SAS® Drivers for Federation Server 4.2: User's Guide

SAS® Documentation
March 29, 2018

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® Drivers for Federation Server 4.2: User's Guide*. Cary, NC: SAS Institute Inc.

SAS® Drivers for Federation Server 4.2: User's Guide

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

March 2018

P1:fedsrvdrvug

Contents

What's New in SAS® Drivers for Federation Server 4.2

Overview

SAS Drivers for Federation Server contains the following enhancements.

Summary of New Features

The following new features are available with this version of the Drivers for SAS Federation Server:

- support for one-time passwords
- driver support for the java.util.Date and java.util.Calendar data types
- new methods supported for Blob data type

One-Time Passwords

The JDBC Driver now supports one-time passwords (OTP). See "API Limitations" for information about using one-time passwords with the **`getSchemas`** method.

New Data Types

The JDBC Driver now supports the java.util.Date and java.util.Calendar data types. See "JDBC Data Type Conversions" for additional information.

Support for Blob Data Type

The following methods now provide support for Blob data type:

`CallableStatement interface`

```
  getBlob(int parameterIndex)
  getBlob(String parameterName)
  setBlob(String parameterName, Blob x)
  setBlob(String parameterName, InputStream inputStream)
  setBlob(String parameterName, InputStream  inputStream, long length)

Connection interface
  createBlob()

PreparedStatement interface
   setBlob(int parameterIndex, Blob blob)
   setBlob(int parameterIndex, InputStream inputStream)
    setBlob(int parameterIndex, InputStream inputStream, long length)

ResultSet interface
  getBlob(int index)
  getBlob(String colName)
  updateBlob(int columnIndex, Blob value)
  updateBlob(int columnIndex, InputStream inputStream)
  updateBlob(int columnIndex, InputStream inputStream, long length)
  updateBlob(String columnName, Blob value)
  updateBlob(String columnName, InputStream inputStream)
  updateBlob(String columnName, InputStream inputStream, long length)

RowSet interface
  setBlob(int index, Blob value)
  setBlob(int parameterIndex, InputStream inputStream)
  setBlob(int parameterIndex, InputStream inputStream, long length)
  setBlob(String parameterName, Blob blob)
  setBlob(String parameterName, InputStream inputStream)
  setBlob(String parameterName, InputStream inputStream, long length)
```

Chapter 1

Overview

About the Drivers for SAS Federation Server

The JDBC and ODBC drivers provide standard interfaces that enable applications to interact with SAS Federation Server. Using the drivers, applications can input and output data by performing the conversions that are necessary to transport data to and from SAS Federation Server.

The drivers are implemented as application programming interfaces (APIs). Here are the basic steps for using a driver:

1. Define a data source on SAS Federation Server.

   Note: See the *SAS Federation Server: Administrator's Guide* for information about defining data sources.

2. Create a data source name (DSN) on your client host to connect to the data source on SAS Federation Server.
3. Configure your client application to use an ODBC or JDBC driver to access the data source.

About Security

The SAS Drivers for Federation Server are installed by default with the product DataFlux Secure. The security product enables you to replace the default encryption algorithm, SAS Proprietary Encryption (SASPROPRIETARY) with 256-bit AES encryption.

Encryption is used for all network traffic to and from SAS Federation Server. You can also encrypt the passwords that are included in your data source names (DSNs) to secure them for storage on disk. The level of encryption that you use for your passwords must match the level of encryption that is configured on your SAS Federation Server.

To learn more about security and encryption, see "Why Encrypt?".

Chapter 2

Getting Started with the Drivers for SAS Federation Server

Configure the ODBC Driver for SAS Federation Server

Windows

In the Windows operating environment, no further configuration is needed after you install the ODBC Driver for SAS Federation Server.

UNIX or Linux

The distributions for Linux include pre-installed versions of unixODBC that are located in the **/usr/lib** or **/usr/lib64** directories. If the distribution contains release 2.3.1 or 2.3.2 of unixODBC, you do not need to build the ODBC libraries. After installation of the SAS Drivers for Federation Server on UNIX or Linux, follow these steps:

1. Download the source code for the unixODBC driver to SAS Federation Server. The source code is located at http://www.unixodbc.org/download.
2. Before you create the distribution, read the installation steps located in the directory **/opt/unixODBC-*release*/INSTALL**. Also read the installation steps that apply to your operating environment, such as **/opt/unixODBC-*release*/README.SOLARIS**.
3. Set environment variables for your operating system using one of the following procedures.

   AIX
   Set environment variables as follows. You should set the environment variables before creating libraries and programs.

   ```
   export OBJECT_MODE=64
   export CFLAGS="-q64 -DBUILD_REAL_64_BIT_MODE
   ```

```
                DSIZEOF_LONG=8"
export CC=xlc_r
export CPPFLAGS=$CFLAGS
```

Change to the output directory and issue the following commands:

```
cd '/opt/unixODBC-(release/aix/lib'
ar -X64 -x -v libodbc.a
ar -X64 -x -v libodbccr.a
ar -X64 -x -v libodbcinst.a
ln -s ./libodbc.so.1     ./libodbc.so
ln -s ./libodbccr.so.1   ./libodbccr.so
ln -s ./libodbcinst.so.1 ./libodbcinst.so
```

HP-UX H6I Itanium

Set environment variables as follows:

```
export CFLAGS="+DD64 -DBUILD_REAL_64_BIT_MODE
                -DSIZEOF_LONG=8"
export CPPFLAGS=$CFLAGS
```

Solaris

Set environment variables as follows:

```
export CFLAGS="-m64 -DBUILD_REAL_64_BIT_MODE
               -DSIZEOF_LONG=8"
export CC=cc
export CPPFLAGS=$CFLAGS
```

Linux

Set environment variables as follows:

```
export CFLAGS="-m64 -DBUILD_REAL_64_BIT_MODE
               -DSIZEOF_LONG=8"
export CPPFLAGS=$CFLAGS
```

4. Set the link order so that you link to the POSIX thread library before you link to the standard C library. Linking in this order ensures that you load all of the libraries that are required by the ODBC Driver for SAS Federation Server. The following example command shows the required loading order:

```
/usr/ccs/bin/ld -o UserApplication -u___exit
      -umain logger.o ODBCClient.o -L /usr/local/lib
      -L /usr/lib -L /opt/unixODBC-2.3.0/h6i/lib -lodbc
      -lpthread -lc
```

Configure the JDBC Driver for SAS Federation Server

Recommendations

The JDBC Driver for SAS Federation Server implements interfaces that conform to the standard JDBC API (Application Programming Interface) from Oracle. The driver is used to connect to various data sources configured on SAS Federation Server.

- JDK version: 1.6

- JRE version 1.6 or later. It is highly recommended that you use the SAS Private JRE that is shipped with the product and available through the SAS Deployment Wizard installation.

Note: See the Oracle Java Database Connectivity (JDBC) API documentation for information about JDBC specification standards.

Add JAR Files

After you install the JDBC Driver for SAS Federation Server, add the driver's JAR files to your application's CLASSPATH environment variable. By default, the JAR files are located in the **lib** directory of the installer: ***SASHOME*\SASFederationServerClient*release*\lib**. Here is the standard list of required JAR files:

```
icu4j-<version>.jar
log4j-<version>.jar
sas.core.jar
sas.core.nls.jar
sas.nls.collator.jar
sas.oda.tkts.jar
sas.oda.tkts.nls.jar
sas.security.sspi.jar
sas.svc.connection.jar
sas.svc.connection.nls.jar
```

Note: For JAR files that require a version, update the CLASSPATH to the installation location of the JAR files with the explicit name, for example,

```
classpath=C:\Program Files\SASHome\SASFederationServerClient\4.1\lib\icu4j-4.8.jar;
C:\Program Files\SASHome\SASFederationServerClient\4.1\lib\log4j-1.2.jar;...
```

The extra package of JAR files, installed with DataFlux Secure, are required for encryption. See "Encrypt Passwords in DSNs" on page 10 for additional information.

```
sas.rutil.jar
sas.rutil.nls.jar
sastpj.rutil.jar
```

Chapter 3

Using the ODBC Driver for SAS Federation Server

Create ODBC DSNs on Windows

Prerequisites

Before you create the ODBC data source names in the Windows operating environment, complete the following tasks:

- Install the SAS Driver for ODBC on your Windows application host.
- Configure data sources on the SAS Federation Server using administration DDL or SAS Federation Server Manager. For more information, see the *SAS Federation Server: Administrator's Guide*.

Create ODBC DSNs

Follow these steps to create ODBC DSNs on Windows:

1. In Windows, start the ODBC Data Source Administrator, as described in Open the ODBC Data Source Administrator.
2. Select the **User DSN** or **System DSN** tab.
3. Click **Add**.
4. In the Create New Data Source dialog box, select the driver **SAS 32-bit Federation Server** or **SAS 64-bit Federation Server**.
5. Click **Finish**.
6. In the SAS Federation Server ODBC Driver Setup dialog box, enter the required options and values. To learn about required options, see "Specify Required Options for ODBC Data Sources".
7. Add or change the values of the Advanced Options section as required by your application. To learn about advanced options, see "Specify Advanced Options for ODBC Data Sources".
8. Select **Test Connection** to confirm the validity of your driver setup values.
9. Click **OK** to complete the setup process.

Note: Installation of ODBC 4.1 does not update existing DSNs to point to the current release. Therefore, delete existing DSNs and re-create them using ODBC release 4.1.

Example Registry Entry

The following example depicts a registry entry for **FedServerDSN1**, which connects to the Federation Server with an Oracle base data source named ORACLE_DSN.

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\ODBC\ODBC.INI\FedServerDSN1]
"Driver"="C:\Program Files (x86)\SASHOME\bin\dfodbc.dll"
"Description"="" "Server"="myserver.us.orion.com" "Port"="21032" "Protocol"="BRIDGE"
"UID"="SAS\myuser" "FSDSN"="ORACLE_DSN" "Cei"="Win cp1252-latin1"
"PWD"="{SAS002}9C943B705636DF691286BEA34C6547D1" "CONCURR_RO"="false"
"CUR_FO"="false" "PSB"="prepare" "XCODE"="ignore" "FSCONSTR"=""
"TRACEFILE"="client.log" "TRACEFLAGS"="" "TRACELEVEL"="off"
"PARMS_FILE"="dfodbcJournal.log" "PARMS_TRACE"="off"
```

Create ODBC DSNs on UNIX or Linux

Overview

To enable an application to connect to ODBC data sources on SAS Federation Servers, set environment variables on the application host and define your ODBC data sources in a file.

Set Environment Variables

Set the following environment variables in the shell of your application:

LD_LIBRARY_PATH
: Set or append the location of the unixODBC binaries and also set or append the library for the ODBC driver, in ***install-path*/lib**. The following example command appends the two paths, using typical values:

```
export LD_LIBRARY_PATH=/wire/develop/odbc/2.3.1/lax/lib:/opt/SASHOME/fedclient/lib
```

If the platform provides the unixODBC libraries, set the following environment variable:

```
export LD_LIBRARY_PATH=/usr/lib64:/opt/SASHOME/fedclient/lib
```

ODBCINI
: Set the location of the .ini file that defines the ODBC data sources on your Federation Servers (for example, **/opt/SASHOME/fedclient/lib/dfodbc.ini**).

SAS SECURE
: Set the library path for SAS Secure:

```
export TKERSA2_LIB_PATH=/opt/SASHOME/fedclient/lib
```

TKPATH
: Set or append the location of the libraries that are used by the ODBC Driver for SAS Federation Server. Set or append a value such as the following: **/opt/SASHOME/fedclient/lib/**.

The library path specified for TKPATH reflects the fact that the ODBC Driver for SAS Federation Server can be used only with that version of the SAS Threaded Kernel library. Applications that use other SAS threaded kernel libraries cannot use the ODBC Driver for SAS Federation Server.

The following example uses the script dfsenv to illustrate how to update the TKPATH and LD_LIBRARY_PATH environment variables. If your application runs in a sh, ksh, or bash shell, then execute the following command to update the environment variables:

```
eval './bin/dfsenv sh'
```

If your application runs in a csh or tcsh shell, then execute the following command:

```
eval './bin/dfsenv csh'
```

Define Data Sources

After installing the ODBC Driver for SAS Federation Server on a UNIX or Linux client host, create a system configuration file to provide information for your ODBC data sources.

1. Navigate to the **/lib** directory, usually located in **SASHome/FederationServerClient/**, and create a text file to serve as the .ini file for your ODBC data sources.

   Note: The name and location of the file must match what is specified by the ODBCINI environment variable.

2. Edit the file to create a section with the name of [ODBC Data Sources].

3. Add an entry for each ODBC data source in the [ODBC Data Sources] section.
4. Create a section with the name of each entry that you created in step 2. For example, [FS_ORACLE] in the following sample specifies ODBC for Oracle. The file must contain a section for each data source that you are using.
5. Save the file, with a .ini extension, in your **/lib** directory. For example, **SASHome/FederationServerClient/lib/*dfodbc.ini***

Note: To add a comment line, begin the line with a **#** character.

Here is an example of an ODBC.INI file:

```
[ODBC Data Sources]
FS_ORACLE=Oracle ODBC

[FS_ORACLE]
Description = SAS Federation Server DSN
FSDSN = ORACLE_DSN
Driver = /opt/SASHOME/lib/dfodbc.so
SERVER = testFed01.us.orion.com
PORT = 21032
PROTOCOL = BRIDGE
UID = appConnect
PWD = 25o7xWd0gJzK
CEI = UTF-8
CONCURR_RO = false
CUR_FO = false
PSB = prepare
XCODE = ignore
```

Encrypt Passwords in DSNs

Why Encrypt?

You encrypt the passwords in your DSNs to protect those values when they are stored on disk. Network connections between your client application and the SAS Federation Server are encrypted by default.

Encryption Tools

To encrypt passwords, use the encryption tool **dfs_crypt** (in Windows) or **dfsadmin crypt** (in UNIX or Linux). For information about using the encryption tools, refer to "Utilities for SAS Federation Server" in the *SAS Federation Server: Administrator's Guide*.

Encryption Level

The level of encryption that you use for your passwords must match the level of encryption that is used by the SAS Federation Server. The default encryption level is SAS Proprietary Encryption (SASPROPRIETARY). If you upgrade the level of encryption on the SAS Federation Server, then you need to update your existing DSNs. The passwords need to be encrypted again using the encryption algorithm that is used by your SAS Federation Server.

Entering Passwords into Your Client Application

If your client application requires users to enter passwords to access data sources, then it might be necessary for those users to enclose their password entries in single quotation marks. The following example shows how single quotation marks are required by the Microsoft ODBC Test Tool. In the **`SQLDriverConnect`** statement, the password value for **`inConnectionString`** requires single quotation marks:

```
DSN=ORACLE_GENERIC;UID=Local/dsnadm;
     PWD='{SAS003}1804E6B4DEC1DD52DD9552D39BC82C96AD6E';
```

If quotation marks are expected but not provided, you will receive a parsing error message that references the encrypted password value.

Note: Quotation marks are not used with encrypted password values in DSNs. Quotation marks are not used in the Windows registry, the Windows ODBC Administrator, or in the UNIX or Linux file that defines DSNs (typically dfodbc.ini).

Specify Required Options for ODBC Data Sources

The following table defines the required values for ODBC data sources. The fields and values apply in the Windows, UNIX, and Linux operating environments.

Table 3.1 *Required Options for Data Sources*

Field	Description	Example
Data Source Name	The ODBC data source name refers to the collection of information that is used to access a SAS Federation Server data source. The name will likely provide information about the data source and possible options. The maximum length of names is specified in sqlext.h by SQL_MAX_DSN_LENGTH. The default length is 32 characters. Names are checked by the SQLValidDSN function to ensure that the names do not include the following characters: [ ] { } ( ) , ; ? ! @ \	FedSrvDB2
Description	Provides additional information about the data source.	Federation Server DB2
Server Name	The network name of the target host of the SAS Federation Server.	dev003.orion.com
Port Number	The port number that is assigned to the SAS Federation Server.	24141
Encoding or CEI	The character encoding that is used by SAS Federation Server.	Win cp1252-latin1 or UTF-8
User ID	The user name that your application uses to connect to the SAS Federation Server. Specify a domain if necessary. The user name appears in plaintext.	HQNT\JSmith

Field	Description	Example
Password	The password that is associated with the user ID. On Windows, the password is encrypted and stored in the Windows registry. On UNIX or Linux, the password is stored in plaintext or encrypted. To learn how to encrypt passwords, see "Encrypt Passwords in DSNs".	Plaintext: **`72Hken11`** Encrypted: **`{SAS003}1804E6B4DEC1DD52DD9552D39BC82C96AD6E`**
FS DSN	The SAS Federation Server data source name.	DB2_DSN
Locale	The locale used for the session. The default locale is English, United States.	en_US-English_United States

Specify Advanced Options for ODBC Data Sources

Introduction

You can specify advanced options in your ODBC DSNs to configure the SQL cursor, parameter set behavior, and transcode errors.

Advanced Options

The following table defines advanced options for ODBC data sources accessed through SAS Federation Server. These values apply in the Windows, UNIX, and Linux operating environments.

An example of an application that requires advanced options is the OpenOffice application, which requires that you enable the options Cursor Concurrency and Cursor Library. Enabling these options prevents SQLFetchScroll from reporting SQL_ERROR with an SQL State of HY106, which indicates that a fetch type is out of range.

Note: In the following table, the keyword value is the name of the option as it appears in the data source definition.

Table 3.2 *Advanced Options for ODBC Data Sources*

Option	Description
Cursor Concurrency	Values: false or true Default: false Keyword: **`CONCURR_RO`** Description: This option specifies the allowed values for the **`SQL_ATTR_CONCURRENCY`** attribute. This attribute is set using the **`SQLSetStmtAttr`** ODBC API. When this option is disabled, the ODBC driver places no limits on the values for the **`SQL_ATTR_CONCURRENCY`** attribute. When this option is enabled, the driver limits **`SQL_ATTR_CONCURRENCY`** to only **`SQL_CONCUR_READ_ONLY`**. Attempts to set **`SQL_ATTR_CONCURRENCY`** to a value other than **`SQL_CONCUR_READ_ONLY`** , such as **`SQL_CONCUR_LOCK`**, result in an SQL state of HYC00.

Option	Description
Cursor Library	Values: false or true Default: false Keyword: **CUR_FO** Description: This option specifies the allowed values for the **SQL_ATTR_CONCURRENCY** attribute. This attribute is set using the **SQLSetStmtAttrr** ODBC API. When this option is disabled, the ODBC driver places no limits on values for the **SQL_ATTR_ODBC_CURSORS** attribute. When this option is enabled, the driver explicitly sets **SQL_ATTR_ODBC_CURSORS** to **SQL_CUR_USE_IF_NEEDED** just prior to connection.
Parameter Set Behavior	Values: emulate, direct, or prepare Default: prepare Keyword: **PSB** Description: This option specifies the manner in which FedSQL emulates parameterized INSERT, DELETE, and UPDATE statements. emulate indicates that FedSQL always emulates parameterized INSERT, DELETE, and UPDATE statements using bulk operations (BulkOps). Parameter arrays should always work, provided that the underlying driver supports BulkOps. However, because the user has the option of setting the **PARAMSET_SIZE** after calling Prepare, some parameterized statements that could have been pushed down are emulated instead. direct indicates that FedSQL attempts to support the emulation of parameterized INSERT, DELETE, and UPDATE statements. When parameterized statements are attempted, if the underlying driver cannot support those statements, then those statements fail. prepare indicates that FedSQL limits **PARAMSET_SIZE** prior to Prepare. Attempts to set **PARAMSET_SIZE** generate an error. FedSQL determines at prepare-time whether to off-load the parameterized INSERT, DELETE, or UPDATE statement to the underlying driver or to attempt to emulate that statement via BulkOps.
Transcode Errors Behavior	Values: error, warning, or ignore Default: ignore Keyword: **XCODE** Description: This option determines how to report errors to applications when the ODBC Driver for SAS Federation Server is unable to transform character data (Transcode) to or from the data source. These errors typically occur when a character cannot be represented in both the application and the data source. The values specified by this option are used to set the **TKTS_ATTR_XCODE_WARN** attribute using the **TKTSSetStmtAttr** API provided by SAS Federation Server. error reports transcode errors at the error level. warning reports transcode errors at the warning level. ignore does not report transcode errors.

Usage Notes for the ODBC Driver

Changing the Default Current Catalog

The ODBC driver returns all of the schemas and tables that are associated with the current catalog. To access tables in a catalog other than the current catalog, you must specify a different catalog using one of the following methods:

- Specify a current catalog name in the connection string using the CATALOG= connection option, for example, **CATALOG=*catalog-name***.
- Set the **SQL_ATTR_CURRENT_CATALOG** connection attribute using the **SQLSetConnectAttr** function of the ODBC API. This attribute can be set before or after connection.

Zero-Length Strings Treated as NULL

Some data sources convert zero-length strings into NULL values. As a result, when an application reads these columns, the data returned is an output indicator of negative one (–1) instead of the expected length of the column. Zero-length string conditions should be checked in data sources where NULL and zero-length are not the same.

Using Microsoft Master Data Services

When you use Microsoft Excel to access data sources on a SAS Federation Server, be sure to configure your data service as case-insensitive. If your data service is case-sensitive, you might receive an error message that states that you submitted an invalid schema name. The requirement for case-insensitivity applies whenever you use Microsoft Master Data Services to access data sources on a SAS Federation Server.

Chapter 4

Using the JDBC Driver for SAS Federation Server

Importing the SQL Package and Loading the Driver

Import the SQL Package

The Java class, **`com.sas.tkts.TKTSDriver`**, provides implementation of **`java.sql.Driver`** in SAS Federation Server. Use this class name when registering the driver or when configuring software to use the JDBC Driver for SAS Federation Server. Before creating a connection, you must import the Java SQL packages:

```
import java.sql.*;
```

You might also want to import the **Properties** class:

```
import java.util.Properties;
```

Note: You can use an instance of the Properties class to pass connection properties when initiating a connection, rather than specifying individual options within the connection string.

Load the JDBC Driver

Use the **Class.forName()** method to load the JDBC Driver for SAS Federation Server:

```
Class.forName("com.sas.tkts.TKTSDriver");
```

Opening a Connection and Specifying Connection Options

Creating a Connection

There are two ways that you can specify connection properties when creating a connection for JDBC – In the connection string using a URL, or using the properties object, **java.util.Properties**, that you pass to the **getConnection()** call. JDBC applications require a valid server and port number that can be specified in a URL or as a property. When values are specified in both locations, the value specified as a property has precedence.

Specify Connection Options in a URL

Here is the syntax to create a connection and specify connection options using a URL:

```
jdbc:sastkts://fed-server-host:fed-server-port
      ?property-name1=property-value1
      &property-name2=property-value2...
      &property-nameX=property-valueX
```

The following example demonstrates the syntax:

```
jdbc:sastkts://devtest001.orion.com:2171?constring=(DSN=DB2DSN1)
      &userName=local\\user1&password=user1password
```

To connect to a SAS Federation Server that runs on the same host as your client, use the value **localhost** or **127.0.0.1** for ***fed-server-host***.

In a long JDBC URL, it is easy to confuse the JDBC connection properties and the SAS Federation Server driver's connection options. For example, consider this URL:

```
jdbc:sastkts://localhost:2171?cursorType=TKTS_CUR_USE_DRIVER
      &constring=(DRIVER=FIREBIRD;CATALOG=fbdetail;
      DATABASE='c:\fbdata\fbtest.fdb';)
```

In this case, **cursorType** and **constring** are JDBC connection properties. They are separated by an **&** and preceded by a **?**.

Using a Properties Object

To set connection properties in the **java.util.Properties** object, add key-value pairs to the object, and pass the object to **DriverManager.getConnection()** or **Driver.connect()**, or use the **set*()** methods on a data source. This method requires a database URL that includes a server name and port.

```
Properties props = new Properties();

        props.put("user", "fedserveruser");
        props.put("password", "{SAS003}5F5F60A0CEC0B921B503926075B4B6EBD08F");
        props.put("constring", "DSN=MYTRANDSN");

        Connection connection = DriverManager.getConnection(
                    "jdbc:sastkts://mymachine.orion.com:2171", props);
```

Specifying Connection Properties

The following table shows the connection options that are used with the JDBC Driver for SAS Federation Server. See Connection Properties for Statement Pool Manager to specify additional connection properties.

Table 4.1 *Connection Properties*

Property	Description
APPLICATION_NAME	**APPLICATION_NAME=*name_option*** Specifies the application name to include in SQL Logging for SAS Federation Server. This corresponds to an entry of **X{Client.AppName}** in the SQL Logging configuration file. The default value is **FEDSRV_JDBC_V4R2** specifying the name, version, and release values for the JDBC driver for SAS Federation Server. This connection keyword is optional. *Note:* When using the SAS Federation Server Data REST API client, the default APPLICATION_NAME is set to **SAS Federation Server Data REST API V4R2**.
compress	**compress=best|fast|none** Specifies the type of compression for the connection. The default is none (no compression). Use this option only when connecting to a remote server with limited bandwidth. It is suggested that performance be compared with and without compress enabled.
constring	Specifies a connection string for a data source.
cursorType	Specifies the type of cursor to use when traversing rowsets. The valid options are: **TKTS_CUR_USE_TKTS, TKTS_CUR_USE_DRIVER**, or **TS_CUR_USE_IF_NEEDED**.

Property	Description
debugUnit	**debugUnit=s|ms|ns|min** Controls the measurement unit for statistics logging. The default value is seconds (s). s Default. Logs statistics records in seconds. ms Logs statistics records in milliseconds. ns Logs statistics records in nanoseconds. min Logs statistics records in minutes.
Flush	**Flush=Y|N** Specifies whether the logging appender is immediate flush or buffered IO. The default is Yes – immediate flush.
defaultFetchSize	Specifies the default fetch size for the statements created by the connection. The default value is 100.
logLevel	**logLevel=Off|Fatal|Error|Warn|Info|Debug|Trace|All** Enables logging for the JDBC Driver and specifies the level of information to capture. For an explanation of the various logging levels, see "Overview" on page 31.
NULL_BEHAVIOR	Specifies how to insert missing values. **NULL_BEHAVIOR** controls how the user passes and retrieves null or missing values with the double data type. This property defines the behavior for all of the statements beneath it. The values for **NULL_BEHAVIOR** are as follows: default specifies that the JDBC driver does not set the statement attribute to null behavior, and it defaults to what is specified on the server. ANSI sets the **"TKTS_NB_MISSING"** statement attribute to **"TKTS_NB_ANSI"** indicating that the client wants to deal with ANSI NULLs only when binding a double. missing sets the **"TKTS_NB_MISSING"** statement attribute to **"TKTS_NB_MISSING"** indicating that the client wants to deal with SAS missing values only when binding a double or when binding strings. *Note:* JDBC does not support the retrieval of missing values. See "Reading and Writing NULL and Missing Values" on page 26 for more information.
password	The password associated with the **user** property that authenticates the client application for access to SAS Federation Server. To encrypt the password, see "Encrypt Passwords in DSNs".

Property	Description
port	Required. Specifies the port number used to access the data source. The default port is 21032. *Note:* JDBC applications require a valid server and port number. The server and port number can be specified as a property or directly in a URL. When values are specified in both locations, the value specified as a property has precedence.
proxylist	Specifies a list of one or more proxy servers used to connect to a server. If specifying more than one proxy server, use a semicolon to separate the server names. There is no default. This property is optional.
Schema_Scope	Limits the scope of schemas returned for the **getSchema()** and **getSchemas(CatalogName, SchemaPattern)** methods. **Schema_Scope** has two values: **all** and **default**. If no value is specified, the default is **all**. all specifies that all catalogs are returned for the **getSchema()** and **getSchemas(CatalogName, SchemaPattern)** methods. default specifies that only schemas associated with the default catalog are returned for the **getSchema()** and **getSchemas(CatalogName, SchemaPattern)** methods.
server	Required. Specifies the name of the server that is used to host the data source. *Note:* JDBC applications require a valid server and port number. The server and port number can be specified as a property or directly in a URL.
user	Specifies the user name of the login that is used to authenticate the client application for access to the SAS Federation Server. The user property is an alternate to **userName**. When both of these properties are used, the user property has precedence over the userName property. You must specify a valid user name when establishing connection to a secure SAS Federation Server. Alias: **userName**

The following table shows the connection options that are used with the Statement Pool Manager.

Table 4.2 *Connection Properties — Statement Pool Manager*

Property	Description
IDLE_TIMEOUT	**-DIDLE_TIMEOUT=*time-in-seconds*** Specifies the interval, in seconds, after which an idle pooled statement handle is closed and discarded. The default value is 120.
IS_ALLOCATE_OVER_THE_LIMIT	**-DIS_ALLOCATE_OVER_THE_LIMIT=TRUE|FALSE** Flags the Statement Pool Manager to allocate new statements after the maximum number of statements in the pool has been reached. The default value is TRUE.

Property	Description
IS_POOLED_STATEMENT_ENABLED	**-DIS_POOLED_STATEMENT_ENABLED=TRUE|FALSE** Indicates whether the Statement Pool Manager is active for the current connection. The default value is FALSE.
IS_RECORD_STATISTICS	**-DIS_RECORD_STATISTICS=TRUE|FALSE** Indicates whether the Statement Pool Manager displays statistics when the connection is closed. The default value is FALSE.
MAX_STATEMENT_POOL_SIZE	**-DMAX_STATEMENT_POOL_SIZE=*number-of-statements*** Specifies the maximum number of statements that can be stored in the pool. There is no wait if the pool reaches its maximum size. A request immediately allocates a new statement handle if **IS_ALLOCATE_OVER_THE_LIMIT** is set to TRUE, or it returns an error. The pool never exceeds the **MAX_STATEMENT_POOL_SIZE**. The Statement Pool Manager disposes of a previously pooled statement handle and replaces it with a new one. The maintenance task is freed to close and discard idle statements. The default value is 100.
POOL_MAINTENANCE_INTERVAL	**-DPOOL_MAINTENANCE_INTERVAL=*time-in-seconds*** Specifies the interval, in seconds, between executions of the statement pool maintenance task. The default value is 2 (seconds).
STATISTICS_OUTPUT_DIRECTORY	**-DSTATISTICS_OUTPUT_DIRECTORY=*absolute-path*** Specifies a directory where statistics records are written. The default is **C:\Users*username*\AppData\Local\Temp**. The filename has the following format: **stmtPooling*YYMMDD_HHMMSSmmm_pool-Hash-Code.log***. This property is used with the Statement Pool Manager only.

Working with Sample Code

JDBC Connection Sample

You can use the sample code provided below to configure a JDBC connection for your SAS Federation Server.

```
package com.sas.fs.doc.example;

import java.sql.*;
import java.util.Properties;

public class DocTest {
       static {
              try {
                     Class.forName("com.sas.tkts.TKTSDriver");
              } catch (ClassNotFoundException e) {
                     e.printStackTrace();
              }
       }

  public static void main(String argv[])
```

```
    {
       try {

            Properties props = new Properties();

            props.put("user", "userid");
            props.put("password", "password");
            props.put("constring", "DSN=DSN-nameMYTRANDSN");

            Connection connection = DriverManager.getConnection(
                        "jdbc:sastkts://machine-name.domain:port", props);

            Statement statement = connection.createStatement();
            ResultSet result = statement.executeQuery("SELECT * FROM EMPLOYEE");
            ResultSetMetaData rsmd = result.getMetaData();
            int numberOfColumns = rsmd.getColumnCount();

            for (int i = 1; i <= numberOfColumns; i++) {
                  System.out.print(rsmd.getColumnLabel(i) + " ");
            }

            System.out.println("");
            while (result.next()) {
                  for (int i = 1; i <= numberOfColumns; i++) {
                        System.out.print(result.getString(i) + " ");
                  }

                 System.out.println("");
            }
            statement.close();
            connection.close();
       }

      catch (Exception e) {
            System.out.println("error " + e);
        }
      }
}
```

Modifying the Sample Code

Use the following procedure to update your installation with the sample code.

1. Save the sample code to a file called DocTest.java and place the file in an accessible directory on your drive, for example, **`c:\temp\Fed_JDBC_DSN`**.
2. Edit the file to remove the first line, which is the package declaration:

   ```
   package com.sas.fs.doc.example;
   ```

3. Edit the file to specify the properties for your environment, and specify a table name in the **`executeQuery`** statement:

   ```
   props.put("user", "<userid>");
   props.put("password", "<password>");
   props.put("constring", "DSN="<dsn name>");

   Connection connection = DriverManager.getConnection(
   "jdbc:sastkts://< federation-server-hostname:port>",props);
   ```

```
Statement statement=connection.createStatement();
ResultSet result=statement.executeQuery("SELECT*FROM <table name>");
```

4. Compile the file:

```
javac DocTest.java
```

5. Run the program, including all the JAR files in your classpath and the DocTest file that was compiled in the previous step:

```
C:\temp\Fed_JDBC_DSN>java -classpath c:\temp\Fed_JDBC_DSN;
classpath=C:\Program Files\SASHome\SASFederationServerClient\4.1\lib\icu4j-4.8.jar;
C:\Program Files\SASHome\SASFederationServerClient\4.1\lib\log4j-1.2.jar;
      . . .  DocTest
```

Note: Output is written to **`stdout`**.

Using the Statement Pool Manager

Introduction

The Statement Pool Manager of the JDBC Driver for SAS Federation Server manages a list of named handles that apply to a set of frequently used SQL statements.

To enable and configure the Statement Pool Manager, set JDBC connection properties, as shown in the table, Connection Properties for Statement Pool Manager.

You can set properties as Java Virtual Machine (JVM) arguments or as TKTS connection properties. Properties defined at the JVM level are the default for all TKTS connections. To override the JVM properties, applications must define the properties at connection time.

Functionality

Operational Sequence

When it is initialized, the Statement Pool Manager starts a pool maintenance task. At intervals specified by the property **`POOL_MAINTENANCE_INTERVAL`**, the maintenance task closes and discards idle statements. Statements are removed if they have been idle for the interval specified by the property **`IDLE_TIMEOUT`**. When your application requests a statement, the Statement Pool Manager responds as follows:

1. If the statement is in the pool, the manager removes it from the pool and makes it available to your application.
2. If the statement is not in the pool and the pool is not full, the manager provides your application with a new statement handle. When the newly created statement is closed by your application, it is put into the pool.
3. The manager throws an exception **`StatementPoolFullException`** to your application if all of the following conditions are true:
   - The statement is not in the pool.
   - The pool is already full.
   - The property **`IS_ALLOCATE_OVER_THE_LIMIT`** is set to FALSE.

4. The manager provides your application with a new statement handle if either of these sets of conditions are met:
   - The statement is not in the pool, and the statement pool is not full.
   - The statement is not in the pool, the statement pool is already full, and the **IS_ALLOCATE_OVER_THE_LIMIT** property is set to TRUE.

About Statement Attributes

To use the Statement Pool Manager effectively, it is important to understand which statement attributes are preserved. The **ResultSetType** and **ResultSetConcurrency** statement attributes are the only attributes guaranteed to be unaltered. All other statement attributes are reset when you return a statement to the pool. Before executing a pooled statement, each application must set all statement attributes as it would with a newly created statement.

About Statement Handles

Statement pooling improves performance by reducing the time it takes to allocate and prepare statements. When using statement pooling, it is important to use a PREPARE statement with parameters rather than explicit values. For example, an application needs to collect information about worldwide sales by region. The following SQL statement allows statement pooling to improve the performance when collecting this information.

```
select * from WORLD_SALES where region = ?
```

The following SQL statements could be used to collect the same information. However, this approach would consume more resources in the statement pool and be less likely to receive the same level of performance improvement as the prior statement.

```
select * from WORLD_SALES where region = 1
select * from WORLD_SALES where region = 7
select * from WORLD_SALES where region = 9
```

In general, it is better to use parameters rather than explicit values. For complex SQL statements, it might be necessary to use explicit values to provide information to the Data Source optimizer. The Data Source optimizer can then format the request to reduce the number of rows to read while processing the request. This is especially important for requests against tables with a large number of rows.

Creating a Statement Pool with Connection Properties

The following example shows how to create a statement pool and set properties:

```
import com.sas.tkts.sql.StatementPoolManager;
Properties connectionProperties = new Properties();
connectionProperties.put(StatementPoolManager.PROP_IS_POOLED_STATEMENT_ENABLED,
     "true");
connectionProperties.put(StatementPoolManager.PROP_MAX_STATEMENT_POOL_SIZE,
     "100");
connectionProperties.put(StatementPoolManager.PROP_IS_ALLOCATE_OVER_THE_LIMIT,
     "true");
connectionProperties.put(StatementPoolManager.PROP_POOL_MAINTENANCE_INTERVAL,
     "1");
connectionProperties.put(StatementPoolManager.PROP_IDLE_TIMEOUT,
     "60");
Connection conn = DriverManager.getConnection("jdbc:sastkts://hostName:2100,
      connectionProperties);
```

Each of these properties is explained in the table, Connection Properties for Statement Pool Manager .

Note: Calling StatementPoolManager with a parameter that is a closed connection will result in an SQLException: Unable to establish a connection. Connection has been closed.

Setting Properties with JVM Arguments

To set default properties for the Statement Pool Manager using JVM arguments, use the following syntax to add options to your Java command. For option definitions, see the table, Connection Properties for Statement Pool Manager.

```
-DIS_POOLED_STATEMENT_ENABLED=TRUE|FALSE
-DMAX_STATEMENT_POOL_SIZE=number-of-statements
-DIS_ALLOCATE_OVER_THE_LIMIT=TRUE|FALSE
-DPOOL_MAINTENANCE_INTERVAL=time-in-seconds
-DIDLE_TIMEOUT=time-in-seconds
-DIS_RECORD_STATISTICS=TRUE|FALSE
-DSTATISTICS_OUTPUT_DIRECTORY=absolute-path
```

Retrieving Information about Statement Pool Manager

Use the following static methods in the **StatementPoolManager** class to retrieve information about the statement pool. Each method receives a connection as its input parameter:

public static long getCurrentPoolSize(Connection conn)
returns the current number of statements in the pool.

public static int getIdleTimeout(Connection conn)
returns the interval, in seconds, after which an idle pooled statement handle is closed and discarded.

public static int getMaxStatementPoolSize(Connection conn)
returns the maximum number of statement handles that can be stored in the pool.

public static long getPoolFullHits(Connection conn)
returns the number of times the pool manager tried to create a new statement but the pool was full.

public static long getPoolHits(Connection conn)
returns the number of times a statement was delivered from the pool.

public static int getPoolMaintenanceInterval(Connection conn)
returns the interval, in seconds, between executions of the statement pool maintenance task.

public static long getPoolMisses(Connection conn)
returns the number of times the pool manager did not find an available statement handle in the pool.

public static long getPoolPurges(Connection conn)
returns the number of times the maintenance task purged a pooled statement.

public static boolean isAllocateOverTheLimit(Connection conn)
returns a flag indicating if the pool manager will allocate a new statement after the maximum number of statements in the pool has been reached.

public static boolean isPooledStatementEnabled(Connection conn)
returns a flag indicating if statement pooling is in use for the current connection.

public static boolean isPurgeOnWaitTimeout(Connection conn)
returns true when the pool maintenance task will purge older statements when a **waitTimeout** is detected.

public static int getPurgeThreshold(Connection conn)
returns a threshold that defines the requested number of statements allowed in the pool. The number of active statements might exceed this value. At the end of a maintenance interval any idle statements in excess of this threshold are closed and removed from pool.

public static boolean isCacheOverTheLimit(Connection conn)
returns true to allow new statements to be added to the pool that are in excess of the pool limit.

public static int getWaitTimeOut(Connection conn)
returns the number of seconds to wait for a statement to be returned to the pool before creating a new statement.

public static int getActiveTasks()
returns the number of pool managers using the monitor task.

Using a Proxy Server

Introduction

You can specify one or more proxy servers for a JDBC connection. Using the PROXYLIST connection option, the client can communicate with SAS Federation server through HTTP configurations.

Specifying a PROXYLIST

In the event that a JDBC application needs to interact with a server that has been secured and cannot be accessed directly, use the **PROXYLIST** option to specify a proxy for the JDBC connection. Specify the **PROXYLIST** directly on the URL or as a property. This example demonstrates how to specify a **PROXYLIST** in a URL:

```
String url = "jdbc:sastkts://hostname:21032?PROXYLIST=http://myProxy.xyz.com";
DriverManager.getConnection(url);
```

This example specifies a **PROXYLIST** property:

```
String url = "jdbc:sastkts"."//hostname:21032"'
props = new Properties();
props.put("PROXYLIST","http://myProxy.xyz.com");
DriverManager.getConnection(url, props);
```

The default port for the proxy connection is 80. If a port is not specified, the default port is used. To customize the proxy port, add the port number after the proxy hostname, preceded by a colon:

```
String url = "jdbc:sastkts://hostname:21032?PROXYLIST=http://myProxy.xyz.com:3061";
```

You can also specify the port number as a property of proxylist:

```
props.put("proxylist", "http://myProxy.xyz.com:3061");
```

Reading and Writing NULL and Missing Values

Overview

The JDBC Driver for SAS Federation Server enables clients to set the behavior for SQL NULL and SAS missing values with the **NULL_BEHAVIOR** property. **NULL_BEHAVIOR** controls how the user passes and retrieves null and missing values for double data types.

What Are NULL and Missing Values?

An **ANSI SQL NULL** value indicates that a particular value is not known. A SAS missing value indicates that no data is stored for the variable in the current observation. There are three types of SAS missing values: numeric, character, and special numeric. Although **ANSI SQL NULL** values are sometimes equivalent to SAS missing values, the two concepts sometimes differ. For example:

- Users can specify many types of SAS missing values for numeric data. **ANSI SQL NULL** represents nonexistent data in one way only.
- If a NULL value is written to a character type column, the driver interprets the value as a SAS missing value and returns **FALSE** for the **wasNull** method.

Note: The **wasNull** method returns true for any numeric type column that holds an SQL NULL or SAS missing value.

For more information about SAS missing values, see *SAS Language Reference: Concepts*.

Configuring the SAS Missing Environment

To insert missing values, the **NULL_BEHAVIOR** property must be set on the JDBC connection. The **NULL BEHAVIOR** connection property has to be set to **MISSING** before a missing value can be inserted. Here is an example:

```
Properties connProperties = new Properties();
connProperties.setProperty("NULL_BEHAVIOR", "missing");
connection = DriverManager.getConnection(connURL, connProperties);
```

See the Connection Properties table for an explanation of the values that are available for the **NULL_BEHAVIOR** property. The values are not case sensitive.

Writing Values

Writing SQL NULL Values

SAS uses libraries to organize collections of tables. In most cases, when a table name is used in an SQL string, the name must be prefixed by a library name. If the SAS Work library is used, the prefix is not needed. The examples below demonstrate how to read and write null and missing values, and assume that the Work library is in use. This example creates a sample table named books and inserts two **SQL NULL** values:

```
stmt.executeUpdate("CREATE TABLE books (c1 varchar(32), c2 double precision)");
stmt.executeUpdate("INSERT INTO books VALUES(null, null)");
```

This example uses a prepared statement to store **SQL NULL** values:

```
stmt.executeUpdate("CREATE TABLE books (c1 varchar(32), c2 double precision)");
PreparedStatement ps = connection.prepareStatement("INSERT INTO books VALUES(?,?)");
ps.setNull(1, Types.VARCHAR);
ps.setNull(2, Types.DOUBLE);
ps.executeUpdate();
```

Writing SAS Missing Values

If writing to character columns, writing null values is more portable than writing missing values. Writing missing values to numeric type columns in SAS data sets might be appropriate if the data sets are used for reporting and the type of missing value is important.

Note: You must set the **NULL_BEHAVIOR** connection property to **MISSING** before inserting a missing value.

The following example code stores missing values:

```
stmt.executeUpdate("CREATE TABLE books (c1 varchar(32), c2 double precision)");
stmt.executeUpdate("INSERT INTO books VALUES(' ', .a)"); /* NOTE: just .a */
```

The following example code stores missing values by using a prepared statement.

```
stmt.executeUpdate("CREATE TABLE books (c1 varchar(32), c2 double precision)");
PreparedStatement ps = connection.prepareStatement("INSERT INTO books VALUES(?, ?");
ps.setString(1, " ");
ps.setObject(2, ".a", Types.DOUBLE);
ps.executeUpdate();
```

Missing character values are represented by a single blank space. Numeric values can have special missing value representations. A special missing value enables you to represent different categories of missing data by using the letters A–Z, a–z, a period, or an underscore. It is important to note that attempts to set double columns to these special values without setting the **NULL_BEHAVIOR** connection property to **MISSING** produces a Java Language exception.

NULL Values Determined by Other Data Sources

Some data sources convert zero-length strings into NULL values. Subsequently, when an application reads these columns, the data return is an output indicator of **-1** instead of the expected length of the column.

Chapter 5

Data Type Conversions for JDBC

JDBC Data Type Conversions

Overview

To facilitate the conversion of Java programming language data types, the JDBC Driver for SAS Federation Server provides data type conversions as defined in the JDBC specification. Here is additional information that could be helpful when converting certain data types.

Conversions to Boolean Class

The following methods might require conversion of an Object into a Boolean value:

- **`CallableStatement.getBoolean`**
- **`ResultSet.getBoolean`**
- **`PreparedStatement.ResultSet.setObject`**

When converting to Boolean, the following values are accepted for **`getBoolean`** in the **`FSCallableStatement`** and **`FSResultSet`** methods.

```
BOOLEAN_TRUE_CONSTANTS
   Strings CASE_INSENSITIVE
      "T", "Y", "1", "TRUE"  => true
      "F", "N", "0", "FALSE" => false
```

When converting Number data types (Byte, Short, Integer, Long, Float, Double, and BigDecimal), any nonzero value returns **`true`**.

Conversions to Number Classes

The various set and get methods for a Numeric Object might require a conversion where the target class supports a different range of values. The **`PreparedStatement`**,

`CallableStatement`, and **`ResultSet`** classes perform conversions and might generate warnings or exception when out-of-range values are encountered.

Exact Number data types

Data Types: **`BigDecimal, BigInt, Integer, Short, Byte`**

For Read operations converting from a data type supporting a larger range to a data type with a smaller range, the out-of-range values are mapped to the closest negative or positive number supported by target data type.

Approxiamte Number data types

Data Types: **`Double, Float`**

Out of range values are mapped to positive and negative infinity for these data types.

Conversions of Date and Calendar

java.util.Date, java.util.Calendar

Conversions of the **`java.util.Date`** and **`java.util.Calendar`** objects are transformed to the target SQL type. When the target SQL class is **`String`**, these objects are first converted into **`java.sql.Timestamp`** and then converted to **`String`**. This behavior is to allow for conversion to other SQL types such as **`java.sql.Date`**, **`java.sql.Time`**, and **`java.sql.Timestamp`**.

Chapter 6

JDBC Logging

Diagnostics and Event Logging

Overview

The JDBC Driver for SAS Federation Server includes a logging feature that when enabled, records events as they occur while the JDBC driver is running. Logged events can include SQL exceptions or detailed JDBC events such as opening and closing JDBC methods. The supported logging levels are as follows:

- Off
- Fatal
- Error
- Warn
- Info
- Debug
- Trace
- All

The most informative level of logging is INFO. Trace is used for debugging purposes.

Logs are written to a temporary directory belonging to the user or application. On Windows systems the logs are usually located in a path similar to **`C:\Users\`*`user-name`*`\AppData\Local\Temp\`*`log-name`*`.log`**.

About the Levels of Logging

Here is an explanation of each level of logging for the JDBC driver.

Off
: indicates that logging is off. No events are logged. This is the default setting.

Fatal
: indicates that there are problems registering a driver, or there are errors that prevent the application from continuing.

Error
: logs all application exceptions or errors in the application, some of which might be recoverable. The name of the error log is fs_jdbc_error.log.

Warn
: logs all SQLWarnings and application warnings.

Info
: logs information messages about operations completed by the application that include statement execution, status, and connection properties.

Debug
: logs statistics for JDBC internal API calls. Debug is used primarily for troubleshooting performance problems. When debug is enabled, all other logging levels are disabled. Use the following argument, or connection property, to control statistics logging when logging is set at debug: **`DebugAll=[Yes|No]`**

  When DebugAll is set at **`Yes`**, statistics for each individual method call are recorded. When the option is set to **`No`**, overall statistics are written to the log. The name of the statistics log is dfs_jdbc_stats.log.

  You can also specify the measurement unit for statistics logging by using the **`debugUnit`** connection property. The available options are **`debugUnit=s|ms|ns|min`**:

  s logs statistics records in seconds. This is the default value.

  ms logs statistics records in milliseconds.

  ns logs statistics records in nanoseconds.

  min logs statistics records in minutes.

Trace
: enables logging for application debugging, typically capturing the operational flow through the application.

All
: enables all logging levels except for Debug.

The following table shows the loggers that are active at each logging level.

JDBC Logging Levels	TRACE Activated	DEBUG Activated	INFO Activated	WARN Activated	ERROR Activated	FATAL Activated
TRACE	Y	N	Y	Y	Y	Y
DEBUG	N	Y	N	N	N	N
INFO	N	N	Y	Y	Y	Y
WARN	N	N	N	Y	Y	Y
ERROR	N	N	N	N	Y	Y
FATAL	N	N	N	N	N	Y

JDBC Logging Levels	TRACE Activated	DEBUG Activated	INFO Activated	WARN Activated	ERROR Activated	FATAL Activated
ALL	Y	N	Y	Y	Y	Y
OFF	N	N	N	N	N	N

Enable JDBC Logging

Enable Logging Using JVM Arguments

To enable logging using a JVM argument, issue the following statement and specify a logging level at a start up command:

```
-DdfsJDBCLoggingEnabled=info[off|fatal|warn|trace|debug|all]
```

For example:

```
java -DdfsJDBCLoggingEnabled=”info” -jar myJDBCTest.jar
```

Enable Logging with Connection Properties

To enable logging using a connection property, you must first import **`java.util.Properties`**. Then, specify a logging level using this connection property:**`logLevel=Off`**|**`Fatal`**|**`Error`**|**`Warn`**|**`Info`**|**`Debug`**|**`Trace`**|**`All`**.

```
import java.util.Properties;

       Properties props = new Properties();

       props.put("user", "domain\\user-id");
       props.put("password", "user-password");
       props.put("constring", "DSN=BASE");
       props.put("logLevel", "info");

       try {
              connection = DriverManager.getConnection(
                           "jdbc:sastkts://d76784.na.sas.com:2171", props);
       } catch (SQLException e) {
              e.printStackTrace();
       }
```

Logger requests that are invoked using a VM argument always supersede logger requests that are defined with a connection property. For example, if a VM logging argument is set to INFO while a connection property is set to TRACE, the logger is set to INFO. When a VM argument is not set but a connection property is, the connection property defines the logging level. The logging configuration in place for the first established connection persists for all subsequent connections.

Logging File Appender

Use the following VM argument, or connection property, to control whether the logging file appender is immediate flush or buffered I/O: **`Flush=[Y`**|**`N Yes`**|**`No]`**. The default is yes – immediate flush.

Chapter 7

JDBC Class and Method Limitations

API Limitations: Server Constraints and Driver Restrictions

Limitations by Class

Connection

Connection.setCatalog
Changing the catalog might impact entities created under the connection. Entities like Statements, ResultSet, and their descendants might fail or return inconsistent information. For example, prepared statements or **`DataBaseMetadata.getSchemas`** resultSet could be affected by changing catalogs.

DatabaseMetaData

DatabaseMetaData.getAttributes
SAS Federation Server does not support user-defined types or the API to collect information about attributes. This method returns an empty result set.

DatabaseMetaData.getClientInfoProperties
SAS Federation Server does not support client information properties. This method returns an empty result set.

DatabaseMetaData.getColumns()
Columns 19 through 22 returns **`NULL`** and column 23 returns an empty string:

```
19 SCOPE_CATLOG String ==>NULL
20 SCOPE_SCHEMA String==>NULL
21 SCOPE_TABLE String==>NULL
22 SOURCE_DATA_TYPE short==>NULL
23 IS AUTO_INCREMENT==>(empty string)
```

DatabaseMetaData.getFunctions(), DatabaseMetaData.getFunctionColumns()
Returns an empty result set because SAS Federation Server does not support function columns.

DatabaseMetadata.getSchemas
When using One Time Passwords (OTP), the **`getSchemas`** method can change the connection catalog, which might impact entities created under the connection. Entities like Statements, ResultSet, and their descendants can fail or return inconsistent information.

DatabaseMetaData.getSuperTables
The Federation Server does not support super tables. This method returns an empty result set.

DatabaseMetaData.getSuperTypes
The Federation Server does not support user-defined types and associated hierarchies. This method returns an empty result set.

DatabaseMetaData.getUDTs
The Federation Server does not support user-defined types and associated hierarchies. This method returns an empty result set.

DatabaseMetaData.getUserName()
When the user identifier is not provided while establishing connection, the SAS Federation Server JDBC driver is unable to provide it for **`DatabaseMetaData getUserName`**.

Here are some examples where you would be able to provide a **`userid`**:

```
Properties conProps = new Properties();
  conProps.put("user", myUID);
  conProps.put("password", myPWD);
  conProps.put("constring", myConstring);
  String url = "jdbc:sascloud://testhost";
  Connection tConn = DriverManager.getConnection(url, conProps);

  String url = "jdbc:sascloud://testhost&constring=(DSN=MY_DSN)";
  Connection tConn = DriverManager.getConnection(url,  myUID, nyPWD);
```

The following examples do not provide a **`userid`**:

```
 String url =
"jdbc:sascloud://testhost&constring=(DSN=MY_DSN,PWD=XXX,user=userid)";
  Connection tConn = DriverManager.getConnection(url);

  Properties conProps = new Properties();
  conProps.put("constring", "(DSN=MY_DSN,PWD=XXX,user=userid)");
  String url = "jdbc:sascloud://testhost";
  Connection tConn = DriverManager.getConnection(url, conProps);
```

ResultSet, ResultSetMetaData

ResultSetMetaData getMetaData()
When calling **`getMetaData()`** on a **`ResultSet`** object, an SQLException is thrown in two instances:

- if a database access error occurs.
- if the method is called on a closed result set.

It should be noted that **`ResultSetMetaData`** instances created by a **`ResultSet`** that was previously closed are still accessible.

Statement Pooling

To prevent conflicts with statement pooling, the JDBC driver does not support the following methods for **`PreparedStatement`** and **`CallableStatement`** classes. The application should use multiple statements rather than updating the SQL text of a prepared statement.

PreparedStatement.*method_name*
CallableStatement.*method_name*

```
void addBatch(String sql)
ResultSet executeQuery(String query)
int executeUpdate(String query)
int executeUpdate(String query, int autoGeneratedKeys)
int executeUpdate(String query, int[] columnIndexes)
int executeUpdate(String query, String[] columnNames)
boolean execute(String query)
boolean execute(String query, int autoGeneratedKeys)
boolean execute(String query, int[] columnIndexes)
boolean execute(String query, String[] columnNames)
```

Chapter 8

JDBC Best Practices

JDBC Driver for SAS Federation Server

Closing Result Sets, Statements, and Connections

The **`ResultSet`**, **`Statement`**, and **`Connection`** objects can consume quite a bit of memory, so it is strongly recommended that you close these instances when you are finished using them. This practice should release database resources.

Setting JDBC Fetch Size

Setting a fetch size to a value smaller than the default of 100 might contribute to slow fetch performance. A lower fetch size value results in more server trips, which increase execution times. On the other hand, increasing fetch size could improve performance, but setting it too high could result in an out of memory issue. Here is an example setting for fetch size:

```
PreparedStatement stmt = connection.prepareStatement(query);
int fetchSize = 500;
stmt.setFetchSize(fetchSize);
```

Limiting the Number of Fetched Rows

When working with large amounts of data, it is important to limit the number of rows to fetch. To limit the number of rows retrieved from a database, use the statement interface, **`setMaxRows`**. Here is an example that illustrates this:

```
PreparedStatement stmt = connection.prepareStatement(query);
int maxRows = 100;
stmt.setMaxRows(maxRows);
```

Use JDBC Batch Update for Large Inserts

It is a best practice to insert and update large amounts of data in batches. When performing a large number of parameterized inserts, use JDBC Batch Update. When you submit multiple SQL statements in batch instead of individually, the number of round trips to the database is reduced, which results in significant performance improvement. Here is an example:

```
PreparedStatement pstmt = null;

pstmt = connection.prepareStatement("INSERT INTO \"USERS\"VALUES (?, ?, ?)");

pstmt.setString(1, "john_doe");
pstmt.setInt(2, 25);
pstmt.setString(3, "like to walk");
pstmt.addBatch();

pstmt.setString(1, "john_moe");
pstmt.setInt(2, 50);
pstmt.setString(3, "like to play");
pstmt.addBatch();

pstmt.setString(1, "john_coe");
pstmt.setInt(2, 60);
pstmt.setString(3, "like to eat");
pstmt.addBatch();

pstmt.executeBatch();
```

SAS Federation Server conforms to the following Oracle JDBC rule:

For any given statement, an application should not modify the value argument passed to a **`setXXX`** method after the **`setXXX`** method is called and before the subsequent **`execute`**, **`executeQuery`**, **`executeUpdate`**, **`excuteBatch`**, or **`clearParameters`** method is called. An application can modify the value argument after the **`execute`**, **`executeQuery`**, **`executeUpdate`**, **`excuteBatch`**, or **`clearParameters`** method is called, if there is a subsequent **`setXXX`** method call that overwrites the previous value or if the statement is not reused. Failure to conform to this restriction can result in unpredictable behavior.

Statement Pooling

Use the statement pooling feature when executing the same statement multiple times. Statement pooling improves performance by reducing processing for the **`allocate`** and **`prepare`** statements.

Invoking last and isLast Methods

Some databases and underlying drivers do not support a scrollable result set. In this case when requesting this type of a result set the following error message within an exception is returned: **`Unable to create Statement with requested result set concurrency[1008]`**. Here is an example of this request:

```
try {
stmt = currentConn.createStatement(
```

```
                ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
} catch (SQLException e1) {
       printSQLExceptions(e1, "HY000", UNSUPPORTED_SCROLL_MESSAGE);
}
```

If a scrollable result set is supported, then an application can call **`last()`** and **`isLast()`** methods. Calling **`last()`** moves the cursor to the last row in the result set and calling **`isLast()`** determines whether the cursor is on the last row. Calling **`isLast()`** can impact performance because the JDBC driver might need to fetch ahead one row in order to determine whether the current row is the last row in the result set. If high performance is essential, avoid using the **`isLast()`** method. Here is an example that uses the **`last()`** and **`isLast()`** methods:

```
boolean moved = false;
String selectFromTable = "SELECT * FROM \"EMPLOYEES\"";

try {
       stmt = currentConn.createStatement(
              ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
} catch (SQLException e1) {
       printSQLExceptions(e1);
}

try {
       rs = stmt.executeQuery(selectFromTable);
} catch (SQLException e) {
       printSQLExceptions(e);
}

// move cursor to the last row
try {
       moved = rs.last();
} catch (SQLException e) {
       printSQLExceptions(e);
}

// determine if this is a last row
       try {
              if (rs.isLast()){
                     System.out.println("This is a last row in a result set.");
              }
       } catch (SQLException e) {
              printSQLExceptions(e);
}
```

In a case when a scrollable result set is not supported calling **`last()`** on a **`ResultSet`** object produces the following error: **`Unable to perform requested operation on a forward only result set`**. Calling the **`isLast()`** method results in the error: **`Method not supported when resultSetType is TYPE_FORWARD_ONLY`**.