model	Body_style	Engine_liters	Trans_type	Ext_color	Options	Mileage	Condition
2	1982	Ford	Fiesta	hatch	1.2	M	silver	0000001	70000	3

See Also

SPD Server LIBNAME options:

- “ENDOBS= LIBNAME Statement Option” on page 159

SPD Server macro variables:

- “SPDSEOBS Macro Variable” on page 219

IOBLOCKSIZE= Table Option

Specifies the number of rows in a block to be stored in or read from an SPD Server table.

Valid in: SPD Server

Defaults: 8K compressed

32K encrypted (uncompressed)

Interaction: Use IOBLOCKSIZE= in conjunction with the COMPRESS= or ENCRYPT= table option.

Syntax

IOBLOCKSIZE=*n*

Required Argument

n
the size of the block.

Details

The software reads and stores a server table in blocks. IOBLOCKSIZE= is useful on compressed or encrypted tables. The server software does not use IOBLOCKSIZE= on server tables that are not compressed or encrypted.

For tables that are compressed or encrypted, 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 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. Values smaller than 32K are ignored when encrypting data.

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;
```

See Also

SPD Server table options:

- "COMPRESS= Table Option" on page 250
- "ENCRYPT= Table Option" on page 252

MINMAXVARLIST= Table Option

Creates a list that documents the minimum and maximum values of specified columns.

Valid in: SPD Server

Interaction:

Note: SPD Server WHERE clause evaluations use MINMAXVARLIST= lists to include or eliminate member tables in a server dynamic cluster table from SQL evaluation scans.

See: "MINMAX Variable List" on page 130

Syntax

MINMAXVARLIST=(*variable-name(s)*)

Required Argument

***(variable-name(s)*)**
server table column names. If there is more than one name, separate the names with a space.

Details

The MINMAXVARLIST= table option is used on server tables that will become members of a dynamic cluster table. The option facilitates searches where specific members in the dynamic cluster table contain a set or range of values. An example would be sales data for a given month.

When a 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 server WHERE planner includes or eliminates member tables in the dynamic cluster for evaluation.

MINMAXVARLIST= uses the list of columns that 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 cluster tables, it also works on standard server tables. MINMAXVARLIST=

can help reduce the need to create many indexes on a table, which can save valuable resources and space.

Example

```
%let host=kaboom ;
%let port=5201 ;

LIBNAME mylib sasspds "path3"
   server=&host..&port
   user='anonymous' ;

/* Create three tables called */
/* xy1, xy2, and xy3.         */

data mylib.xy1(minmaxvarlist=(x y));
  do x = 1 to 10;
  do y = 1 to 3;
  output;
  end;
end;
run;

data mylib.xy2(minmaxvarlist=(x y));
  do x = 11 to 20;
  do y = 4 to 6 ;
  output;
  end;
end;
run;

data mylib.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=mylib;
   cluster create cluster_table
      mem=xy1
      mem=xy2
      mem=xy3
     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=mylib.cluster_table ;
   where x eq 3
   and y eq 3;
run;


/* Examine the other tables */

PROC PRINT data=mylib.cluster_table ;
   where x eq 3
   and y eq 3 ;
run;

PROC PRINT data=mylib.cluster_table ;
   where x eq 3
   and y eq 3;
run;

PROC PRINT data=mylib.cluster_table ;
   where x between 1 and 10
   and y eq 3;
run;

PROC PRINT data=mylib.cluster_table ;
   where x between 11 and 30
   and y eq 8 ;
run;


/* Delete the dynamic cluster table. */

PROC DATASETS lib=mylib nolist;
   delete cluster_table ;
quit ;
```

NETPACKSIZE= Table Option

Specifies the size of the SPD Server network data packet.

Valid in: SPD Server

Default: 32768 (32 KB)

Interaction: Corresponding macro variable is SPDSNETP.

Syntax

NETPACKSIZE=*size-of-packet*

Required Argument

size-of-packet
the size of the network packet in bytes

Details

This option controls the size of the buffer used for data transfer between SPD Server and a SAS client. The default is 32 KB. 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, the 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 that it must travel through. An optimum size will flow the data continuously without significant pauses between packets.

Example

Create a 12 KB buffer in the memory of the server to send three rows from MYTABLE in each network packet. (The row size in MYTABLE is 4 KB.)

```
data mylib.mytable (netpacksize=12288);
```

See Also

SPD Server macro variables:

- "SPDSNETP Macro Variable" on page 227

NOINDEX= Table Option

Specifies whether to use the table's indexes when processing WHERE clauses. NOINDEX= can also be used to disable index use for BY order determination.

Valid in: SPD Server

Default: NO

Restriction: NOINDEX= affects index usage for BY ordering only if index usage for BY sorts is enabled on the server. The server parameter that enables BY sorts, BYINDEX, is maintained by the server administrator. Ask your administrator if BYINDEX is set for your site.

Syntax

NOINDEX=YES | NO

Required Arguments

YES
ignores indexes when processing WHERE clauses.

NO
uses indexes when processing WHERE clauses.

Details

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

You created an index for the Type column in table TempData.AudiCars but decide to test whether it is necessary for your processing. You issue a SELECT statement on the table that sets NOINDEX=YES and SELECT statement that sets NOINDEX=NO so that you can compare processing. You also set the SPDSWDEB macro variable.

```
libname tempdata sasspds "conversion_area" server=husky.5105
    user="siteusr1" password="userpwd";

proc sql;
create table tempdata.audicars as
select * from sashelp.cars
where make="Audi";
create index type on tempdata.audicars(type);
quit;

/*Turn on the macro variable SPDSWDEB */
/* to show whether the index is used */
/* during the table processing. */

%let spdswdeb=YES;

proc sql;
    select * from tempdata.audicars(noindex=yes)
where type="Sedan";
    select * from tempdata.audicars(noindex=no)
where type="Sedan";
quit;
```

The following output is written to the SAS log:

```
89   %let spdswdeb=YES;
90
91   proc sql;
92        select * from tempdata.audicars(noindex=yes) where type="Sedan";

whinit: WHERE (Type='Sedan')
whinit: wh-tree presented
           /-NAME = [Type]
 --CEQ----|
           \-LITC = ['Sedan']
whinit returns: ALL EVAL2

93        select * from tempdata.audicars(noindex=no) where type="Sedan";

whinit: WHERE (Type='Sedan')
whinit: wh-tree presented
           /-NAME = [Type]
 --CEQ----|
           \-LITC = ['Sedan']
whinit: wh-tree after split
 -- <empty>
whinit: INDEX Type uses 100% of segs (WITHIN maxsegratio 100%)
whinit: INDEX tree after split
            /-NAME = [Type] <1>INDEX Type (Type)
--CEQ----|
            \-LITC = ['Sedan']
whinit costing: 1 segs with est 1% yield reduces whthreads from 32 to 1
whinit returns: ALL EVAL1(w/SEGLIST)
```

The **`whinit returns`** message shows ALL EVAL1 when the index is used, and ALL EVAL2 when a sequential pass is used.

See Also

SPD Server macro variables:

- "SPDSNIDX Macro Variable" on page 228
- "SPDSWDEB Macro Variable" on page 241

PARTSIZE= Table Option

Specifies the size of an SPD Server table partition.

Valid in: SPD Server

Default: 16 MB for domains that are not Hadoop domains, 128 MB for Hadoop domains

Restriction: The PARTSIZE= specification is limited by MINPARTSIZE=, a server parameter maintained by the server administrator. Ask your administrator what the MINPARTSIZE= setting is for your site. If you use PARTSIZE=, the value of PARTSIZE= must be greater than the value of MINPARTSIZE= to have any effect.

Syntax

PARTSIZE=*n*

Required Argument

n
the size of the partition. The number given is assumed to be in megabytes.

Details

Specifying PARTSIZE= forces the software to partition (split) the 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* megabytes.

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 server table at fixed-sized intervals, the software can introduce a high degree of scalability for these operations. The software can do this by launching 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.

The PARTSIZE= specification is limited by the MINPARTSIZE= server parameter. MINPARTSIZE= ensures that an over-zealous user does not create arbitrarily small partitions, thereby generating a large number of files. When MINPARTSIZE= is omitted from the server parameter file, the default value is 16 MB for domains that are not Hadoop domains and 128 MB for Hadoop domains. These are the absolute minimum recommended settings for each environment. Many sites specify a higher MINPARTSIZE= value in their server parameter file.

Note: The partition size for a table cannot be changed after a table is created. If you must change the partition size, use PROC COPY to duplicate the table and specify a different PARTSIZE= setting on the output table.

Example

Using PROC SQL, create a table with a partition size of 50 MB. If 50 MB is greater than the MINPARTSIZE= setting and you are setting the option for a table that is not in a Hadoop domain, the value will be applied. Otherwise, PARTSIZE= will have no effect.

```
proc sql;
create table SPDSCEN.HR80SPDS(partsize=50)
  as select
    state,
    age,
    sex,
    hour89,
    industry,
    occup
   from SPDSCEN.PRECS
   where hour89 > 40;
quit;
```

See Also

SPD Server macro variables:

- “SPDSSIZE Macro Variable” on page 233

SEGSIZE= Table Option

Specifies the size of the segment for an index file associated with an SPD Server table.

Valid in: SPD Server

Default: 8192 table rows

Syntax

SEGSIZE=*n*

Required Argument

n
the number of table rows to include in the index segment.

Details

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 64 KB for Mylib.Mytable.

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

Note: Tests show that increasing the size of the segment does not significantly increase performance.

STARTOBS= Table Option

Specifies the start row number in a user-defined range for the processing of a given table.

Valid in: SPD Server

Syntax

STARTOBS=*n*

Required Argument

n
is the number of the start row.

Details

By default, the 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. When STARTOBS= is used in a WHERE expression, the STARTOBS= value represents the first observation on which to apply the WHERE expression.

Examples

Example 1: Use STARTOBS= with ENDOBS=

Create server table TempData.Old_Autos and print only rows 2-4.

```
libname tempdata sasspds 'conversion_area'
   server=husky.5105
   user='siteusr1'
   prompt=yes;

proc sql;
create table tempdata.old_autos(
   Year char(4),
   Manufacturer char(12),
   Model char(10),
   Body_style char(5),
   Engine_liters num,
   Trans_type char(1),
   Ext_color char(10),
   Options char(10),
   Mileage num,
   Condition num);

insert into tempdata.old_autos values('1971', 'Buick', 'Skylark', 'conv', 5.8,'A',
'yellow', '0000001', 143000, 2);
insert into tempdata.old_autos values('1982', 'Ford', 'Fiesta', 'hatch', 1.2, 'M',
'silver', '0000001', 70000, 3);
insert into tempdata.old_autos values('1975', 'Lancia', 'Beta', '2door', 1.8, 'M',
'drk blue', '0000010', 80000, 4);
insert into tempdata.old_autos values('1966', 'Oldsmobile','Toronado', '2door', 7.0,
'A', 'black', '11000010', 210000, 3);
insert into tempdata.old_autos values('1969', 'Ford', 'Mustang', 'sptrf', 7.1, 'M',
'red', '0000111', 125000, 3);
quit;

proc print data=tempdata.old_autos (startobs=2 endobs=4);
run;
```

The SAS System

Obs	Year	Manufacturer	Model	Body_style	Engine_liters	Trans_type	Ext_color	Options	Mileage	Condition
2	1982	Ford	Fiesta	hatch	1.2	M	silver	0000001	70000	3
3	1975	Lancia	Beta	2door	1.8	M	drk blue	0000010	80000	4
4	1966	Oldsmobile	Toronado	2door	7.0	A	black	11000010	210000	3

Example 2: Use STARTOBS= with a WHERE Clause

Print rows from table TempData.Old_Autos where Trans_type="M" and STARTOBS=4. STARTOBS= specifies to begin applying the WHERE expression at row 4.

```
proc print data=tempdata.old_autos(startobs=4);
where Trans_type="M";
run;
```

The SAS System

Obs	Year	Manufacturer	Model	Body_style	Engine_liters	Trans_type	Ext_color	Options	Mileage	Condition
5	1969	Ford	Mustang	sptrf	7.1	M	red	0000111	125000	3

See Also

SPD Server LIBNAME options:

- "STARTOBS= LIBNAME Statement Option" on page 172

SPD Server macro variables:

- "SPDSSOBS Macro Variable" on page 235

SYNCADD= Table Option

Specifies to process one row at a time or multiple rows at a time when adding rows.

Valid in: SPD Server

Default: NO

Interaction: Corresponding macro variable is SPDSSADD.

Tip: Use the UNIQUESAVE table option (or SPDSUSAV macro variable) to save rejected rows when SYNCADD=NO.

Syntax

SYNCADD=YES | NO

Required Arguments

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

NO
processes multiple rows at a time (asynchronously).

Details

When SYNCADD= is set to YES, rows are processed one at a time. With PROC SQL, if you are inserting rows into a table with a unique index, and SPD Server encounters a row with a non-unique value, the following occurs:

- the insert operation stops

- all transactions just added are backed out
- the original data set on disk is unchanged.

SYNCADD=NO is faster. However, SYNCADD=NO also handles unique indexes differently. If a non-unique value is found when inserting rows into a table that has a unique index, the following occurs:

- SPD Server rejects the row
- SPD Server continues processing
- a status code is issued only at the end of the insert operation.

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

Example

In the following example, two identical tables, WITH_NO and WITH_YES, are created. Both have a unique index. PROC SQL is used to insert three new rows, one of which has duplicate values. The SYNCADD=YES option is used. PROC SQL stops when the duplicate values are encountered and restores the table. PROC SQL is used again to insert these three new rows (as before). In this case, the SYNCADD=NO option is used. The row with duplicate values is rejected. The SAS log is shown:

```
301  libname tempdata sasspds "conversion_area" server=husky.5105
302       user="siteusr1" password=XXXXXXXXX;
NOTE: User siteusr1(ACL Group CORE) connected to SPD(LAX) 5.3 server at
10.24.7.79.
NOTE: Libref TEMPDATA was successfully assigned as follows:
      Engine:        SASSPDS
      Physical Name: :3030/bigdisk/lax/pubs_d4spds53/test_domains/husky/
conversion_area/
303
304  data tempdata.with_no(index=(x /unique))
305       tempdata.with_yes(index=(x /unique));
306     input z $ 1-20 x y;
307  list;
308  datalines;

RULE:      ----+----1----+----2----+----3----+----4----+----5----+----6----
+----7----+----8----+----9----+----0
309          one                 1 10
310          two                 2 20
311          three               3 30
312          four                4 40
313          five                5 50
NOTE: The data set TEMPDATA.WITH_NO has 5 observations and 3 variables.
NOTE: The data set TEMPDATA.WITH_YES has 5 observations and 3 variables.
314  ;
315  run;
316
317
318  proc sql;
319    insert into tempdata.with_yes(syncadd=yes)
320       values('six_yes', 6, 60)
321       values('seven_yes', 2, 70)
322       values('eight_yes', 8, 80)
323
324  ;
ERROR: Duplicate values not allowed on index x for file WITH_YES.
NOTE: This insert failed while attempting to add data from VALUES clause 2 to
the data set.
NOTE: Deleting the successful inserts before error noted above to restore table
to a consistent state.
325  quit;
326
327  proc sql;
328    insert into tempdata.with_no(syncadd=no)
329       values('six_no', 6, 60)
330       values('seven_no', 2, 70)
331       values('eight_no', 8, 80)
332  ;
NOTE: 3 rows were inserted into TEMPDATA.WITH_NO.

WARNING: Duplicate values not allowed on index x for file TEMPDATA.WITH_NO.
(Occurred 1 times.)
333  quit;
334
335  proc compare data=tempdata.with_no compare=tempdata.with_yes;
336  run;

NOTE: There were 7 observations read from the data set TEMPDATA.WITH_NO.
NOTE: There were 5 observations read from the data set TEMPDATA.WITH_YES.
```

See Also

SPD Server macro variables:

- "SPDSSADD Macro Variable" on page 233

SPD Server table options:

THREADNUM= Table Option

Specifies the number of threads to be used for WHERE clause evaluations.

Valid in: SPD Server

Default: THREADNUM= is set equal to the value of the MAXWHTHREADS server parameter.

Interactions: Corresponding macro variable is SPDSTCNT.
THREADNUM= is affected by the MAXWHTHREADS= server parameter.

Syntax

THREADNUM=*n*

Required Argument

n
the number of threads.

Details

THREADNUM= enables you to specify the thread count that the 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 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 to parallel table scans (EVAL2 strategy), parallel indexed evaluations (EVAL1 strategy), parallel BY-clause processing, and parallel GROUP_BY evaluations. See "Optimizing WHERE Clauses" on page 128.

Example

The server administrator set MAXWHTHREADS=128 in the server 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;
```

See Also

SPD Server macro variables:

- "SPDSTCNT Macro Variable" on page 237

UNIQUESAVE= Table Option

Specifies to save rows that contain duplicate values (which would be rejected) when appending or inserting data in tables with unique indexes.

Valid in: SPD Server

Default: NO

Interactions: Corresponding macro variable is SPDSUSAV.

Use in conjunction with the SPDSUSDS reserved macro variable.

Note: UNIQUESAVE= has no effect when the SYNCADD= table option (or SPDSSADD macro variable) is set to YES.

Syntax

UNIQUESAVE=YES | NO | REP

Required Arguments

YES
writes rejected rows to a separate, system-created table. This table can be accessed by a reference to the macro variable SPDSUSDS.

NO
ignores duplicate rows rejected by an append or insert operation.

REP
replaces the current row in the master table with the duplicate row from the insert or append operation, instead of saving the rows to a separate table. This setting is useful when updating a master table from a transaction table, where the two tables share identical column structures.

Details

When the SPDSAUNQ macro variable is set to NO (the default value), rows with duplicate index values are rejected unless you specify UNIQUESAVE=YES (or set the SPDSUSAV macro variable to YES). By using UNIQUESAVE=YES, you can save rejected values to a hidden system table. When UNIQUESAVE=YES, a NOTE on the log identifies the name of the table. To access that table, you can either cut-and-paste from the log, or refer to that table by using the reserved macro variable SPDSUSDS.

Examples

Example 1: Using UNIQUESAVE=YES

The following example creates three tables that contain employee names. You create a unique index for the table, NAMES1. Then you append table NAMES2 to the NAMES1 table. In the append, you specify UNIQUESAVE=YES to store any rejected rows in a system file. When that operation is complete, you specify to append table NAMES3 to table NAMES1. UNIQUESAVE=YES does not support appends from multiple files.

```
libname employee sasspds "conversion_area" server=husky.5105
    user="siteusr1" prompt=yes;

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;
```

```
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.
```

This is the result of the request to print table NAMES1:

The NAMES1 table with unique namesfrom NAMES2

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

This is the result of the request to print the rows rejected by the append operation.

Duplicate (nonunique) name rows found inNAMES2

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

Example 2: Using UNIQUESAVE=REP

This example creates two tables, TempData.Master and TempData.Trans. It uses the UNIQUESAVE=REP option to replace the current row in the Master table with the duplicate row from table Trans.

```
libname tempdata sasspds "conversion_area" server=husky.5105
    user="siteusr1" prompt=yes;


/* Create a Master table to update. */
/* ID will get a UNIQUE index */
```

```
data tempdata.master;
  input ID value $;
  cards;
   1 one
   2 two
   3 three
;

proc datasets lib=tempdata;
modify master;
index create ID/unique;
quit;

/* Create transaction table Trans to use to */
/* drive update/appends to Master */
data tempdata.trans;
  input ID value $;
  cards;
   1 ONE
   3 THREE
   4 FOUR
   4 FOUR*
;
/* Use 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=tempdata.trans
out=tempdata.master(uniquesave=rep);
run;

proc print data=tempdata.master; run;
```

The SAS log provides the following message:

```
NOTE: Appending TEMPDATA.TRANS to TEMPDATA.MASTER.
NOTE: There were 4 observations read from the data set TEMPDATA.TRANS.
NOTE: 1 observations added.
NOTE: The data set TEMPDATA.MASTER has 4 observations and 2 variables.
```

The resulting Master table looks like this:

Append with UNIQUESAVE=REP

Obs	ID	value
1	1	ONE
2	2	two
3	3	THREE
4	4	FOUR*

See Also

SPD Server macro variables:

- "SPDSUSAV Macro Variable" on page 237
- "SPDSUSDS Reserved Macro Variable" on page 208

SPD Server table options:

- "SYNCADD= Table Option" on page 269

VERBOSE= Table Option

Specifies whether the CONTENTS procedure output includes details about compressed blocks, data partition size, and indexes.

Valid in: SPD Server

Default: NO

Interaction: Corresponding macro variable is SPDSVERB.

Syntax

VERBOSE=YES | NO

Required Arguments

YES
requests detail information about compressed blocks, data partition size, and indexes. This option can be used only with PROC CONTENTS.

NO
suppresses detail information about compressed blocks, data partition size, and indexes. This is the default value.

Example

Request details of the compressed blocks, data partition size, and indexes for the table CLASS.

```
PROC CONTENTS data=mylib.class(verbose=yes);
run;
```

Output 22.3 *Output 14.4: Verbose Details for the Table CLASS*

```
Engine/Host Dependent Information

             Blocking Factor (obs/block)              992
             ACL Entry                                NO
             ACL User Access(R,W,A,C)                 (Y,Y,Y,Y)
             ACL UserName                             ANONYMOU
             ACL OwnerName                            ANONYMOU
             Data set is Ranged                       NO
             Data set is a Cluster                    NO
             Alphabetic List of Index Info            .
             Index                                    Name
             KeyValue (Min)                           Alfred
             KeyValue (Max)                           William
             # of Discrete values                     19
             Index                                    age_sex
             KeyValue (Min)                           11.000000   ,F
             KeyValue (Max)                           16.000000   ,M
             # of Discrete values                     11
             Compressed Info                          .
             # Compressed blocks                      1
             Raw data blocksize                       32736
             # blocks with overflow                   0
             Max overflow chain len                   0
             Block # for max chain                    0
             Min overflow area                        0
             Max overflow area                        0
             Data Partsize                            16793568
```

See Also

SPD Server macro variables:

- "SPDSVERB Macro Variable" on page 239

WHERENOINDEX= Table Option

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

Valid in: SPD Server

Default: No

Syntax

WHERENOINDEX=(*name(s)*)

Required Argument

name(s)

a list of index names that you want to exclude from the WHERE planner. If more than one name, separate with a space.

Example: Using the WHERNOINDEX= Table Option

You have a table PRECS with indexes defined as follows:

```
PROC DATASETS lib=spdscen;
modify precs(index=(hour89));
index create
  stser=(state serialno)
  occind=(occup industry)
  hour89;
quit;
```

When evaluating the next query, you want the server to exclude from consideration indexes for both the STATE and HOUR89 columns.

In this case, you 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.

Chapter 23

SPD Server Access Library API Reference

Introduction to Access Using Library API

This chapter describes the SPD Server SQL access library API (application programming interface) and provides some simple examples. This chapter refers to the SPD 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 the server's SQL processor. Because the library was designed for multi-threaded applications, the code is thread-safe, except where noted in the following sections.

For more information, see Chapter 23, "SPD Server Access Library API Reference," on page 281.

Overview of SPQL Usage

SPQL enables you to write application programs that can connect to and access SPD Server hosts using the SQL language. SPQL is based on connections, enabling you to submit SQL statements to one or more server's SQL processors that execute SQL statements on your behalf.

SPQL API Description

The C-language H file spql.h is provided for customer-written applications. It describes the programming interfaces that are required for user-written programs that access SPD Server SQL. This chapter describes the API functions, their use, and restrictions.

SPQL Library

The SPQL library for SAS SPD Server is available from the SAS support website. Navigate to the SPD Server Downloads and Hot Fixes page on support.sas.com, and then navigate to the appropriate platform link for your version of SAS Scalable Performance Data Server. Download and install the appropriate **spdsclntlibs** client library for your installation.

The spdclntlibs download contains the SPQL spdslib library, a set of message files needed by the library, the **`spql.h`** header file needed to write an SPQL program, and a sample **`spqlsample.c`** program that you can use to test the SPQL library.

SPQL Function Return Codes

Some SPQL functions generate return codes, enabling 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)].

List of SPQL Function Return Codes

The following table contains the function return code and a description.

Function Return Code	Description
SPQL_SUCCESS(==0)	Successful completion of the SPQL function call.
SPQL_ENDDATA (WARNING)	All rows selected were read from the statement token.

Function Return Code	Description
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 the server's SQL processor. 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.

SPQL API Functions

The following sections describe the SPQL API functions.

Dictionary

SPQLCOLINFO() API Function

Gets column information from a statement token.

Valid in: SPD Server

Syntax

int **spqlcolinfo**(void *stmttok, int *ncols, spqlcinfo_t **colvec)

Required Arguments

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.

Details

INT 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.

Returns: 0 if successful.

SPQLCONNECT() API Function

Establishes a connection to a specified server's SQL processor.

Valid in: SPD Server

Syntax

int **spqlconnect(char *constr, void **contok)**

Required Arguments

constr
specifies all the connection information needed to establish the connection to the server's SQL processor. When a connection is made successfully, a connection, token (**contok**) is returned to the caller.

char *constr
A null-terminated string identifying the server's SQL processor 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. For more information about SQL pass-through CONNECT statements, see "Specify SQL Options by Using Explicit Pass-Through Code" on page 78.

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.

Details

Returns: 0 if successful; SPQL_NOMEM if unable to allocate memory for the connection token; SPQL_CONFAILED if unable to connect successfully to the server's SQL processor.

SPQLDISCONNECT() API Function

Terminates a connection from the server's SQL processor specified with an SPQLDISCONNECT().

Valid in: SPD Server

Syntax

int **spqldisconnect**(void *contok)

Required Argument

void *contok
connection token previously obtained from **SPQLCONNECT()**.

Details

INT disconnects from a specified server's SQL processor. The caller passes the connection token that was returned from an **SPQLDISCONNECT()** call. Then, the server's SQL processor associated with the connection is disconnected from the caller, and the memory associated with connection token is returned to the system.

Returns: 0 if successful.

SPQLFETCH() API Function

Gets row data from a statement token.

Valid in: SPD Server

Syntax

int **spqlfetch**(void *stmttok, void **bufptr, int *bufsize)

Required Arguments

int

void *stmttok
the statement token to use to access row data from the SELECT statement.

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 result set buffer.

int *bufsize
returns the size of the row buffer that was returned to the caller.

Details

INT fetches each row that an executing statement returns. 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.

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.

SPQLFREESTOK() API Function

Frees resources used by a previously performed statement.

Valid in: SPD Server

Syntax

int **spqlfreestok(**int void *stmttok)

Required Argument

void *stmttok
statement token to free resources.

Details

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() function are read. If you do, the remaining unread rows (from the previous select) are discarded.

Returns: 0 if successful.

SPQLGMSG() API Function

Accesses thread-specific error or diagnostic message buffer contents.

Valid in: SPD Server

Syntax

int **spqlgmsg**(char **mbuf)

Required Argument

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.

Details

The SPQLGMSG() function returns a pointer to the threads error or diagnostic message buffer. Call **SPQLGMSG()** function to get any diagnostic messages if you encounter an error executing an SPQL function. If there is message information, SPQLGMSG() function returns the message pointer in the **mbuf** parameter as well as the length of the message (the function return value).

SPQLINIT() API Function

Initializes the SPQL library for operation.

Valid in: SPD Server

Syntax

int **spqlinit**(void)

Required Argument

(void)
performs a one-time initialization that enables the SPQL library to function.

Details

SPQLINIT() function performs a one-time initialization that enables the SPQL library to function. For this reason, you must call SPQLINIT() function 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() function 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.

Returns: 0 if successful; SPQL_INITFAILED if the initialization fails.

SPQLPERFORM() API Function

Submits an SQL statement for execution on a given connection.

Valid in: SPD Server

Syntax

int **spqlperform**(void *contok, char *stmtbuf, int stmtlen, int *actions, void **stmttok)

Required Arguments

void *contok
connection used to execute the SQL statement.

int stmtlen
length of the SQL statement in buffer; -1 if null-terminated.

char *stmtbuf
buffer that holds the SQL statement to perform.

int *actions
returns post-processing notification 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.

Details

SPQLPERFORM() function 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())
```

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() function cannot allocate memory for the statement token.

SPQLTABINFO() API Function

Gets table information from a statement token.

Valid in: SPD Server

Syntax

int **spqltabinfo**(void *stmttok, spqltinfo_t **tinfo)

Required Arguments

void *stmttok
statement token to use to access table information from a 'select'.

spqltinfo **tinfo
returns pointer to **spqltinfo_t** structure into the statement token memory.

Details

INT 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.

Returns: 0 for successful completion.

SPQLTERM() API Function

Is the termination counterpart of the spqlinit() function.

Syntax

int **spqlterm**(void)

Required Argument

(void)
terminates the SPQL library session.

Details

SPQLTERM() function terminates the SPQL library session, disconnecting all active server's SQL processor connections and freeing up the memory resources associated with the SPQL run-time library executables.

Returns: 0 if successful.

Chapter 24

National Language Support

National Language Support

Overview of National Language Support

National Language Support (NLS) is a set of features that enable a software product to function properly in every global market for which the product is targeted. The SAS System contains NLS features to ensure that SAS applications can be written so that they conform to local language conventions. Typically, software that is written in the English language works well for users who use the English language and use data that is formatted using the conventions that are observed in the United States. However, without NLS, these products might not work well for users in other regions of the world. NLS in SAS enables users in regions such as Asia and Europe to process data successfully in their native languages and environments.

For information and detailed syntax for all SAS language elements that contain NLS properties, see *SAS National Language Support (NLS): Reference Guide*.

SPD Server NLS Support

SPD Server supports a subset of the SAS System NLS functionality. The server supports encoding and locale.

- An encoding maps each character in a character set to a unique numeric representation, which results in a table of all code points. This table is referred to as 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.
- A locale reflects the language, local conventions such as data formatting, and culture for a geographical region. Local conventions might include specific formatting rules for dates, times, and numbers and a currency symbol for the country or region.

Collating sequence, paper size, postal addresses, and telephone numbers can also be included in locale.

All tables that are produced by the server and SAS inherit the SAS session's default encoding and locale settings. By default, the 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 server host, unless server transcoding has been disabled.

SPD Server NLS Limitations

Affected Data

The server hosts are restricted in how they handle NLS character strings. Server hosts are restricted to data that is contained in character columns in tables and some metadata structures. Server hosts store table and column labels using the NLS encoding that they were created in. If a SAS session that uses a different NLS encoding requests server table data, the label names are not transcoded for printing or logging.

Column names, index names, table names, and catalog names are not supported in the server NLS. Column names, index names, table names, and catalog names are still dependent on ASCII support. The server SQL is subject to the same NLS restrictions.

SQL Explicit Pass-Through

SPD Server SQL explicit pass-through does not support NLS. SQL explicit pass-through operates in the encoding and locale of the SAS session that initiates the CONNECT statement to the SASSPDS engine.

Case Folding and Sort Sequences

SPD Server NLS code supports 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 are restricted to lexical sorts for all combinations. The server does not support linguistic sorting.

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 server host table's encoding and locale settings, index use is unrestricted. Otherwise, index use 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 functions and formats that produce or accept textual date, time, and date/time representations are not locale-sensitive.

INENCODING= and OUTENCODING= Options

The INENCODING= and OUTENCODING= LIBNAME statement options override and change the encoding when reading or writing data. These options are not supported and produce a Warning message if submitted for the SASSPDS engine.

ENCODING= Option

The ENCODING= data set option, which overrides and transcodes the encoding of external files, is supported for output processing only. Character data is assumed to be in the encoding of the session that initiates the CONNECT to SASSPDS engine. If you specify ENCODING= for a table that is not an output table and the specified value does not match the table's encoding, the server produces a Warning message. The message states that the ENCODING= option is ignored and that the encoding values do not match.

TRUNCWARN= LIBNAME Statement Option

The TRUNCWARN= LIBNAME statement option for the SASSPDS engine suppresses hard failure on transcoding overflow and character mapping errors. The default setting is NO, which causes hard Read and Write stops when transcoding overflow or mapping errors are encountered. When you use TRUNCWARN=YES, data integrity can be compromised, because significant characters can be lost. When TRUNCWARN=YES is specified and an overflow or character mapping error occurs, a Warning message is displayed in the SAS log when the table is closed if overflow occurs, but the data overflow is lost.

Part 6

ODBC and JDBC Clients

Chapter 25

Using SPD Server with ODBC and JDBC Clients

Introduction to Access Using ODBC and JDBC

SPD Server provides ODBC (Open Database Connectivity) and JDBC (Java Database Connectivity) access to SPD Server data stores from all supported platforms. When the appropriate drivers are installed on the network, SPD Server allows queries on tables from third-party applications that do not use SAS software.

Using ODBC to Access SPD Server Tables

To access SPD Server tables from ODBC:

1. Download and install the SAS ODBC Driver from the SAS Support site Downloads: SAS Drivers for ODBC.
2. Download and install the SPD Server Data Client Libraries from the SAS Support site Downloads: SAS Scalable Performance Data Server. See the SPD Server Data Client Libraries ReadME for more details.
3. Copy the spds.dll and SPD Server message files (*.m files) from the SPD Server Data Client Libraries installation to the folder or directory where you installed the SAS ODBC Driver.
4. Configure your application to connect to SPD Server. For more information, see “Setting Up a Connection to SPD Server” in Chapter 2 of the SAS 9.4 Drivers for ODBC on the SAS support site.

Using JDBC to Access SPD Server Tables

Access SPD Server Tables from JDBC

JDBC access to SPD Server is performed through the SPD Server SNET process. Review your server start-up logs to verify that the spdssnet process is running.

To access SPD Server Tables from JDBC:

1. Download and install the SAS JDBC Driver from Downloads: SAS Drivers for JDBC.
2. Configure the SAS JDBC driver properties file to use the SPD Server SNET process as its sharenet server.
3. Provide a server connection string to the server schema (domain) that contains your server tables.

JDBC Properties File Configuration Example

The following example configures a SAS JDBC driver properties file to connect to the server SNET process that is running at port 5401 on host **myhost.unx.sas.com** to access tables for a server running on host myhost.unx.sas.com at port 5400:

```
//CONNECT TO THE SPD SERVER HOST BY USING A CONNECTION PROPERTY LIST
      Class.forName("com.sas.net.sharenet.ShareNetDriver");
      props = new Properties();
      props.setProperty("dbms", "SPDS");
      props.setProperty("dbmsOptions", "dbq='spdstmp' host='myhost.unx.sas.com'
          serv='5400' ");
      props.setProperty("shareUser","spduser");
      props.setProperty("sharePassword", "spdpw");
      props.setProperty("shareRelease", "V9");
      connection = DriverManager.getConnection(
          "jdbc:sharenet://myhost.unx.sas.com:5401", props);
```

In the code:

dbms
set to SPD Server.

dbms_options
the server domain name (**spdstmp**), the server host name (**myhost.unx.sas.com**), and the port number (**5400**) assigned to the name server.

shareUser
the server user ID.

sharePassword
the server user password.

shareRelease
is set to **V9**.

jdbc:sharenet
the server SNET process host name and port number.

For more detailed information about JDBC connections for server data, see SAS 9.4 Drivers for JDBC Cookbook. Chapter 4 contains connection recipe information for accessing DBMS and SPD Server data.

Recommended Reading

Here is the recommended reading list for this title:

- *SAS Scalable Performance Data Server: Processing Data in Hadoop*
- *SAS Scalable Performance Data Server: Administrator's Guide*
- *SAS SQL Procedure User's Guide*
- *SAS 9.4 DS2 Language: Reference, Sixth Edition*
- *SAS 9.4 FedSQL Language: Reference, Fifth Edition*
- *SAS 9.4 Procedures Guide, Sixth Edition*
- *The Little SAS Book: A Primer*

For a complete list of SAS publications, go to sas.com/store/books. If you have questions about which titles you need, please contact a SAS Representative:

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025
Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: sas.com/store/books

Glossary

access control list (ACL)
a list of users and permission types that each user has for a data resource such as a file, directory, or table.

ACL
See access control list.

authentication
See client authentication.

Base SAS
the core product that is part of SAS Foundation and is installed with every deployment of SAS software. Base SAS provides an information delivery system for accessing, managing, analyzing, and presenting data.

big data
information (both structured and unstructured) of a size, complexity, variability, and velocity that challenges or exceeds the capacity of an organization to handle, store, and analyze it.

block
a group of observations in a data set. By using blocks, thread-enabled applications can read, write, and process the observations faster than if they are delivered as individual observations.

Certificate Revocation List (CRL)
a list of revoked digital certificates. CRLs are published by Certification Authorities (CAs), and a CRL contains only the revoked digital certificates that were issued by a specific CA.

client authentication (authentication)
the process of verifying the identity of a person or process for security purposes. Authentication is commonly used in providing access to software, and to data that contains sensitive information.

component file
any of several file types in a logical file structure that is tracked and indexed as a single table. Each SPD Server table includes a metadata file (.mdf), at least one data file (.dpf), and might also include index files (.hbx or .idx).

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

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

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

CRL
See Certificate Revocation List.

data partition
a physical file that contains data and which is part of a collection of physical files that comprise the data component of a table. *See also* partition.

data resource
any of a collection of domains, tables, catalogs and other types of data that users (with permissions) can access with SPD Server.

directory cleanup utility (spdsclean)
a component of SPD Server that performs routine maintenance functions on directories.

distinguished name (DN)
a unique identifier of an entry in an LDAP network directory. In effect, a distinguished name is the path to the object in the directory information tree.

distributed data
data that is divided and stored across multiple connected computers.

distributed locking
provides synchronization and group coordination services to clients over a network connection. The service provider is the Apache ZooKeeper coordination service, specifically the implementation of the recipe for Shared Lock that is provided by Apache Curator.

DN
See distinguished name.

domain
for SPD Server, a specific directory of file storage locations. The SPD Server Administrator defines the domain in the libnames.parm parameter file and assigns a name. Users connect to the SPD Server domain by specifying the domain name, for example, in the LIBNAME statement for the SASSPDS engine.

dynamic cluster table
two or more SPD Server tables that are virtually concatenated into a single entity, using metadata that is managed by the SAS SPD Server.

dynamic locking
provides multiple users concurrent access to tables. Users can perform read and write functions, and the integrity of the table contents is preserved. Clients that use dynamic locking connect to a separate SPD user proxy process for each connection in the domain.

explicit pass-through
a form of the SQL pass-through facility that passes the user-written, DBMS-specific SQL query code directly to a particular DBMS for processing. *See also* implicit pass-through.

firewall
a set of related programs that protect the resources of a private network from users from other networks. A firewall can also control which outside resources the internal users are able to access.

format
See SAS format.

function
See SAS function.

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

implicit pass-through
a form of the SQL pass-through facility that translates SAS SQL query code to the DBMS-specific SQL code, enabling the translated code to be passed to a particular DBMS for processing. *See also* explicit pass-through.

informat
See SAS informat.

JAR (Java Archive)
the name of a package file format that is typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform.

Java Archive
See JAR.

Java Database Connectivity (JDBC)
a standard interface for accessing SQL databases. JDBC provides uniform access to a wide range of relational databases. It also provides a common base on which higher-level tools and interfaces can be built.

JDBC
See Java Database Connectivity.

LDAP (Lightweight Directory Access Protocol)
a protocol that is used for accessing directories or folders. LDAP is based on the X.500 standard, but it is simpler and, unlike X.500, it supports TCP/IP.

libnames.parm file
an SPD Server parameter file that defines the domains by establishing the names and file storage locations for data resources. The file also serves as a tool for controlling access to the domains and for managing storage of SPD Server files.

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

Lightweight Directory Access Protocol
See LDAP.

macro variable (symbolic variable)
a variable that is part of the SAS macro programming language. The value of a macro variable is a string that remains constant until you change it.

name server
an SPD Server server process that converts domain names to data storage locations.

national language support (NLS)
the set of features that enable a software product to function properly in every global market for which the product is targeted.

NLS
See national language support.

ODBC
See Open Database Connectivity.

Open Database Connectivity (ODBC)
an interface standard that provides a common application programming interface (API) for accessing data. Many software products that run in the Windows operating environment adhere to this standard so that you can access data that was created using other software products.

parallel execution
See parallel processing.

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

parallel processing (parallel execution)
a method of processing that divides a large job into multiple smaller jobs that can be executed simultaneously on multiple CPUs. *See also* threading.

parameter file
a document that contains the data that SPD Server needs to perform its functionality. SPD Server uses a libnames.parm parameter file and an spdsserv.parm parameter file.

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

pass-through facility
See SQL pass-through facility.

password database
registers users to enable access to SPD Server. The database stores each user ID and password, a user's ACL group memberships, authorization level, performance class, account expiration information, and server access records.

password database utility
a component that creates and manages the password database, and that enables users to access SPD Server. It is an interactive command-line utility that begins with the psmgr command.

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

RAID (redundant array of independent disks)
a type of interleaved storage system that comprises multiple disks to store large amounts of data inexpensively. RAIDs can have several levels. For example, a level-0 RAID combines two or more hard drives into one logical disk drive. Various RAID levels provide differing amounts of redundancy and storage capability. Also, because the same data is stored in different places, I/O operations can overlap, which can result in improved performance. *See also* redundancy.

record-level locking
locking at the record level in a table or data set. The user who owns the lock has exclusive access to a single record, while other users can access other records in the same table or data set.

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

redundant array of independent disks
See RAID.

RLS
See row-level security.

row-level security (RLS)
a security feature that controls access to rows in a table in order to prevent users from accessing restricted data.

SAS format (format)
a type of SAS language element that is used to write or display data values according to the data type: numeric, character, date, time, or timestamp.

SAS function (function)
a type of SAS language element that is used to process one or more arguments and then to return a result that can be used in either an assignment statement or an expression.

SAS informat (informat)
a type of SAS language element that is used to read data values according to the data's type: numeric, character, date, time, or timestamp.

SAS Management Console
a Java application that provides a single user interface for performing SAS administrative tasks.

SAS Metadata Server
a multi-user server that enables users to read metadata from or write metadata to one or more SAS Metadata Repositories.

SAS Scalable Performance Data Server (SPD Server)
a server that restructures data in order to enable multiple threads, running in parallel, to read and write massive amounts of data efficiently.

sasroot
a representation of the name for the directory or folder in which SAS is installed at a site or a computer.

SASSPDS
the SAS engine that provides access to the SAS SPD Server.

scalability
the ability of a software application to function well and with minimal loss of performance, despite changing computing environments, and despite changes in the volume of computations, users, or data. Scalable software is able to take full advantage of increases in computing capability such as those that are provided by the use of SMP hardware and threaded processing. *See also* scalable software, server scalability.

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

Secure Sockets Layer
See SSL.

serde
an interface that enables serialization or deserialization of one or more file formats.

server scalability
the ability of a server to take advantage of SMP hardware and threaded processing in order to process multiple client requests simultaneously. That is, the increase in

computing capacity that SMP hardware provides increases proportionately the number of transactions that can be processed per unit of time. *See also* threaded processing.

session
a single period during which a software application is in use, from the time the application is invoked until its execution is terminated.

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

sort indicator
an attribute of a data file that indicates whether a data set is sorted, how it was sorted, and whether the sort was validated. Specifically, the sort indicator attribute indicates the following information: 1) the BY variable(s) that were used in the sort; 2) the character set that was used for the character variables; 3) the collating sequence of character variables that was used; 4) whether the sort information has been validated. This attribute is stored in the data file descriptor information. Any SAS procedure that requires data to be sorted as a part of its process uses the sort indicator.

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

SPD Server
See SAS Scalable Performance Data Server.

SPD Server STARJOIN Facility (STARJOIN Facility)
a component of the SPD Server SQL Planner that optimizes N-way star schema joins for qualified SPD Server tables.

SPDO procedure
the operator interface for SPD Server. The procedure defines and manages SPD Server ACLs, defines row-level security for tables, manages proxies, defines and manages cluster tables, refreshes server parameters and domains, performs table management functions such as truncating tables, and executes SPD Server utilities from a central point.

SPDSBASE process
accesses or creates SPD Server resources for users. Several SPDSBASE processes can be active simultaneously in an SPD Server installation, handling work requests for different users or different SAS sessions. The SPDSBASE process can take on the role of either an SPD Server user proxy, an SPD Server SQL proxy, or an SPD Server SQL user proxy.

spdsclean
See directory cleanup utility.

spdsserv.parm file
an SPD Server parameter file that defines the server configuration and performance parameters that control processing behavior and use of resources.

SQL pass-through facility (pass-through facility)
the technology that enables SQL query code to be passed to a particular DBMS for processing. *See also* record-level locking.

SQL query rewrite facility
examines SQL queries to optimize processing performance. When an SPD Server user submits SQL statements that contain subexpressions, the SQL query rewrite facility optimizes the SQL query when possible.

SSL (Secure Sockets Layer)
an encryption protocol for securing client-server communication. *See also* Transport Layer Security.

star schema
tables in a database in which a single fact table is connected to multiple dimension tables. This is visually represented in a star pattern. SAS OLAP cubes can be created from a star schema.

STARJOIN Facility
See SPD Server STARJOIN Facility.

symbolic variable
See macro variable.

symmetric multiprocessing
See SMP.

thread
the smallest unit of processing that can be scheduled by an operating system.

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

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

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

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

threading
a high-performance technology for either data processing or data I/O in which a task is divided into threads that are executed concurrently on multiple cores on one or more CPUs.

time to solution
the elapsed time that is required for completing a task. Time-to-solution measurements are used to compare the performance of software applications in

different computing environments. In other words, they can be used to measure scalability. *See also* scalability.

TLS
See Transport Layer Security.

Transport Layer Security (TLS)
the successor to Secure Sockets Layer (SSL), a cryptographic protocol that is designed to provide communication security. TLS uses asymmetric cryptography for authentication and confidentiality of the key exchange, symmetric encryption for data/message confidentiality, and message authentication codes for message integrity.

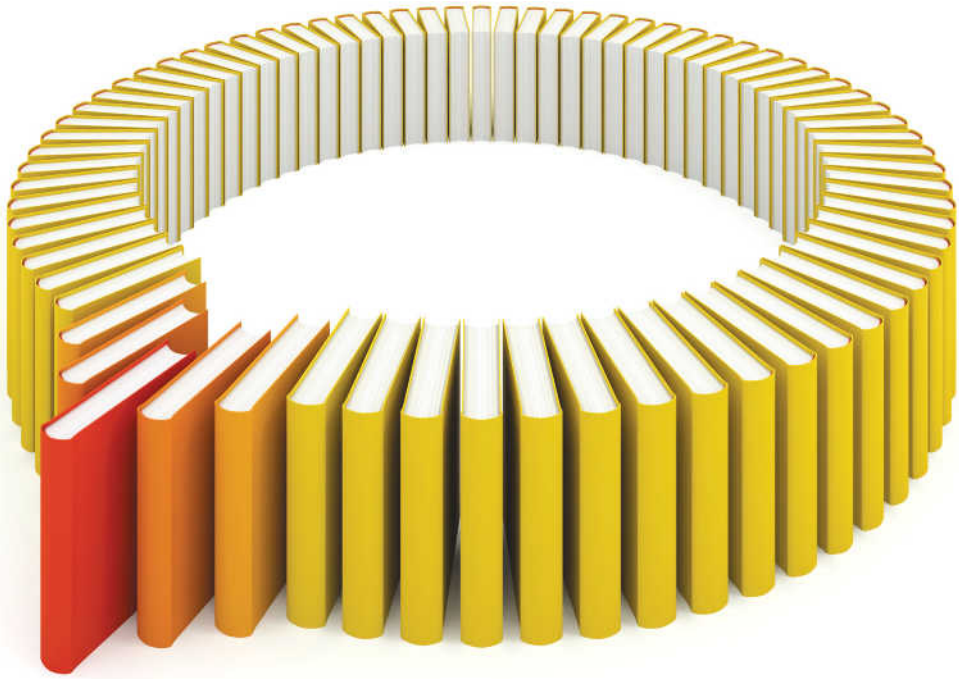
WHERE clause
a syntax string that is composed of the keyword WHERE, followed by one or more WHERE expressions. A WHERE clause defines the conditions to be used for selecting observations in a data set. *See also* WHERE expression.

WHERE clause planner
uses factors of cardinality and distribution to calculate relative processor costs of various WHERE clause options. The SPD Server WHERE clause planner avoids computation-intensive operations and uses simple computations where possible.

WHERE expression
is a syntax string within a WHERE clause that defines the criteria for selecting observations. For example, in a membership database, the expression "WHERE member_type=Senior" returns all senior members. *See also* compound WHERE expression, WHERE processing.

WHERE processing
a method of conditionally selecting rows for processing by using a WHERE expression. *See also* WHERE expression.

workspace tables
a list of paths that contain temporary SPD Server work tables and temporary intermediate files that are associated with the declared domain.



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.