Progress DataDirect *for* JDBC for SQL Server

User's Guide

Release 6.0.0

Copyright

Updated: 2020/01/22

Table of Contents

1

Welcome to the Progress DataDirect for JDBC for SQL Server: Version 6.0.0

The Progress® DataDirect® *for* JDBC™ for SQL Server™ driver supports the JDBC API for SQL read-write access to Microsoft SQL Server and Microsoft Azure, including Microsoft Azure Synapse Analytics and Microsoft Analytics Platform System.

For details, see the following topics:

- What's new in this release?
- Data source and driver classes
- Connection URL
- Requirements
- Version string information
- Connection properties
- Data types
- Contacting Technical Support

What's new in this release?

Support and certification

Visit the following web pages for the latest support and certification information.

- Release Notes
- Supported Configurations
- DataDirect Support Matrices

Changes since the 6.0.0 release

- Driver enhancements
    - The driver has been enhanced to support encrypted parameters in stored procedures when using the Always Encrypted feature.
    - The driver has been enhanced to support the Always Encrypted feature. Beginning with SQL Server 2016, Azure SQL and SQL Server databases support Always Encrypted, which allows sensitive data to be stored on the server in an encrypted state such that the data can only be decrypted by an authorized application. The following are highlights of this enhancement:
        - The driver detects all supported native data types stored in encrypted columns and transparently encrypts values bound to SQL parameters or decrypts values returned in results.
        - The driver supports configurable caching of column encryption keys for improved performance.
        - The driver supports using Java KeyStore and Azure Key Vault as keystore providers.

      You can enable support for Always Encrypted using the following new options: ColumnEncryption, AEKeyCacheTTL, AEKeystoreClientSecret, AEKeystoreLocation, AEKeystorePrincipalId, and AEKeystoreSecret. See Always Encrypted on page 88 for details.

      Important: Always Encrypted support requires the driver to run on a Java Virtual Machine (JVM) that is Java SE 8 or higher.

Changes for the 6.0.0 release

- Driver enhancements
    - The driver has been enhanced to transparently connect to Microsoft Azure Synapse Analytics and Microsoft Analytics Platform System data sources. See Azure Synapse Analytics and Analytics Platform System on page 77 for more information about supported features and functionality.
    - The driver has been enhanced to support Always On Availability Groups. Introduced in SQL Server 2012, Always On Availability Groups is a replica-database environment that provides a high-level of data availability, protection, and recovery. See Always On Availability Groups on page 100 for details on using the driver with this feature.
    - The driver has been enhanced to support Azure Active Directory authentication (Azure AD authentication). Azure AD authentication is an alternative to SQL Server Authentication that allows administrators to centrally manage user permissions to Azure SQL Database data stores. See Configuring Azure Active Directory authentication on page 80 for details.

  - The driver has been enhanced to support Kerberos constrained delegation. Constrained delegation is a Kerberos mechanism that allows a client application to delegate authentication to a second service. See Configuring the driver for Kerberos authentication on page 81 and Constrained delegation on page 85 for details.

- Changed behavior
  - For Kerberos authentication environments, the following changes have been implemented.
    - The driver no longer sets the java.security.auth.login.config system property to force the use of the installed `JDBCDriverLogin.conf` file as the JAAS login configuration file. By default, the driver now uses the default JAAS login configuration file for Java, unless you specify a different location and file using the java.security.auth.login.config system property.
    - The driver no longer sets the java.security.krb5.conf system property to force the use of the `krb5.conf` file installed with the driver jar files in the `/lib` directory of the product installation directory.

    See Configuring the driver for Kerberos authentication on page 81 for details.

Data source and driver classes

The driver provides the following driver class.

```
com.ddtek.jdbc.sqlserver.SQLServerDriver
```

The driver provides the following data source class that supports the functionality for all JDBC specifications and Java SE 6 or higher.

```
com.ddtek.jdbcx.sqlserver.SQLServerDataSource
```

See also

Connecting using data sources on page 40

Connection URL

The connection URL format for the driver is:

```
jdbc:datadirect:sqlserver://hostname:port[;property=value[;...]]
```

where:

`hostname`

is the IP address or host name of the server to which you are connecting. See Using IP addresses on page 106 for details on using IP addresses.

`port`

is the number of the TCP/IP port.

property = *value*

specifies connection properties. For a list of connection properties and their valid values, see Connection property descriptions on page 185.

Notes

- Untrusted applets cannot open a socket to a machine other than the originating host.

Example

```
jdbc:datadirect:sqlserver://MyServer:1433;User=test;Password=secret;DatabaseName=MyDB
```

See Connecting to named instances on page 77 for instructions on connecting to named instances.

See also

Using connection properties on page 59

Requirements

The driver is compatible with JDBC 2.0, 3.0, 4.0, 4.1, and 4.2.

The driver requires a Java Virtual Machine (JVM) that is Java SE 6 or higher, including Oracle JDK, OpenJDK, and IBM SDK (Java) distributions.

Note: To use the driver on a Java Platform with standard Security Manager enabled, certain permissions must be set in the security policy file of the Java Platform. See Required permissions for Java SE with the standard Security Manager enabled on page 48 for details.

Version string information

The `DatabaseMetaData.getDriverVersion()` method returns a driver version string in the format:

```
M.m.s.bbbbbb(FYYYYYY.UZZZZZZ)
```

where:

M is the major version number.

m is the minor version number.

s is the service pack number.

bbbbbb is the driver build number.

YYYYYY is the framework build number.

ZZZZZZ is the utl build number.

For example:

```
6.0.0.000002(F000001.U000002)
      |____| |_____| |_____|
      Driver  Frame    Utl
```

Connection properties

The driver includes over 60 connection properties. You can use these connection properties to customize the driver for your environment. Connection properties can be used to accomplish different tasks, such as implementing driver functionality and optimizing performance. You can specify connection properties in a connection URL or within a JDBC data source object.

See also

Using connection properties on page 59

Connection property descriptions on page 185

Data types

The following table lists supported data types supported and how they are mapped to JDBC data types.

Table 1: Microsoft SQL Server Data Types

Microsoft SQL Server Data Type	JDBC Data Type
bigint	BIGINT
bigint identity	BIGINT
binary	BINARY
bit	BIT
char	CHAR
date	DATE
datetime	TIMESTAMP
datetime2	TIMESTAMP
datetimeoffset[1]	VARCHAR or TIMESTAMP
decimal	DECIMAL
decimal() identity[2]	DECIMAL
float	FLOAT
image[2]	LONGVARBINARY

[1] When FetchTSWTZasTimestamp=false (default), this data type is mapped to the JDBC VARCHAR data type; when FetchTSWTZasTimestamp=true, it is mapped to the JDBC TIMESTAMP data type.

[2] Not supported for Microsoft Azure Synapse Analytics and Microsoft Analytics Platform System.

Microsoft SQL Server Data Type	JDBC Data Type
int	INTEGER
int identity	INTEGER
money	DECIMAL
nchar	NCHAR
ntext[2]	LONGNVARCHAR
numeric	NUMERIC
numeric() identity[2]	NUMERIC
nvarchar	NVARCHAR
nvarchar(max)	LONGNVARCHAR
real	REAL
smalldatetime	TIMESTAMP
smallint	SMALLINT
smallint identity[2]	SMALLINT
smallmoney	DECIMAL
sql_variant[2]	VARCHAR
sysname	VARCHAR
text[2]	LONGVARCHAR
time[3]	TIME or TIMESTAMP
timestamp	BINARY
tinyint	TINYINT
tinyint identity[2]	TINYINT
uniqueidentifier	CHAR
varbinary	VARBINARY
varbinary(max)	LONGVARBINARY

[3] When FetchTWFSasTime=true, this data type is mapped to the JDBC TIME data type. When FetchTWFSasTime=false (the default), this data type is mapped to the JDBC TIMESTAMP data type.

Microsoft SQL Server Data Type	JDBC Data Type
varchar	VARCHAR
varchar(max)	LONGVARCHAR
xml[2,4]	SQLXML

getTypeInfo

The following table provides `getTypeInfo()` results for supported data types.

TYPE_NAME = bigint
AUTO_INCREMENT = false
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -5 (BIGINT)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = bigint
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = 10
	PRECISION = 19
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

TYPE_NAME = bigint identity
AUTO_INCREMENT = true
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -5 (BIGINT)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = bigint identity
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 0
	NUM_PREC_RADIX = 10
	PRECISION = 19
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

[4] The XMLDescribeType property overrides the mappings for XML data.

TYPE_NAME = binary
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = *length*
DATA_TYPE = -2 (BINARY)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = 0x
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = binary
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 8000
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = bit
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -7 (BIT)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = bit
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 1
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = char
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = *length*
DATA_TYPE = 1 (CHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = char
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 8000
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = date
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 91 (DATE)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = date
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 10
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = datetime
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 93 (TIMESTAMP)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = datetime
MAXIMUM_SCALE = 3
	MINIMUM_SCALE = 3
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 23
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = datetime2
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 93 (TIMESTAMP)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = datetime2
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 27
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = datetimeoffset
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 12 (VARCHAR) or 93 (TIMESTAMP)[5]
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = datetimeoffset
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 34
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = decimal
AUTO_INCREMENT = false
CASE_SENSITIVE = false
CREATE_PARAMS = *precision*,*scale*
DATA_TYPE = 3 (DECIMAL)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = decimal
MAXIMUM_SCALE = 38
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = 10
	PRECISION = 38
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

**TYPE_NAME = decimal() identity**[6]
AUTO_INCREMENT = true
CASE_SENSITIVE = false
CREATE_PARAMS = *precision*
DATA_TYPE = 3 (DECIMAL)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = decimal() identity
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 0
	NUM_PREC_RADIX = 10
	PRECISION = 38
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

[5] When FetchTSWTZasTimestamp=false, the data type that is returned by DATA_TYPE is VARCHAR; when FetchTSWTZasTimestamp=true, the data type that is returned is TIMESTAMP.

[6] Not supported for Microsoft Azure Synapse Analytics and Microsoft Analytics Platform System.

TYPE_NAME = float
AUTO_INCREMENT = false
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 6 (FLOAT)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = float
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = 2
	PRECISION = 53
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

**TYPE_NAME = image**[6]
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -4 (LONGVARBINARY)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = 0x
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = image
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 2147483647
	SEARCHABLE = 0
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = int
AUTO_INCREMENT = false
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 4 (INTEGER)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = int
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = 10
	PRECISION = 10
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

TYPE_NAME = int identity
AUTO_INCREMENT = true
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 4 (INTEGER)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = int identity
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 0
	NUM_PREC_RADIX = 10
	PRECISION = 10
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

TYPE_NAME = money
AUTO_INCREMENT = false
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 3 (DECIMAL)
FIXED_PREC_SCALE = true
LITERAL_PREFIX = $
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = money
MAXIMUM_SCALE = 4
	MINIMUM_SCALE = 4
	NULLABLE = 1
	NUM_PREC_RADIX = 10
	PRECISION = 19
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

TYPE_NAME = nchar
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = *length*
DATA_TYPE = -15 (NCHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = N'
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = nchar
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 4000
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

**TYPE_NAME = ntext**[6]
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -16 (LONGNVARCHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = N'
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = ntext
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 1073741823
	SEARCHABLE = 1
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = numeric
AUTO_INCREMENT = false
CASE_SENSITIVE = false
CREATE_PARAMS = *precision*,*scale*
DATA_TYPE = 2 (NUMERIC)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = numeric
MAXIMUM_SCALE = 38[7]
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = 10
	PRECISION = 38
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

**TYPE_NAME = numeric() identity**[6]
AUTO_INCREMENT = true
CASE_SENSITIVE = false
CREATE_PARAMS = *precision*
DATA_TYPE = 2 (NUMERIC)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = numeric() identity
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 0
	NUM_PREC_RADIX = 10
	PRECISION = 38
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

[7] This value can be configured with a server option in SQL Server and Azure.

TYPE_NAME = nvarchar
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = *max length*
DATA_TYPE = -9 (NVARCHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = N'
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = nvarchar
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 4000
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = nvarchar(max)
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -16 (LONGNVARCHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = N'
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = nvarchar(max)
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 1073741823
	SEARCHABLE = 1
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = real
AUTO_INCREMENT = false
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 7 (REAL)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = real
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = 2
	PRECISION = 24
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

TYPE_NAME = smalldatetime
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 93 (TIMESTAMP)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = smalldatetime
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 16
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = smallint
AUTO_INCREMENT = false
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 5 (SMALLINT)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = smallint
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = 10
	PRECISION = 5
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

**TYPE_NAME = smallint identity**[6]
AUTO_INCREMENT = true
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 5 (SMALLINT)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = smallint identity
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 0
	NUM_PREC_RADIX = 10
	PRECISION = 5
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

TYPE_NAME = smallmoney
AUTO_INCREMENT = false
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 3 (DECIMAL)
FIXED_PREC_SCALE = true
LITERAL_PREFIX = $
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = smallmoney
MAXIMUM_SCALE = 4
	MINIMUM_SCALE = 4
	NULLABLE = 1
	NUM_PREC_RADIX = 10
	PRECISION = 10
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = false

**TYPE_NAME = sql_variant**[6]
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 12 (VARCHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = sql_variant
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = 10
	PRECISION = 8000
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = sysname
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 12 (VARCHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = N'
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = sysname
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 0
	NUM_PREC_RADIX = NULL
	PRECISION = 128
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = text[6]
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -1 (LONGVARCHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = text
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 2147483647
	SEARCHABLE = 1
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = time
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 93 (TIMESTAMP)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = time
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 16
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = timestamp
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -2 (BINARY)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = 0x
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = timestamp
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 0
	NUM_PREC_RADIX = NULL
	PRECISION = 8
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = tinyint
AUTO_INCREMENT = false
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -6 (TINYINT)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = tinyint
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 1
	NUM_PREC_RADIX = 10
	PRECISION = 3
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = true

**TYPE_NAME = tinyint identity**[6]
AUTO_INCREMENT = true
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -6 (TINYINT)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = NULL
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = tinyint identity
MAXIMUM_SCALE = 0
	MINIMUM_SCALE = 0
	NULLABLE = 0
	NUM_PREC_RADIX = 10
	PRECISION = 3
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = true

TYPE_NAME = uniqueidentifier
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = 1(CHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = uniqueidentifier
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 36
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = varbinary
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = *max length*
DATA_TYPE = -3 (VARBINARY)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = 0x
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = varbinary
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 8000
	SEARCHABLE = 2
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = varbinary(max)
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -4 (LONGVARBINARY)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = 0x
LITERAL_SUFFIX = NULL
LOCAL_TYPE_NAME = varbinary(max)
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 2147483647
	SEARCHABLE = 0
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = varchar
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = *max length*
DATA_TYPE = 12 (VARCHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = varchar
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 8000
	SEARCHABLE = 3
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

TYPE_NAME = varchar(max)
AUTO_INCREMENT = NULL
CASE_SENSITIVE = false
CREATE_PARAMS = NULL
DATA_TYPE = -1 (LONGVARCHAR)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = '
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = varchar(max)
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 2147483647
	SEARCHABLE = 1
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

**TYPE_NAME = xml**[6]
AUTO_INCREMENT = NULL
CASE_SENSITIVE = true
CREATE_PARAMS = NULL
DATA_TYPE = 2009 (SQLXML)
FIXED_PREC_SCALE = false
LITERAL_PREFIX = N'
LITERAL_SUFFIX = '
LOCAL_TYPE_NAME = xml
MAXIMUM_SCALE = NULL
	MINIMUM_SCALE = NULL
	NULLABLE = 1
	NUM_PREC_RADIX = NULL
	PRECISION = 1073741823
	SEARCHABLE = 0
	SQL_DATA_TYPE = NULL
	SQL_DATETIME_SUB = NULL
	UNSIGNED_ATTRIBUTE = NULL

Contacting Technical Support

Progress DataDirect offers a variety of options to meet your support needs. Please visit our Web site for more details and for contact information:

https://www.progress.com/support

The Progress DataDirect Web site provides the latest support information through our global service network. The SupportLink program provides access to support contact details, tools, patches, and valuable information, including a list of FAQs for each product. In addition, you can search our Knowledgebase for technical bulletins and other information.

When you contact us for assistance, please provide the following information:

- Your number or the serial number that corresponds to the product for which you are seeking support, or a case number if you have been provided one for your issue. If you do not have a SupportLink contract, the SupportLink representative assisting you will connect you with our Sales team.
- Your name, phone number, email address, and organization. For a first-time call, you may be asked for full information, including location.

- The Progress DataDirect product and the version that you are using.
- The type and version of the operating system where you have installed your product.
- Any database, database version, third-party software, or other environment information required to understand the problem.
- A brief description of the problem, including, but not limited to, any error messages you have received, what steps you followed prior to the initial occurrence of the problem, any trace logs capturing the issue, and so on. Depending on the complexity of the problem, you may be asked to submit an example or reproducible application so that the issue can be re-created.
- A description of what you have attempted to resolve the issue. If you have researched your issue on Web search engines, our Knowledgebase, or have tested additional configurations, applications, or other vendor products, you will want to carefully note everything you have already attempted.
- A simple assessment of how the severity of the issue is impacting your organization.

2

Getting started

After the driver has been installed and defined on your class path, you can connect from your application to your database in either of the following ways.

- Using the JDBC `DriverManager`, by specifying the connection URL in the `DriverManager.getConnection()` method.
- Creating a JDBC `DataSource` that can be accessed through the Java Naming Directory Interface (JNDI).

For details, see the following topics:

Data source and driver classes

The driver provides the following driver class.

```
com.ddtek.jdbc.sqlserver.SQLServerDriver
```

The driver provides the following data source class that supports the functionality for all JDBC specifications and Java SE 6 or higher.

```
com.ddtek.jdbcx.sqlserver.SQLServerDataSource
```

See also

Connecting using data sources on page 40

Setting the Classpath

The driver must be defined on your CLASSPATH before you can connect. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate JDBC drivers on your computer. If the driver is not defined on your CLASSPATH, you will receive a `class not found` exception when trying to load the driver. Set your system CLASSPATH to include the `sqlserver.jar` file as shown, where ***`install_dir`*** is the path to your product installation directory.

```
install_dir/lib/sqlserver.jar
```

Windows Example

```
CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC_60\lib\sqlserver.jar
```

UNIX Example

```
CLASSPATH=.:/opt/Progress/DataDirect/JDBC_60/lib/sqlserver.jar
```

Connecting using the DriverManager

One way to connect to a SQL Server database or Azure instance is through the JDBC `DriverManager` using the `DriverManager.getConnection()` method. As the following example shows, this method specifies a string containing a connection URL.

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:sqlserver://server1:1433;User=test;Password=secret;DatabaseName=MyDB");
```

Passing the connection URL

After setting the CLASSPATH, the required connection information needs to be passed in the form of a connection URL.

```
jdbc:datadirect:sqlserver://hostname:port[;property=value[;...]]
```

where:

`hostname`

is the IP address or host name of the server to which you are connecting. See Using IP addresses on page 106 for details on using IP addresses.

`port`

is the number of the TCP/IP port.

property=value

specifies connection properties. For a list of connection properties and their valid values, see Connection property descriptions on page 185.

Notes

- Untrusted applets cannot open a socket to a machine other than the originating host.

Example

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:sqlserver://MyServer:1433;
  User=test;Password=secret;DatabaseName=MyDB");
```

See Connecting to named instances on page 77 for instructions on connecting to named instances.

See also

Using connection properties on page 59

Testing the connection

You can also use DataDirect Test™ to establish and test a `DriverManager` connection. The screen shots in this section were taken on a Windows system.

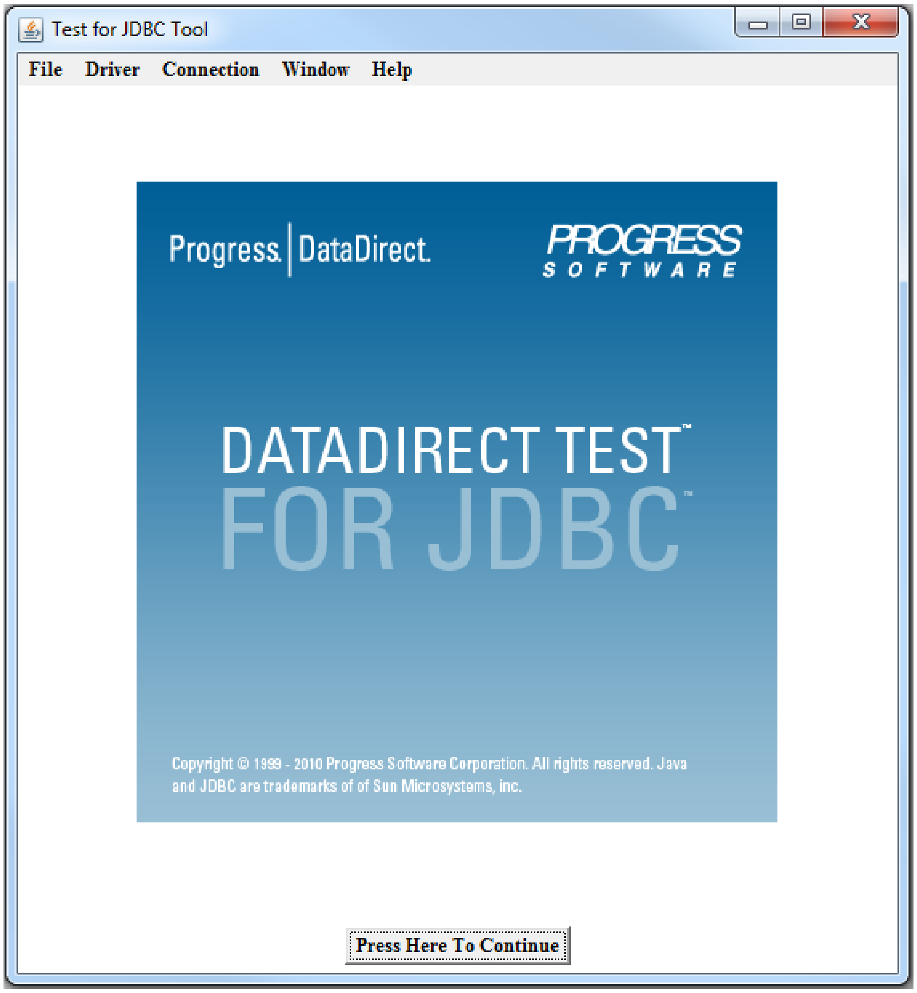
Take the following steps to establish a connection.

1. Navigate to the installation directory. The default location is:
   - Windows systems: `Program Files\Progress\DataDirect\JDBC_60\testforjdbc`
   - UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC_60/testforjdbc`

   Note: For UNIX/Linux, if you do not have access to `/opt`, your home directory will be used in its place.

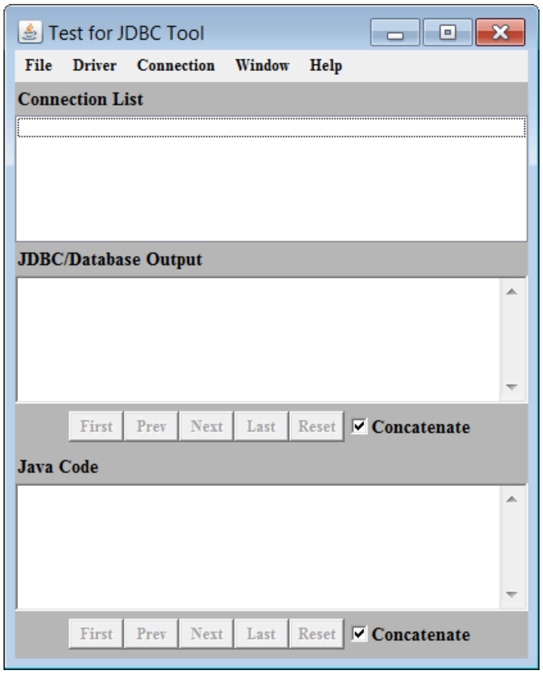
2. From the `testforjdbc` folder, run the platform-specific tool:
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

   The **Test for JDBC Tool** window appears:


Test for JDBC Tool
File Driver Connection Window Help
Progress DataDirect
PROGRESS SOFTWARE
DATADIRECT TEST™
FOR JDBC™
Copyright © 1999 - 2010 Progress Software Corporation. All rights reserved. Java and JDBC are trademarks of of Sun Microsystems, inc.
Press Here To Continue

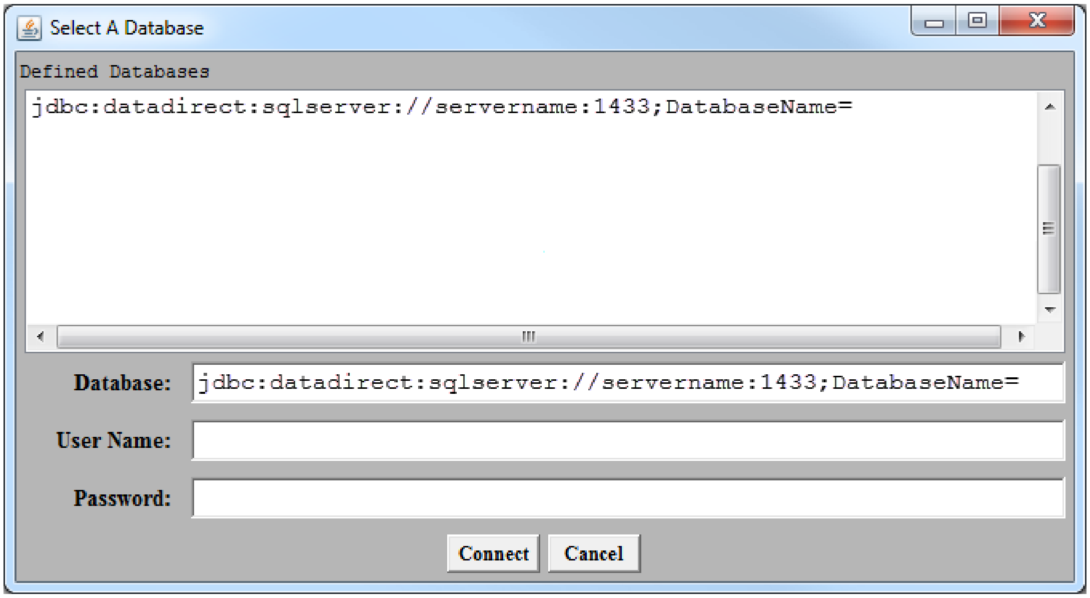
3. Click **Press Here to Continue**.

   The main dialog appears:


Test for JDBC Tool
File Driver Connection Window Help
Connection List
JDBC/Database Output
First Prev Next Last Reset ☑ Concatenate
Java Code
First Prev Next Last Reset ☑ Concatenate

4. From the menu bar, select **Connection > Connect to DB**.

   The **Select A Database** dialog appears:



```
jdbc:datadirect:sqlserver://servername:1433;DatabaseName=
```

Database: `jdbc:datadirect:sqlserver://servername:1433;DatabaseName=`

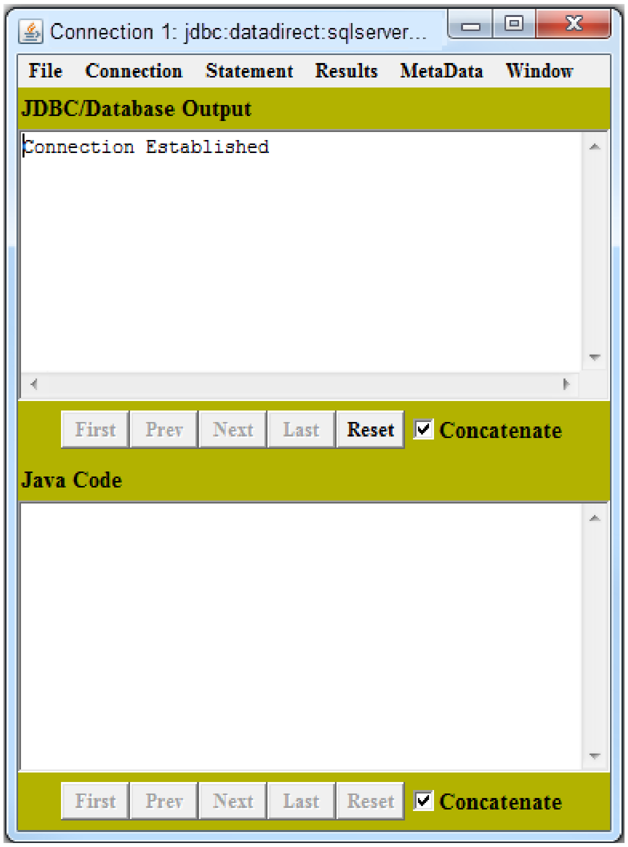
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify the ServerName, PortNumber, and DatabaseName for your SQL Server data source.

   For example:

```
jdbc:datadirect:sqlserver://MyServer:1433;DatabaseName=MyDB
```

7. If you are using user ID/password authentication, enter your user ID and password in the corresponding fields.
8. Click **Connect**.

If the connection information is entered correctly, the **JDBC/Database Output** window reports that a connection has been established. (If a connection is not established, the window reports an error.)



```
Connection Established
```

For more information, see "DataDirect Test."

See also

DataDirect Test on page 147

Connecting using data sources

A *JDBC data source* is a Java object, specifically a `DataSource` object, that defines connection information required for a JDBC driver to connect to the database. Each JDBC driver vendor provides their own data source implementation for this purpose. A Progress DataDirect data source is Progress DataDirect's implementation of a `DataSource` object that provides the connection information needed for the driver to connect to a database.

Because data sources work with the Java Naming Directory Interface (JNDI) naming service, data sources can be created and managed separately from the applications that use them. Because the connection information is defined outside of the application, the effort to reconfigure your infrastructure when a change is made is minimized. For example, if the database is moved to another database server, the administrator need only change the relevant properties of the `DataSource` object. The applications using the database do not need to change because they only refer to the name of the data source.

How data sources are implemented

Data sources are implemented through a data source class. A data source class implements the following interfaces.

- `javax.sql.DataSource`
- `javax.sql.ConnectionPoolDataSource` (allows applications to use connection pooling)

See also

Data source and driver classes on page 15

Connection Pool Manager on page 117

Creating data sources

The following example files provide details on creating and using Progress DataDirect data sources with the Java Naming Directory Interface (JNDI), where *`install_dir`* is the product installation directory.

- *`install_dir`*`/Examples/JNDI/JNDI_LDAP_Example.java` can be used to create a JDBC data source and save it in your LDAP directory using the JNDI Provider for LDAP.
- *`install_dir`*`/Examples/JNDI/JNDI_FILESYSTEM_Example.java` can be used to create a JDBC data source and save it in your local file system using the File System JNDI Provider.

See "Example data source" for an example data source definition for the example files.

To connect using a JNDI data source, the driver needs to access a JNDI data store to persist the data source information. For a JNDI file system implementation, you must download the File System Service Provider from the Oracle Technology Network Java SE Support downloads page, unzip the files to an appropriate location, and add the `fscontext.jar` and `providerutil.jar` files to your CLASSPATH. These steps are not required for LDAP implementations because the LDAP Service Provider has been included with Java SE since Java 2 SDK, v1.3.

Example data source

To configure a data source using the example files, you will need to create a data source definition. The content required to create a data source definition is divided into three sections.

First, you will need to import the data source class. For example:

```
import com.ddtek.jdbcx.sqlserver.SQLServerDataSource;
```

Next, you will need to set the values and define the data source. For example, the following definition contains the minimum properties required for a connection:

```
SQLServerDataSource mds = new SQLServerDataSource();
mds.setDescription("My SQL Server Datasource");
mds.setServerName("MyServer");
mds.setPortNumber(1433);
mds.setUser("User123");
mds.setPassword("secret");
```

Finally, you will need to configure the example application to print out the data source attributes. Note that this code is specific to the driver and should only be used in the example application. For example, you would add the following section for a connection using only the minimum properties:

```
if (ds instanceof SQLServerDataSource)
{
SQLServerDataSource jmds = (SQLServerDataSource) ds;
System.out.println("description=" + jmds.getDescription());
System.out.println("serverName=" + jmds.getServerName());
System.out.println("portNumber=" + jmds.getPortNumber());
System.out.println("user=" + jmds.getUser());
System.out.println("password=" + jmds.getPassword());
System.out.println();
}
```

Calling a data source in an application

Applications can call a Progress DataDirect data source using a logical name to retrieve the `javax.sql.DataSource` object. This object loads the specified driver and can be used to establish a connection to the database.

Once the data source has been registered with JNDI, it can be used by your JDBC application as shown in the following code example.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("EmployeeDB");
Connection con = ds.getConnection("domino", "spark");
```

In this example, the JNDI environment is first initialized. Next, the initial naming context is used to find the logical name of the data source (`EmployeeDB`). The `Context.lookup()` method returns a reference to a Java object, which is narrowed to a `javax.sql.DataSource` object. Then, the `DataSource.getConnection()` method is called to establish a connection.

Testing a data source connection

You can use DataDirect Test™ to establish and test a data source connection. The screen shots in this section were taken on a Windows system.

Take the following steps to establish a connection.

1. Navigate to the installation directory. The default location is:

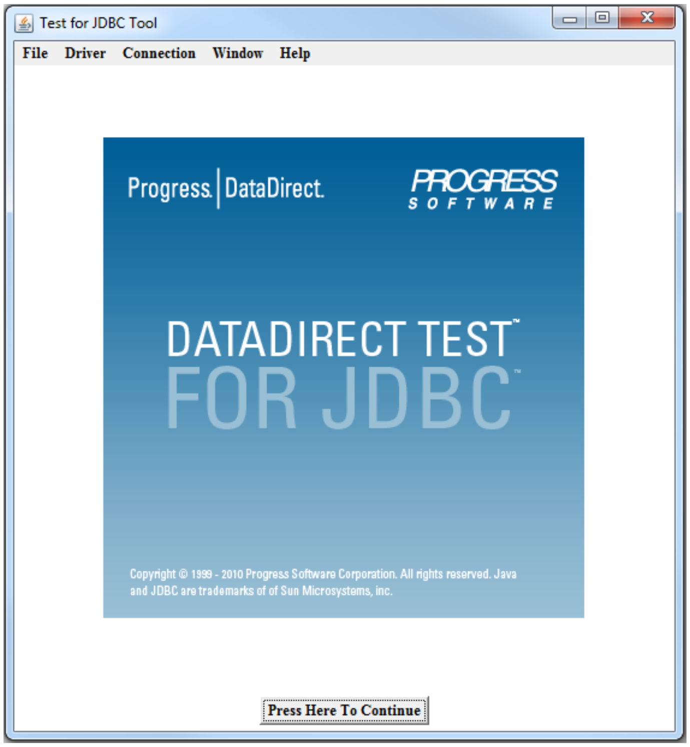
   - Windows systems: `Program Files\Progress\DataDirect\JDBC_60\testforjdbc`
   - UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC_60/testforjdbc`

   Note: For UNIX/Linux, if you do not have access to `/opt`, your home directory will be used in its place.

2. From the `testforjdbc` folder, run the platform-specific tool:

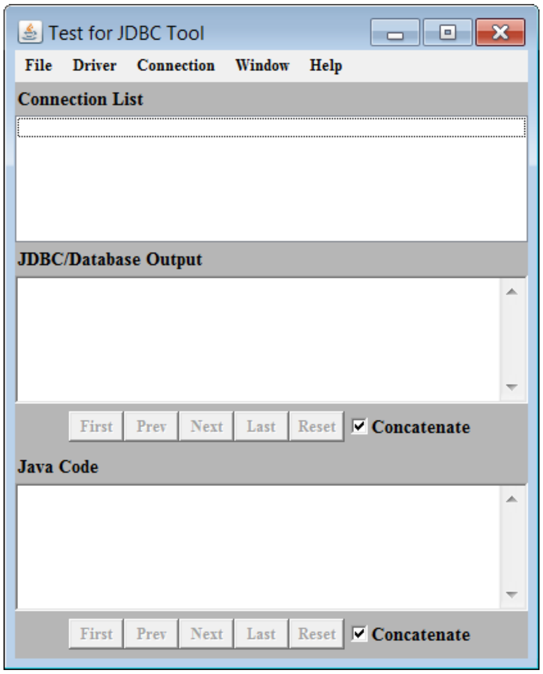
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

   The **Test for JDBC Tool** window appears:


Test for JDBC Tool
File Driver Connection Window Help
Progress DataDirect
PROGRESS SOFTWARE
DATADIRECT TEST™
FOR JDBC™
Copyright © 1999 - 2010 Progress Software Corporation. All rights reserved. Java and JDBC are trademarks of of Sun Microsystems, inc.
Press Here To Continue

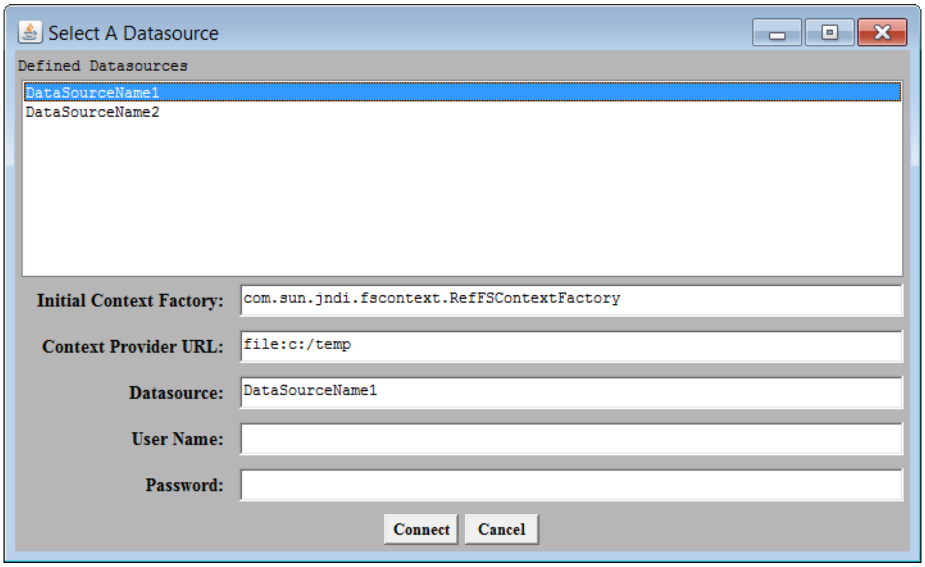
3. Click **Press Here to Continue**.

   The main dialog appears:


Test for JDBC Tool
File Driver Connection Window Help
Connection List
JDBC/Database Output
First Prev Next Last Reset ☑ Concatenate
Java Code
First Prev Next Last Reset ☑ Concatenate

4. From the menu bar, select **Connection > Connect to DB via Data Source**.

   The **Select A Database** dialog appears:



Defined Datasources:

```
DataSourceName1
DataSourceName2
```

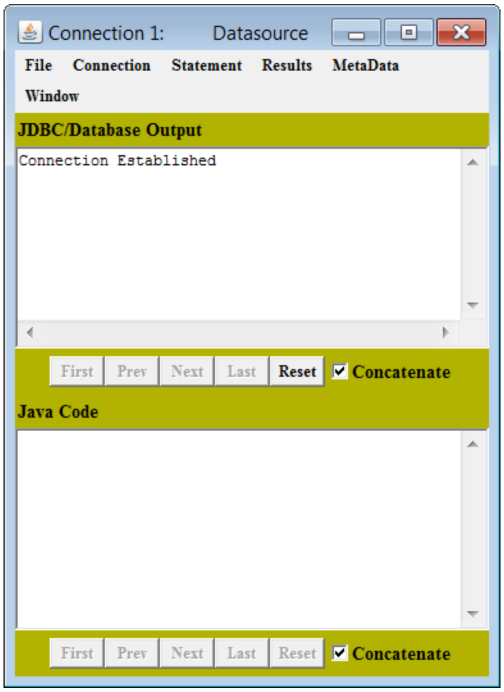
Initial Context Factory: com.sun.jndi.fscontext.RefFSContextFactory

Context Provider URL: file:c:/temp

Datasource: DataSourceName1

5. Select a datasource template from the **Defined Datasources** field.
6. Provide the following information:
   a) In the **Initial Context Factory**, specify the location of the initial context provider for your application.
   b) In the **Context Provider URL**, specify the location of the context provider for your application.
   c) In the **Datasource** field, specify the name of your datasource.
7. If you are using user ID/password authentication, enter your user ID and password in the corresponding fields.
8. Click **Connect**.

If the connection information is entered correctly, the **JDBC/Database Output** window reports that a connection has been established. If a connection is not established, the window reports an error.



```
Connection Established
```

3

Using the driver

This section provides information on how to connect to your data store using either the JDBC Driver Manager or DataDirect JDBC data sources, as well as information on how to implement and use functionality supported by the driver.

For details, see the following topics:

- Required permissions for Java SE with the standard Security Manager enabled
- Connecting from an application
- Using connection properties
- Performance considerations
- Connecting to named instances
- Azure Synapse Analytics and Analytics Platform System
- Authentication
- Data encryption
- Using failover
- Returning and inserting/updating XML data
- DML with results
- Using client information
- Using IP addresses
- Parameter metadata support

- ResultSet metadata support
- Isolation levels
- Using the Snapshot isolation level
- Using scrollable cursors
- Server-side updatable cursors
- JTA support: installing stored procedures
- Distributed transaction cleanup
- Unicode support
- Error handling
- Large object (LOB) support
- Batch Inserts and Updates
- Rowset support
- Auto-generated keys support
- Null values
- Timeouts
- Connection Pool Manager
- Statement Pool Monitor
- DataDirect Bulk Load
- CSV files
- DataDirect Test
- Tracking JDBC calls with DataDirect Spy

Required permissions for Java SE with the standard Security Manager enabled

Using the driver on a Java platform with the standard Security Manager enabled requires certain permissions to be set in the Java SE security policy file `java.policy`. The default location of this file is *`java_install_dir`*`/jre/lib/security`.

Note: Security manager may be enabled by default in certain scenarios, such as running on an application server or in a Web browser applet.

To run an application on a Java platform with the standard Security Manager, use the following command:

```
"java -Djava.security.manager application_class_name"
```

where *`application_class_name`* is the class name of the application.

Refer to your Java documentation for more information about setting permissions in the security policy file.

Permissions for establishing connections

To establish a connection to the database server, the driver must be granted the permissions as shown in the following example:

```
grant codeBase "file:/install_dir/lib/-" {
   permission java.net.SocketPermission "*", "connect";
};
```

where:

install_dir

is the product installation directory.

Granting access to Java properties

To allow the driver to read the value of various Java properties to perform certain operations, permissions must be granted as shown in the following example:

```
grant codeBase "file:/install_dir/lib/-" {
   permission java.util.PropertyPermission "*", "read, write";
};
```

where:

install_dir

is the product installation directory.

Granting access to temporary files

Access to the temporary directory specified by the JVM configuration must be granted in the Java SE security policy file to use insensitive scrollable cursors or to perform client-side sorting of DatabaseMetaData result sets. The following example shows permissions that have been granted for the `C:\TEMP` directory:

```
grant codeBase "file:/install_dir/lib/-" {
// Permission to create and delete temporary files.
// Adjust the temporary directory for your environment.
   permission java.io.FilePermission "C:\\TEMP\\-", "read,write,delete";
};
```

where:

install_dir

is the product installation directory.

Permissions for bulk load from a CSV file

To bulk load data from a comma-separated value (CSV) file with the drivers, the application and driver code bases must be granted security permissions in the security policy file of the Java Platform as shown in the following examples.

```
grant codeBase "file:/install_dir/lib/-" {
   permission java.util.PropertyPermission "true", "read";
   permission java.util.PropertyPermission "file.encoding", "read";
   permission java.util.PropertyPermission "user.dir", "read";
   permission java.lang.RuntimePermission "readFileDescriptor";
};
```

Connecting from an application

Once the driver is installed and configured, you can connect from your application to your database in either of the following ways:

- Using the JDBC Driver Manager, by specifying the connection URL in the `DriverManager.getConnection()` method.
- Creating a JDBC data source that can be accessed through the Java Naming Directory Interface (JNDI).

Data source and driver classes

The driver provides the following driver class.

`com.ddtek.jdbc.sqlserver.SQLServerDriver`

The driver provides the following data source class that supports the functionality for all JDBC specifications and Java SE 6 or higher.

`com.ddtek.jdbcx.sqlserver.SQLServerDataSource`

See also

Connecting using data sources on page 40

Setting the Classpath

The driver must be defined on your CLASSPATH before you can connect. The CLASSPATH is the search string your Java Virtual Machine (JVM) uses to locate JDBC drivers on your computer. If the driver is not defined on your CLASSPATH, you will receive a `class not found` exception when trying to load the driver. Set your system CLASSPATH to include the `sqlserver.jar` file as shown, where *`install_dir`* is the path to your product installation directory.

```
install_dir/lib/sqlserver.jar
```

Windows Example

```
CLASSPATH=.;C:\Program Files\Progress\DataDirect\JDBC_60\lib\sqlserver.jar
```

UNIX Example

```
CLASSPATH=.:/opt/Progress/DataDirect/JDBC_60/lib/sqlserver.jar
```

Connecting using the DriverManager

One way to connect to a SQL Server database or Azure instance is through the JDBC `DriverManager` using the `DriverManager.getConnection()` method. As the following example shows, this method specifies a string containing a connection URL.

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:sqlserver://server1:1433;User=test;Password=secret;DatabaseName=MyDB");
```

Passing the connection URL

After setting the CLASSPATH, the required connection information needs to be passed in the form of a connection URL.

```
jdbc:datadirect:sqlserver://hostname:port[;property=value[;...]]
```

where:

`hostname`

is the IP address or host name of the server to which you are connecting. See Using IP addresses on page 106 for details on using IP addresses.

`port`

is the number of the TCP/IP port.

`property=value`

specifies connection properties. For a list of connection properties and their valid values, see Connection property descriptions on page 185.

Notes

- Untrusted applets cannot open a socket to a machine other than the originating host.

Example

```
Connection conn = DriverManager.getConnection
 ("jdbc:datadirect:sqlserver://MyServer:1433;
  User=test;Password=secret;DatabaseName=MyDB");
```

See Connecting to named instances on page 77 for instructions on connecting to named instances.

See also

Using connection properties on page 59

Testing the connection

You can also use DataDirect Test™ to establish and test a `DriverManager` connection. The screen shots in this section were taken on a Windows system.

Take the following steps to establish a connection.

1. Navigate to the installation directory. The default location is:

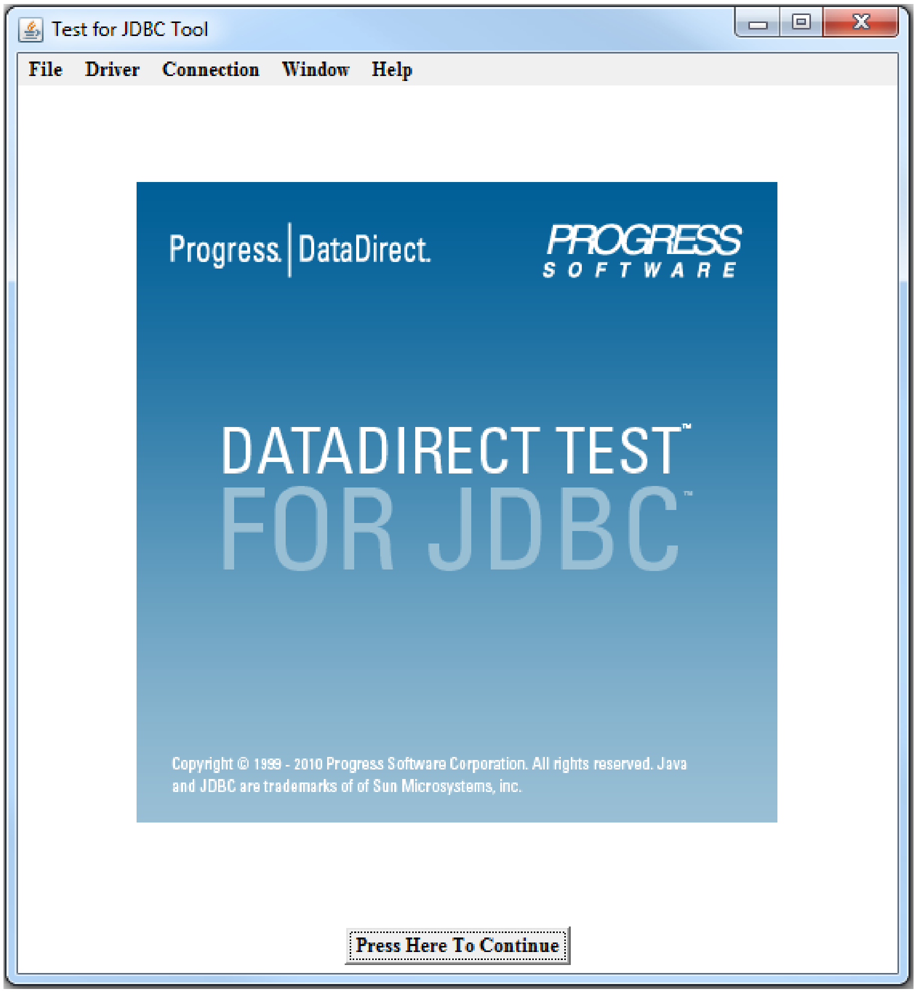
   - Windows systems: `Program Files\Progress\DataDirect\JDBC_60\testforjdbc`
   - UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC_60/testforjdbc`

   Note: For UNIX/Linux, if you do not have access to `/opt`, your home directory will be used in its place.

2. From the `testforjdbc` folder, run the platform-specific tool:

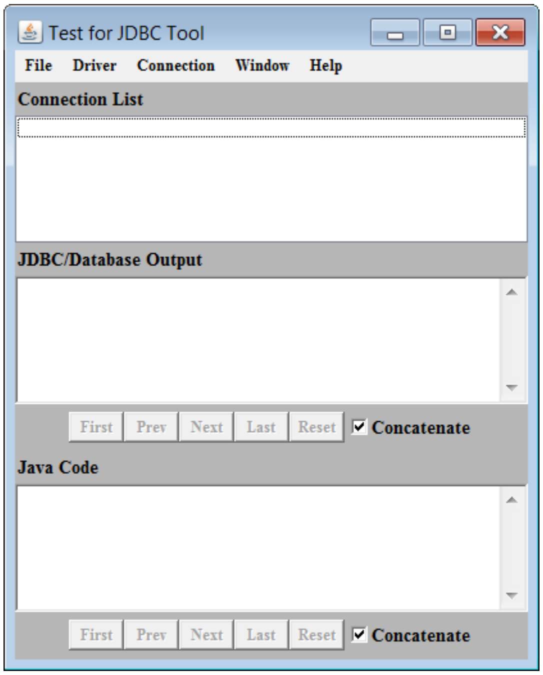
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

   The **Test for JDBC Tool** window appears:



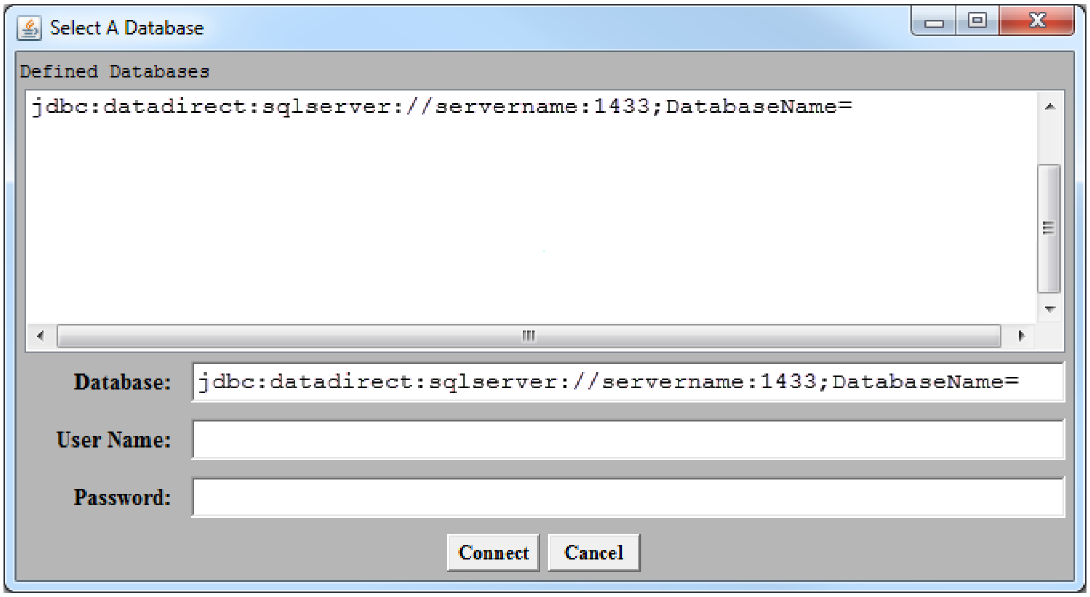
3. Click **Press Here to Continue**.

   The main dialog appears:


Test for JDBC Tool
File Driver Connection Window Help
Connection List
JDBC/Database Output
First Prev Next Last Reset ☑ Concatenate
Java Code
First Prev Next Last Reset ☑ Concatenate

4. From the menu bar, select **Connection > Connect to DB**.

   The **Select A Database** dialog appears:



jdbc:datadirect:sqlserver://servername:1433;DatabaseName=

Database: jdbc:datadirect:sqlserver://servername:1433;DatabaseName=

User Name:

Password:

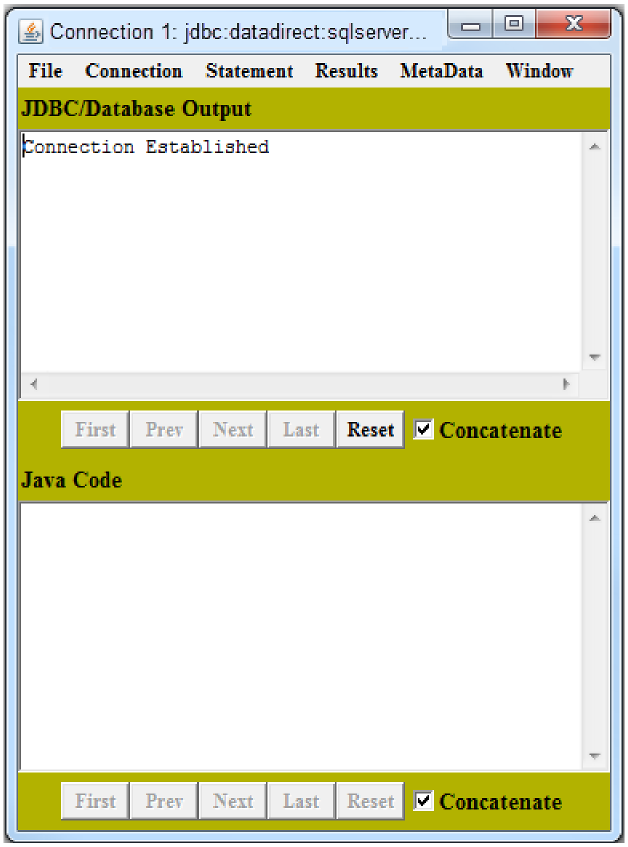
5. Select the appropriate database template from the **Defined Databases** field.
6. In the **Database** field, specify the ServerName, PortNumber, and DatabaseName for your SQL Server data source.

   For example:

   ```
   jdbc:datadirect:sqlserver://MyServer:1433;DatabaseName=MyDB
   ```

7. If you are using user ID/password authentication, enter your user ID and password in the corresponding fields.
8. Click **Connect**.

If the connection information is entered correctly, the **JDBC/Database Output** window reports that a connection has been established. (If a connection is not established, the window reports an error.)



```
Connection Established
```

For more information, see "DataDirect Test."

See also

DataDirect Test on page 147

Connecting using data sources

A *JDBC data source* is a Java object, specifically a `DataSource` object, that defines connection information required for a JDBC driver to connect to the database. Each JDBC driver vendor provides their own data source implementation for this purpose. A Progress DataDirect data source is Progress DataDirect's implementation of a `DataSource` object that provides the connection information needed for the driver to connect to a database.

Because data sources work with the Java Naming Directory Interface (JNDI) naming service, data sources can be created and managed separately from the applications that use them. Because the connection information is defined outside of the application, the effort to reconfigure your infrastructure when a change is made is minimized. For example, if the database is moved to another database server, the administrator need only change the relevant properties of the `DataSource` object. The applications using the database do not need to change because they only refer to the name of the data source.

How data sources are implemented

Data sources are implemented through a data source class. A data source class implements the following interfaces.

- `javax.sql.DataSource`
- `javax.sql.ConnectionPoolDataSource` (allows applications to use connection pooling)

See also

Data source and driver classes on page 15

Connection Pool Manager on page 117

Creating data sources

The following example files provide details on creating and using Progress DataDirect data sources with the Java Naming Directory Interface (JNDI), where *`install_dir`* is the product installation directory.

- *`install_dir`*`/Examples/JNDI/JNDI_LDAP_Example.java` can be used to create a JDBC data source and save it in your LDAP directory using the JNDI Provider for LDAP.
- *`install_dir`*`/Examples/JNDI/JNDI_FILESYSTEM_Example.java` can be used to create a JDBC data source and save it in your local file system using the File System JNDI Provider.

See "Example data source" for an example data source definition for the example files.

To connect using a JNDI data source, the driver needs to access a JNDI data store to persist the data source information. For a JNDI file system implementation, you must download the File System Service Provider from the Oracle Technology Network Java SE Support downloads page, unzip the files to an appropriate location, and add the `fscontext.jar` and `providerutil.jar` files to your CLASSPATH. These steps are not required for LDAP implementations because the LDAP Service Provider has been included with Java SE since Java 2 SDK, v1.3.

Example data source

To configure a data source using the example files, you will need to create a data source definition. The content required to create a data source definition is divided into three sections.

First, you will need to import the data source class. For example:

```
import com.ddtek.jdbcx.sqlserver.SQLServerDataSource;
```

Next, you will need to set the values and define the data source. For example, the following definition contains the minimum properties required for a connection:

```
SQLServerDataSource mds = new SQLServerDataSource();
mds.setDescription("My SQL Server Datasource");
mds.setServerName("MyServer");
mds.setPortNumber(1433);
mds.setUser("User123");
mds.setPassword("secret");
```

Finally, you will need to configure the example application to print out the data source attributes. Note that this code is specific to the driver and should only be used in the example application. For example, you would add the following section for a connection using only the minimum properties:

```
if (ds instanceof SQLServerDataSource)
{
```

```
SQLServerDataSource jmds = (SQLServerDataSource) ds;
System.out.println("description=" + jmds.getDescription());
System.out.println("serverName=" + jmds.getServerName());
System.out.println("portNumber=" + jmds.getPortNumber());
System.out.println("user=" + jmds.getUser());
System.out.println("password=" + jmds.getPassword());
System.out.println();
}
```

Calling a data source in an application

Applications can call a Progress DataDirect data source using a logical name to retrieve the `javax.sql.DataSource` object. This object loads the specified driver and can be used to establish a connection to the database.

Once the data source has been registered with JNDI, it can be used by your JDBC application as shown in the following code example.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("EmployeeDB");
Connection con = ds.getConnection("domino", "spark");
```

In this example, the JNDI environment is first initialized. Next, the initial naming context is used to find the logical name of the data source (`EmployeeDB`). The `Context.lookup()` method returns a reference to a Java object, which is narrowed to a `javax.sql.DataSource` object. Then, the `DataSource.getConnection()` method is called to establish a connection.

Testing a data source connection

You can use DataDirect Test™ to establish and test a data source connection. The screen shots in this section were taken on a Windows system.

Take the following steps to establish a connection.

1. Navigate to the installation directory. The default location is:

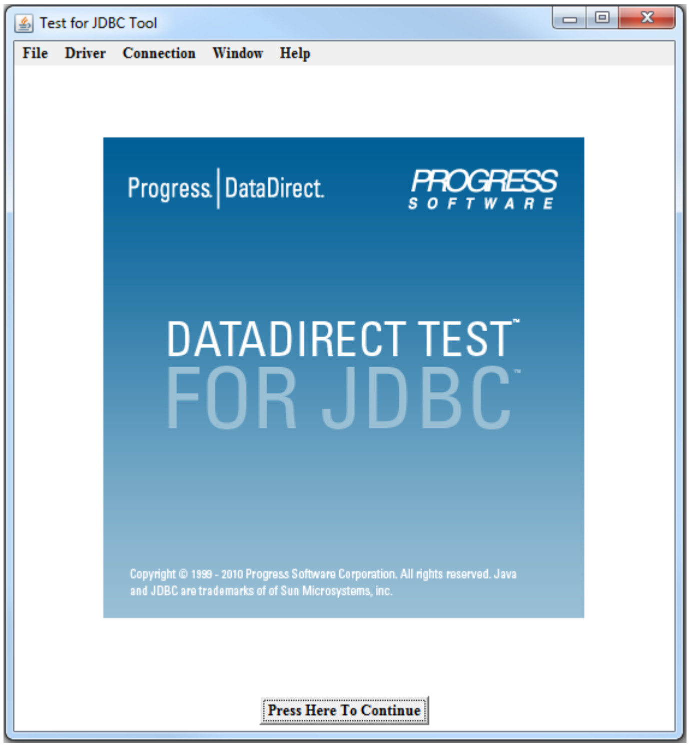
   - Windows systems: `Program Files\Progress\DataDirect\JDBC_60\testforjdbc`
   - UNIX and Linux systems: `/opt/Progress/DataDirect/JDBC_60/testforjdbc`

   Note: For UNIX/Linux, if you do not have access to `/opt`, your home directory will be used in its place.

2. From the `testforjdbc` folder, run the platform-specific tool:

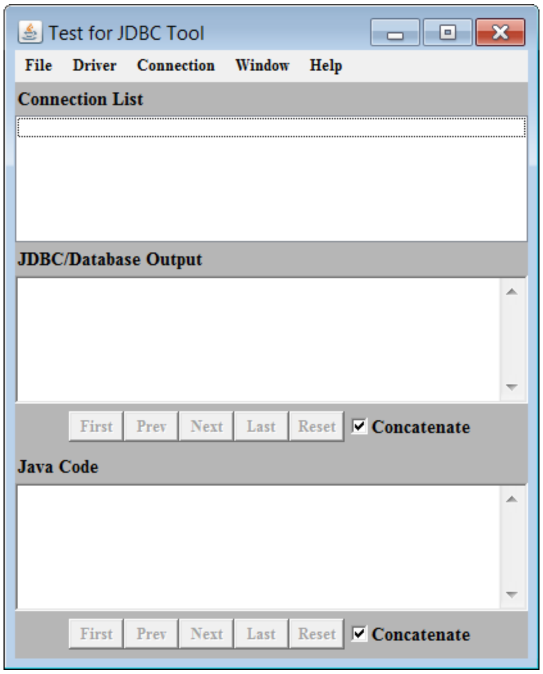
   - `testforjdbc.bat` (on Windows systems)
   - `testforjdbc.sh` (on UNIX and Linux systems)

   The **Test for JDBC Tool** window appears:


Test for JDBC Tool
File Driver Connection Window Help
Progress | DataDirect.
PROGRESS SOFTWARE
DATADIRECT TEST™
FOR JDBC™
Copyright © 1999 - 2010 Progress Software Corporation. All rights reserved. Java and JDBC are trademarks of of Sun Microsystems, inc.
Press Here To Continue

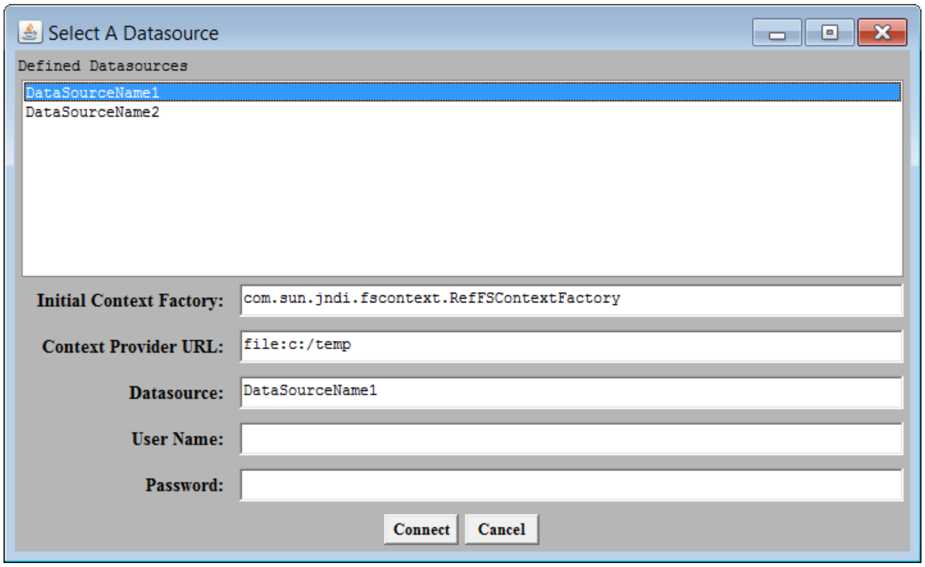
3. Click **Press Here to Continue**.

   The main dialog appears:


Test for JDBC Tool
File Driver Connection Window Help
Connection List
JDBC/Database Output
First Prev Next Last Reset ☑ Concatenate
Java Code
First Prev Next Last Reset ☑ Concatenate

4. From the menu bar, select **Connection > Connect to DB via Data Source**.

   The **Select A Database** dialog appears:



DataSourceName1
DataSourceName2

Initial Context Factory: com.sun.jndi.fscontext.RefFSContextFactory

Context Provider URL: file:c:/temp

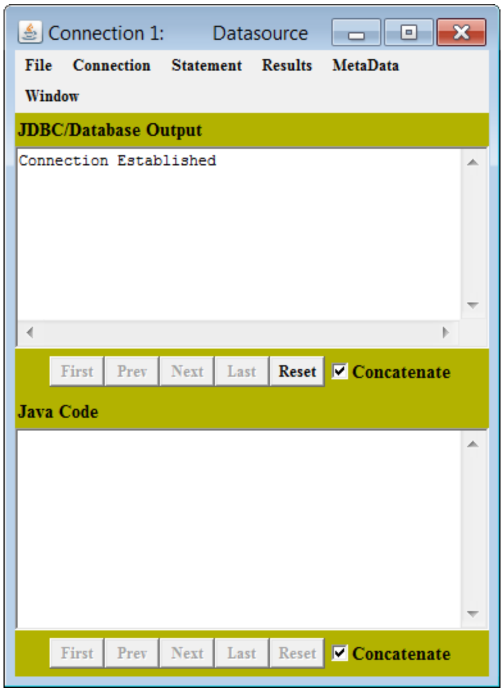
Datasource: DataSourceName1

User Name:

Password:

5. Select a datasource template from the **Defined Datasources** field.
6. Provide the following information:
   a) In the **Initial Context Factory**, specify the location of the initial context provider for your application.
   b) In the **Context Provider URL**, specify the location of the context provider for your application.
   c) In the **Datasource** field, specify the name of your datasource.
7. If you are using user ID/password authentication, enter your user ID and password in the corresponding fields.
8. Click **Connect**.

If the connection information is entered correctly, the **JDBC/Database Output** window reports that a connection has been established. If a connection is not established, the window reports an error.



```
Connection Established
```

Using connection properties

You can use connection properties to customize the driver for your environment. This section organizes connection properties according to functionality. You can use connection properties with either the JDBC `DriverManager` or a JDBC data source. For a `DriverManager` connection, a property is expressed as a key value pair and takes the form *`property=value`*. For a data source connection, a property is expressed as a JDBC method and takes the form `set`*`property`*`(`*`value`*`)`.

Note: Connection property names are case-insensitive. For example, `Password` is the same as `password`.

Note: In a JDBC data source, string values must be enclosed in double quotation marks, for example, `setUser("User123")`.

See "Connection property descriptions" for an alphabetical list of connection properties and their descriptions.

See also

Connecting using the DriverManager on page 36
Connecting using data sources on page 40
Connection property descriptions on page 185

Required properties

The following table summarizes connection properties required to connect to a database.

Table 2: Required properties

Property	Characteristic
PortNumber on page 236	Specifies the TCP port of the primary database server that is listening for connections to the database. The default is `1433`.
ServerName on page 241	Specifies the name or IP address of the server to which you want to connect.

See also

Connection property descriptions on page 185

Authentication properties

The following table describes the connection properties used to configure authentication.

Table 3: Authentication properties

Property	Characteristic
AuthenticationMethod on page 201	Determines which authentication method the driver uses when establishing a connection. The default is `auto`.
Domain on page 216	Specifies the name of the domain server that administers the database. Set this property only if you are using NTLM authentication. If the Domain property is unspecified, the driver tries to determine the domain server name from the User property.
GSSCredential on page 223	Specifies the GSS credential object used to instantiate Kerberos constrained delegation. Constrained delegation is a Kerberos mechanism that allows a client application to delegate authentication to a second service. **Important:** Because the value of this property is a Java object, it cannot be specified in a connection URL. It can only be passed as a `Properties` or `DataSource` object.
LoginConfigName on page 229	Specifies the name of the entry in the JAAS login configuration file that contains the authentication technology used by the driver to establish a Kerberos connection. The default is `JDBC_DRIVER_01`.
Password on page 236	Specifies a password that is used to connect to the database or instance.
ServicePrincipalName on page 242	Specifies the service principal name to be used for Kerberos authentication.
User on page 250	Specifies the user ID for user ID/password authentication or the domain user name for NTLM authentication.

See also

Authentication on page 79

Connection property descriptions on page 185

Data encryption properties

The following table summarizes connection properties which can be used to enable SSL.

Table 4: Data encryption properties

Property	Characteristic
CryptoProtocolVersion on page 210	Specifies a cryptographic protocol or comma-separated list of cryptographic protocols that can be used when SSL is enabled (`EncryptionMethod=SSL`).
EncryptionMethod on page 218	Determines whether data is encrypted and decrypted when transmitted over the network between the driver and database server. The default is `noEncryption`.
HostNameInCertificate on page 224	Specifies a host name for certificate validation when SSL encryption is enabled (`EncryptionMethod=SSL`) and validation is enabled (`ValidateServerCertificate=true`). This property is optional and provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.
TrustStore on page 248	Specifies the directory of the truststore file to be used when SSL is enabled (`EncryptionMethod=SSL`) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.
TrustStorePassword on page 249	Specifies the password that is used to access the truststore file when SSL is enabled (`EncryptionMethod=SSL`) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.
ValidateServerCertificate on page 251	Determines whether the driver validates the certificate that is sent by the database server when SSL encryption is enabled (`EncryptionMethod=SSL`). When using SSL server authentication, any certificate that is sent by the server must be issued by a trusted Certificate Authority (CA). The default is `true`.

See also

Data encryption on page 87

Connection property descriptions on page 185

Failover properties

The following table summarizes the connection properties used for configuring failover.

Table 5: Failover properties

Property	Characteristic
AlternateServers on page 198	One or multiple alternate database servers. An IP address or server name identifying each server is required. Port number and the connection property DatabaseName are optional. If the port number is unspecified, the port number specified for the primary server is used. If a port number is unspecified for the primary server, the default port number of `1433` is used.
ConnectionRetryCount on page 208	Number of times the driver retries the primary database server, and if specified, alternate servers until a successful connection is established. The default is `5`.
ConnectionRetryDelay on page 209	Wait interval, in seconds, between connection retry attempts when the ConnectionRetryCount property is set to a positive integer. The default is `1`.
FailoverGranularity on page 219	Determines whether the driver fails the entire failover process or continues with the process if exceptions occur while trying to reestablish a lost connection. The default is `nonAtomic` (the driver continues with the failover process and posts any exceptions on the statement on which they occur).
FailoverMode on page 220	The failover method you want the driver to use. The default is `connect` (connection failover is used).
FailoverPreconnect on page 221	Specifies whether the driver tries to connect to the primary and an alternate server at the same time. The default is `false` (the driver tries to connect to an alternate server only when failover is caused by an unsuccessful connection attempt or a lost connection).

Property	Characteristic
LoadBalancing on page 228	Sets whether the driver will use client load balancing in its attempts to connect to the database servers (primary and alternate). If client load balancing is enabled, the driver uses a random pattern instead of a sequential pattern in its attempts to connect. The default is `false` (client load balancing is disabled).
MultiSubnetFailover on page 233	Determines whether the driver attempts parallel connections to the failover IP addresses of an Availability Group during initial connection or a multi-subnet failover. When MultiSubnetFailover is enabled, the driver simultaneously attempts to connect to all IP addresses associated with the Availability Group listener when establishing an initial connection or reconnecting after a connection is broken or the listener IP address becomes unavailable. The first IP address to successfully respond to the request is used for the connection. Using parallel-connection attempts offers improved response time over traditional failover, which attempts to connect to alternate servers one at a time. The default is `false` (disabled).

See also

Using failover on page 92
Configuring failover on page 93
Using client load balancing on page 99
Always On Availability Groups on page 100
Connection property descriptions on page 185

Bulk load properties

The following table summarizes the connection properties used to configure how bulk operations are executed by the driver.

Note: The driver also supports DataDirect Bulk Load. If you are developing a new application that needs to perform bulk load operations, you can use DataDirect Bulk Load to send large numbers of rows of data to a database. See DataDirect Bulk Load on page 139 for details.

Table 6: Bulk load properties

Property	Description
BulkLoadBatchSize on page 202	Provides a suggestion to the driver for the number of rows to load to the database at a time when bulk loading data. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Be aware that increasing the number of rows that are loaded also causes the driver to consume more memory on the client. The default is `1000`.

Property	Description
BulkLoadOptions on page 203	Enables bulk load protocol options for batch inserts that the driver can take advantage of when EnableBulkLoad is set to a value of `true`. By default, the TableLock option assigns a table lock for the duration of the bulk copy operation. Other applications cannot update the table until the operation completes. If unspecified, the default bulk locking mechanism specified by the database server is used.
EnableBulkLoad on page 216	Specifies whether the driver uses the native bulk load protocols in the database. Bulk load bypasses the data parsing that is usually done by the database, providing an additional performance gain over batch operations. This property allows existing applications with batch inserts to take advantage of bulk load without requiring changes to the application code. By default, the driver uses standard parameter arrays to perform batch inserts instead of the native bulk load protocols in the database.

See also

DataDirect Bulk Load on page 139
Connection property descriptions on page 185

Data type handling properties

The following table summarizes connection properties which can be used to handle data types.

Table 7: Data type handling properties

Property	Characteristic
ConvertNull on page 210	Controls how data conversions are handled for null values. By default, the driver checks the data type this is requested against the data type of the table column that stores the data. If a conversion between the requested type and column type is not defined, the driver generates an "unsupported data conversion" exception regardless of the data type of the column value.

Property	Characteristic
DateTimeInputParameterType on page 212	Specifies how the driver describes the data type for Date/Time/Timestamp input parameters. By default, the driver uses the following rules to describe the data type of Date/Time/Timestamp input parameters: • If an input parameter is set using setDate(), the driver describes it as date. • If an input parameter is set using setTime(), the driver describes it as time. • If an input parameter is set using setTimestamp(), the driver describes it as datetimeoffset.
DateTimeOutputParameterType on page 213	Specifies how the driver describes the data type for Date/Time/Timestamp output parameters. By default, the driver uses the following rules to describe the data type of Date/Time/Timestamp output parameters: • If an output parameter is set using setDate(), the driver describes it as date. • If an output parameter is set using setTime(), the driver describes it as time. • If an output parameter is set using setTimestamp(), the driver describes it as datetimeoffset.
DescribeInputParameters on page 214	Determines whether the driver attempts to determine, at execute time, which data type to use to send input parameters to the database server. Sending parameters as the data type the database expects improves performance and prevents locking issues caused by data type mismatches. By default, the driver does not attempt to describe input parameters and sends String and Date/Time/Timestamp input parameters to the server as specified by the StringInputParameterType and DateTimeInputParameterType properties.
DescribeOutputParameters on page 215	Determines whether the driver attempts to determine, at execute time, which data type to use to send output parameters to the database server. Sending parameters as the data type the database expects improves performance and prevents locking issues caused by data type mismatches. By default, the driver does not attempt to describe output parameters and sends String and Date/Time/Timestamp output parameters to the server as specified by the StringOutputParameterType and DateTimeOutputParameterType properties.

Property	Characteristic
FetchTSWTZAsTimestamp on page 221	Determines whether column values with the datetimeoffset data type are returned as a JDBC VARCHAR or TIMESTAMP data type. If set to `true`, column values with the datetimeoffset data type are returned as a JDBC TIMESTAMP data type. If set to `false`, column values with the datetimeoffset data type are returned as a JDBC VARCHAR data type. The default is `false`.
FetchTWFSasTime on page 222	Determines whether the driver returns column values for the native TIME data type as the JDBC TIME or TIMESTAMP data type. If set to `true`, the driver returns column values for the native TIME data type as the JDBC TIME data type. The fractional seconds portion of the value is truncated when the value is returned in the java.sql.Time object. If set to `false`, the driver returns column values for the native TIME data type as the JDBC TIMESTAMP data type. The Java Epoch (Jan 1,1970) is returned in the date portion. The default is `false`.
JavaDoubleToString on page 227	Determines which algorithm the driver uses when converting a double or float value to a string value. By default, the driver uses its own internal conversion algorithm, which improves performance.
JDBCBehavior on page 228	Determines how the driver describes database data types that map to the following JDBC 4.0 data types: NCHAR, NVARCHAR, NLONGVARCHAR, NCLOB, and SQLXML. By default, the driver describes the data types using JDBC 3.0-equivalent data types. This allows your application to continue using JDBC 3.0 types in a Java SE 6 or higher environment. Additionally, the PROCEDURE_NAME column contains procedure name qualifiers. For example, for the fully qualified procedure named 1.sp_productadd, the driver would return sp_productadd;1.
XMLDescribeType on page 253	Determines whether the driver maps XML data to the LONGVARCHAR or LONGVARBINARY data type.

See also

Connection property descriptions on page 185

Timeout properties

The following table summarizes timeout connection properties.

Table 8: Timeout properties

Property	Characteristic
EnableCancelTimeout on page 217	Determines whether a cancel request that is sent by the driver as the result of a query timing out is subject to the same query timeout value as the statement it cancels. If set to `true`, the cancel request times out using the same timeout value, in seconds, that is set for the statement it cancels. For example, if your application calls `Statement.setQueryTimeout(5)` on a statement and that statement is cancelled because its timeout value was exceeded, the driver sends a cancel request that also will time out if its execution exceeds 5 seconds. If the cancel request times out, because the server is down, for example, the driver throws an exception indicating that the cancel request was timed out and the connection is no longer valid. If set to `false`, the cancel request does not time out. The default is `false`.
LoginTimeout on page 230	Specifies the amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request. If set to `0`, the driver does not time out a connection request. If set to *x*, the driver waits for the specified number of seconds before returning control to the application and throwing a timeout exception. The default is `0`.
QueryTimeout on page 238	Sets the default query timeout (in seconds) for all statements created by a connection. If set to `-1`, the query timeout functionality is disabled. If set to `0`, the default query timeout is infinite (query does not time out). If set to *x*, the driver uses the value as the default timeout for any statement that is created by the connection. The default is `0`.

See also

Timeouts on page 117

Connection property descriptions on page 185

Statement pooling properties

The following table summarizes statement pooling connection properties.

Table 9: Statement pooling properties

Property	Characteristic
ImportStatementPool on page 225	Specifies the path and file name of the file to be used to load the contents of the statement pool. When this property is specified, statements are imported into the statement pool from the specified file.
MaxPooledStatements on page 232	Specifies the maximum number of prepared statements to be pooled for each connection and enables the driver's internal prepared statement pooling when set to an integer greater than zero (0). The driver's internal prepared statement pooling provides performance benefits when the driver is not running from within an application server or another application that provides its own statement pooling.
RegisterStatementPoolMonitorMBean on page 238	Registers the Statement Pool Monitor as a JMX MBean when statement pooling has been enabled with MaxPooledStatements. This allows you to manage statement pooling with standard JMX API calls and to use JMX-compliant tools, such as JConsole.

See also

Statement Pool Monitor on page 131
Performance considerations on page 75
Connection property descriptions on page 185

Client information properties

The following table summarizes connection properties which can be used to return client information.

Table 10: Client information properties

Property	Characteristic
AccountingInfo on page 192	Defines accounting information. This value is stored locally and is used for database administration/monitoring purposes.
ApplicationName on page 200	The name of the application to be stored in the database. This property sets the program_name column in the sysprocesses table in the database.
ClientHostName on page 205	The host name, or workstation ID, of the client machine to be stored in the database. This property sets the hostname column of the sysprocesses table in the database.
ClientUser on page 205	Specifies the user ID. This value is stored locally and is used for database administration/monitoring purposes.

Property	Characteristic
NetAddress on page 234	The Media Access Control (MAC) address of the network interface card of the application connecting to Microsoft SQL Server. This value is stored in the net_address column of the sys.sysprocesses table.
ProgramID on page 237	The driver name and version information on the client to be stored in the database. This property sets the hostprocess column in the sysprocesses table.

See also

Using client information on page 104

Connection property descriptions on page 185

Always Encrypted properties

The following table summarizes connection properties related to Always Encrypted functionality.

Table 11: Always Encrypted properties

Property	Characteristic
AEKeyCacheTTL on page 193	Specifies the length of time, in seconds, column encryption keys live in the cache before the driver deletes them. This property is used when Always Encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`). If set to `-1`, the driver caches column encryption keys for the life of the connection. The keys are deleted when the connection is closed or sent to the connection pool. If set to `0`, the driver does not cache column encryption keys. If set to *x*, the driver caches column encryption keys for the specified number of seconds before deleting them. The timer starts for a key when it is first accessed and added to the cache. The timer does not reset if you access it after it has been added to the cache. The keys are deleted when the timer expires, or the connection is closed or sent to the connection pool. **Note:** While caching can improve performance, column encryption keys are designed to be deleted from the cache as a security measure and should not be stored for long periods of time. The default is `7200`.

Property	Characteristic
ColumnEncryption on page 207	Specifies whether the driver is enabled for Always Encrypted functionality when accessing data from encrypted columns. If set to `Disabled`, the driver does not use Always Encrypted functionality. The driver does not attempt to decrypt data from encrypted columns, but will return data as binary formatted cipher text. However, statements containing parameters that reference encrypted columns are not supported and will return an error. If set to `ResultsetOnly`, the driver transparently decrypts result sets and returns them to the application. Queries containing parameters that affect encrypted columns will return an error. If set to `Enabled`, the driver fully supports Always Encrypted functionality. The driver transparently decrypts result sets and returns them to the application. In addition, the driver transparently encrypts parameter values that are associated with encrypted columns. The default is `Disabled`.
Azure Key Vault properties	
AEKeystoreClientSecret on page 194	Specifies the Client Secret used to authenticate against the Azure Key Vault. This property is used only when Always Encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`) and Azure Key Vault is the keystore provider. The Azure Key Vault stores the column master key used for Always Encrypted functionality. To access the column master key from the Azure Key Vault, the Client Secret and principal ID must be provided.
AEKeystorePrincipalId on page 196	Specifies the principal ID used to authenticate against the Azure Key Vault. This property is used only when Always Encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`) and Azure Key Vault is the keystore provider. The Azure Key Vault stores the column master key used for Always Encrypted functionality. To access the column master key from the Azure Key Vault, the principal ID and Client Secret must be provided. **Note:** The driver currently supports only Azure App Registration as the principal ID.
Java KeyStore properties	

Property	Characteristic
AEKeystoreLocation on page 195	Specifies absolute path to the Java KeyStore file. This property is used only when Always Encrypted is enabled (`ColumnEncryption=Enabled | ResultsetOnly`) and Java KeyStore is the keystore provider. The Java KeyStore contains the column master key used for Always Encrypted functionality. To specify the password for the Java KeyStore file, use the AEKeyStoreSecret property.
AEKeystoreSecret on page 197	Specifies the password used to access the Java KeyStore file. This property is used only when Always Encrypted is enabled (`ColumnEncryption=Enabled|ResultsetOnly`) and Java KeyStore is the keystore provider. The Java KeyStore contains the column master key used for Always Encrypted functionality. If no value is specified, an empty sting is passed to the KeyStore file.

See also

Connection property descriptions on page 185

Always Encrypted on page 88

Additional properties

The following table summarizes additional connection properties.

Table 12: Additional properties

Property	Characteristic
AlwaysReportTriggerResults on page 199	Determines how the driver reports results that are generated by database triggers (procedures that are stored in the database and executed, or fired, when a table is modified). The driver does not report trigger results if the statement is a single Insert, Update, Delete, Create, Alter, Drop, Grant, Revoke, or Deny statement. In addition, the only result that is returned is the update count that is generated by the statement that was executed (if errors do not occur). Although trigger results are ignored, any errors that are generated by the trigger are reported. Any warnings that are generated by the trigger are enqueued. If errors are reported, the update count is not reported.
ApplicationIntent on page 199	Specifies whether the driver connects to read-write databases or requests read-only routing to connect to read-only database replicas. Read-only routing only applies to connections in Microsoft SQL Server 2012 where AlwaysOn Availability Groups have been deployed. By default, the driver connects to a read-write node in the AlwaysOn environment.

Property	Characteristic
CatalogOptions on page 204	Determines which type of metadata information is included in result sets when an application calls DatabaseMetaData methods. By default, result sets do not contain synonyms.
CodePageOverride on page 206	The code page to be used by the driver to convert Character data. The specified code page overrides the default database code page or column collation. All Character data returned from or written to the database is converted using the specified code page. By default, the driver automatically determines which code page to use to convert Character data. Use this property only if you need to change the driver's default behavior.
DatabaseName on page 211	Specifies the name of the database or instance to which you want to connect.
InitializationString on page 225	Specifies one or multiple SQL commands to be executed by the driver after it has established the connection to the database and has performed all initialization for the connection. If the execution of a SQL command fails, the connection attempt also fails and the driver throws an exception indicating which SQL command or commands failed.
InsensitiveResultSetBufferSize on page 226	Determines the amount of memory used by the driver to cache insensitive result set data. The default is `2048`.
LongDataCacheSize on page 231	Determines whether the driver caches long data (images, pictures, long text, binary data, or XML data) in result sets. To improve performance, you can disable long data caching if your application retrieves columns in the order in which they are defined in the result set. By default, the driver caches long data in result sets in memory with a memory buffer of 2048 KB for caching result set data. If the size of the result set data exceeds available memory, the driver pages the result set data to disk.

Property	Characteristic
PacketSize on page 234	Determines the number of bytes for each database protocol packet that is transferred from the database server to the client machine (Microsoft SQL Server refers to this packet as a network packet). Adjusting the packet size can improve performance. The optimal value depends on the typical size of data that is inserted, updated, or returned by the application and the environment in which it is running. Typically, larger packet sizes work better for large amounts of data. For example, if an application regularly returns character values that are 10,000 characters in length, using a value of 32 (16 KB) typically results in improved performance. By default, the driver uses the maximum packet size that the database server accepts.
ResultSetMetaDataOptions on page 239	Determines whether the driver returns table name information in the ResultSet metadata for Select statements. By default, the driver does not perform additional processing to determine the correct table name for each column in the result set when the ResultSetMetaData.getTableName() method is called. The getTableName() method may return an empty string for each column in the result set.
SelectMethod on page 240	A hint to the driver that determines whether the driver requests a database cursor for Select statements. Performance and behavior of the driver are affected by this property, which is defined as a hint because the driver may not always be able to satisfy the requested method. By default, the database server sends the complete result set in a single response to the driver when responding to a query. A server-side database cursor is not created if the requested result set type is a forward-only result set. Typically, responses are not cached by the driver. Using this method, the driver must process the entire response to a query before another query is submitted. If another query is submitted (using a different statement on the same connection, for example), the driver caches the response to the first query before submitting the second query. Typically, the direct method performs better than the cursor method.
SnapshotSerializable on page 243	Allows your application to use Snapshot Isolation for connections. This property is useful for applications that have the Serializable isolation level set. Using the SnapshotSerializable property allows you to use Snapshot Isolation with no or minimum code changes. If you are developing a new application, you may find that using the constant TRANSACTION_SNAPSHOT is a better choice. By default, the application uses the Serializable isolation level when your application has the transaction isolation level set to Serializable.

Property	Characteristic
SpyAttributes on page 244	Enables DataDirect Spy to log detailed information about calls issued by the driver on behalf of the application. DataDirect Spy is not enabled by default.
SuppressConnectionWarnings on page 246	Determines whether the driver suppresses "changed database" and "changed language" warnings when connecting to the database server. By default, warnings are not suppressed.
TransactionMode on page 247	Controls how the driver delimits the start of a local transaction. By default, the driver uses implicit transaction mode. This means that the database, not the driver, automatically starts a transaction when a transactionable statement is executed. Typically, implicit transaction mode is more efficient than explicit transaction mode because the driver does not have to send commands to start a transaction and a transaction is not started until it is needed. When TRUNCATE TABLE statements are used with implicit transaction mode, the database may roll back the transaction if an error occurs. If this occurs, use the explicit value for this property.
TruncateFractionalSeconds on page 248	Determines whether the driver truncates timestamp values to three fractional seconds. For example, a value of the datetime2 data type can have a maximum of seven fractional seconds. By default, the driver truncates all timestamp values to three fractional seconds.
UseServerSideUpdatableCursors on page 251	Determines whether the driver uses server-side cursors when an updatable result set is requested. By default, the client-side updatable cursors are created when an updatable result set is requested.
XATransactionGroup on page 252	The transaction group ID that identifies any distributed transactions that are initiated by the connection. This ID can be used for distributed transaction cleanup purposes. You can use the XAResource.recover method to roll back any transactions left in an unprepared state. When you call XAResource.recover, any unprepared transactions that match the ID on the connection used to call XAResource.recover are rolled back.

See also

Connection property descriptions on page 185

Performance considerations

You can optimize your application's performance if you set the SQL Server driver connection properties as described in this section:

Always Encrypted: The following options related to the Always Encrypted feature affect performance:

- **ColumnEncryption**: Due to the overhead associated with encrypting and decrypting data, Always Encrypted functionality can adversely affect performance when enabled. If your application does not require access to encrypted columns, you can disable this property (`ColumnEncryption=Disabled`) for improved performance. Alternatively, if your application only needs to retrieve and decrypt columns, not update them, you can improve performance over the behavior of the `Enabled` setting by specifying a value of `ResultsetOnly` for this property. Note that when using this setting, queries containing parameters that affect encrypted columns will return an error.

- **AEKeyCacheTTL**: When Always Encrypted functionality is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`), you can determine how long, in seconds, column encryption keys are cached using the AEKeyCacheTTL property. Caching column encryption keys can provide performance gains by reducing the overhead associated with fetching and decrypting keys for the same data multiple times during a connection. Specifying larger values for this option increases the length of time that a column encryption key persists in the cache; therefore, improving performance in some scenarios. Alternatively, by specifying a value of `-1`, you can configure the driver to persist keys for the life of the connection. Note that column encryption keys are designed to be deleted from the cache as a security measure and should not be stored for long periods of time.

ApplicationIntent: You can shift load away from the read-write nodes of your database cluster to read-only nodes by setting this connection property to `readOnly` and querying read-only database replicas when possible.

EnableBulkLoad: For batch inserts, the driver can use native bulk load protocols instead of the batch mechanism. Bulk load bypasses the data parsing usually done by the database, providing an additional performance gain over batch operations. Set this property to `true` to allow existing applications with batch inserts to take advantage of bulk load without requiring changes to the code.

EncryptionMethod: Data encryption may adversely affect performance because of the additional overhead (mainly CPU usage) required to encrypt and decrypt data.

InsensitiveResultSetBufferSize: To improve performance, result set data can be cached instead of written to disk. If the size of the result set data is greater than the size allocated for the cache, the driver writes the result set to disk. The maximum cache size setting is 2 GB.

LongDataCacheSize: To improve performance when your application retrieves images, pictures, long text, binary data, or XML data, you can disable caching for long data on the client if your application retrieves long data column values in the order they are defined in the result set. If your application retrieves long data column values out or order, long data values must be cached.

MaxPooledStatements: To improve performance, the driver's own internal prepared statement pooling should be enabled when the driver does not run from within an application server or from within another application that does not provide its own prepared statement pooling. When the driver's internal prepared statement pooling is enabled, the driver caches a certain number of prepared statements created by an application. For example, if the MaxPooledStatements property is set to `20`, the driver caches the last 20 prepared statements created by the application. If the value set for this property is greater than the number of prepared statements used by the application, all prepared statements are cached.

See Designing JDBC applications for performance optimization on page 345 for more information about using prepared statement pooling to optimize performance.

PacketSize: Typically, it is optimal for the client to use the maximum packet size that the server allows. This reduces the total number of round trips required to return data to the client, thus improving performance. Therefore, performance can be improved if this property is set to the maximum packet size of the database server.

ResultSetMetaDataOptions: The driver's performance may be adversely affected if you set this option to `1`. If set to `1` and the ResultSetMetaData.getTableName method is called, the driver performs emulations which take additional processing.

SelectMethod: In most cases, using server-side database cursors impacts performance negatively. However, if the following four variables are true in your application, the best setting for this property is cursor, which means use server-side database cursors:

- Your application contains queries that retrieve large amounts of data.
- Your application executes a SQL statement before processing or closing a previous large result set and does this multiple times.
- Large result sets use forward-only cursors.

SnapshotSerializable: Snapshot Isolation provides transaction-level read consistency and an optimistic approach to data modifications by not acquiring locks on data until data is to be modified. This Microsoft SQL Server feature can be useful if you want to consistently return the same result set even if another transaction has changed the data and 1) your application executes many read operations or 2) your application has long running transactions that could potentially block users from reading data. This feature has the potential to eliminate data contention between read operations and update operations. When this connection property is set to `true` (thereby, you are using Snapshot Isolation), performance is improved due to increased concurrency.

UseServerSideUpdatableCursors: In most cases, using server-side updatable cursors improves performance. However, this type of cursor cannot be used with insensitive result sets or with sensitive results sets that are not generated from a database table that contains a primary key.

See Server-side updatable cursors on page 110 for more information about using server-side updatable cursors.

See also

ApplicationIntent on page 199
ColumnEncryption on page 207
EnableBulkLoad on page 216
EncryptionMethod on page 218
InsensitiveResultSetBufferSize on page 226
LongDataCacheSize on page 231
MaxPooledStatements on page 232
Designing JDBC applications for performance optimization on page 345
PacketSize on page 234
ResultSetMetaDataOptions on page 239
SelectMethod on page 240
SnapshotSerializable on page 243
UseServerSideUpdatableCursors on page 251
Server-side updatable cursors on page 110

Connecting to named instances

Microsoft SQL Server supports multiple instances of a database running concurrently on the same server. An instance is identified by an instance name. To connect to a named instance using a connection URL, use the following URL format.

```
jdbc:datadirect:sqlserver://server_name\\instance_name
```

Note: The first backslash character (\) in `\\instance_name` is an escape character.

where:

server_name

is the IP address or hostname of the server.

instance_name

is the name of the instance to which you want to connect on the server.

For example, the following connection URL connects to an instance named instance1 on server1.

```
jdbc:datadirect:sqlserver://server1\\instance1;User=test;Password=secret
```

To connect to a named instance using a data source, you specify the ServerName property. For example:

```
SQLServerDataSource mds = new SQLServerDataSource();
mds.setDescription("My SQLServerDataSource");
mds.setServerName("server1\\instance1");
mds.setDatabaseName("TEST");
mds.setUser("test");
mds.setPassword("secret");
```

Azure Synapse Analytics and Analytics Platform System

The driver transparently connects to Microsoft Azure Synapse Analytics and Microsoft Analytics Platform System; however, the following limitations to features and functionality apply.

- No support for unquoted identifiers. The driver always enforces ANSI rules regarding quotation marks for Azure Synapse Analytics and Analytics Platform System connections.
- No support for connection pooling reauthentication.
- No support for Data Definition Language (DDL) queries within transactions.
- No support for closing holdable cursors when a transaction is committed.
- No support for server-side cursors; therefore:

  - Scroll-sensitive result sets are not supported.
  - The UseServerSideUpdatableCursors property is set to false, and server-side cursors are not used.

- No support for XA connections.
- Support for isolation levels is limited to only the read uncommitted level. See "Isolation Levels" for more information.
- No support for the following SQL Server data types in either the Azure Synapse Analytics or Analytics Platform System.

decimal() identity	timestamp
image	tinyint identity
numeric() identity	ntext
smallint identity	xml
text	

- Support for scalar string functions is limited to the following functions.

ASCII	LEFT	RTRIM
CHAR	LTRIM	SOUNDEX
CONCAT	REPLACE	SPACE
DIFFERENCE	RIGHT	SUBSTRING

- Support for scalar numeric functions is limited to the following functions.

ABS	EXP	ROUND
ACOS	FLOOR	SIGN
ASIN	LOG	SIN
ATAN	LOG10	SQRT
CEILING	PI	TAN
COS	POWER	TRUNCATE
COT	RADIANS	
DEGREES	RAND (Azure Warehouse only)	

- Support for scalar date and time functions is limited to the following functions.

CURDATE	DAYOFWEEK	QUARTER
CURRENT_DATE	DAYOFYEAR	SECOND
CURRENT_TIME	HOUR	WEEK
CURTIME	MINUTE	YEAR
DAYNAME	MONTH	
DAYOFMONTH	MONTHNAME	

See also

Data types on page 17

Scalar functions on page 266

Authentication

Depending on the authentication mechanism used in your environment, additional steps may be required to configure authentication. The proceeding topics provide detailed instructions on configuring user/ID password, Azure Active Directory, Kerberos, and NTLM authentication.

See also

AuthenticationMethod on page 201

Configuring user ID/password authentication on page 79

Configuring Azure Active Directory authentication on page 80

Configuring the driver for Kerberos authentication on page 81

Configuring NTLM authentication on page 86

Configuring user ID/password authentication

Take the following steps to configure user ID/Password authentication.

1. Set the AuthenticationMethod property to `userIdPassword` or `auto`.
2. Set the User property to provide the user ID.
3. Set the Password property to provide the password.
4. Specify values for minimum required properties for establishing a connection.
    a) Set the ServerName property to specify either the IP address in IPv4 or IPv6 format, or the server name for your Azure server.
    b) Set the PortNumber property to specify the TCP port of the primary database server that is listening for connections to the database.

For example, the following is a connection string with only the required properties for making a connection using user ID/password authentication.

```
Connection conn = DriverManager.getConnection
          ("jdbc:datadirect:sqlserver://server1:1433;
          AuthenticationMethod=userIdPassword;User=test;
          Password=secret);
```

See also

AuthenticationMethod on page 201

User on page 250

Password on page 236

Configuring Azure Active Directory authentication

The driver supports Azure Active Directory authentication (Azure AD). Azure AD authentication is an alternative to SQL Server authentication that allows administrators to centrally manage user permissions to Azure.

Note: Azure Active Directory authentication requires Java SE 7 or higher.

Note: When using Azure AD authentication, the driver requires root CA certificates to establish an SSL connection to a database. The driver determines the location of the truststore containing the required certificates by using the default JRE `cacerts` file unless a different file has been specified by the `javax.net.ssl.trustStore` Java system property. The truststore location cannot be specified using the driver's Truststore property.

Take the following steps to configure the driver to use Azure AD authentication.

1. Set the AuthenticationMethod property to specify a value of `ActiveDirectoryPassword`.
2. Set the User property to specify your Active Directory username using the `userid@domain.com` format.
3. Set the Password property to specify your Active Directory password.
4. Specify values for minimum required properties for establishing a connection.
   a) Set the ServerName property to specify either the IP address in IPv4 or IPv6 format, or the server name for your Azure server. For example, `myserver.database.windows.net`.
   b) Set the PortNumber property to specify the TCP port of the primary database server that is listening for connections to the database.

For example, the following is a connection string with only the required options for making a connection using Azure AD authentication.

Note: If the HostNameInCertificate is not specified, the driver automatically uses the value of the ServerName from the URL as the value for validating the certificate.

```
Connection conn = DriverManager.getConnection
("jdbc:datadirect:sqlserver://your_server.database.windows:1433;
AuthenticationMethod=ActiveDirectoryPassword;
User=test@mydomain.com;Password=secret");
```

See also

AuthenticationMethod on page 201
HostNameInCertificate on page 224
User on page 250
Password on page 236
ServerName on page 241
PortNumber on page 236

Configuring the driver for Kerberos authentication

Your Kerberos environment should be fully configured before you configure the driver for Kerberos authentication. You should refer to your SQL Server documentation and Java documentation for instructions on configuring Kerberos. For a Windows Active Directory implementation, you should also consult your Windows documentation. For a non-Active Directory implementation (on a Windows or non-Windows operating system), you should consult MIT Kerberos documentation.

Important: A properly configured Kerberos environment must include a means of obtaining a Kerberos Ticket Granting Ticket (TGT). For a Windows Active Directory implementation, Active Directory automatically obtains the TGT. However, for a non-Active Directory implementation, the means of obtaining the TGT must be automated or handled manually.

Once your Kerberos environment has been configured, take the following steps to configure the driver.

1. Use one of the following methods to integrate the JAAS configuration file into your Kerberos environment. (See "The JAAS login configuration file" for details.)

   Note: The *`install_dir`*`/lib/JDBCDriverLogin.conf` file is the JAAS login configuration file installed with the driver. You can use this file or another file as your JAAS login configuration file.

   Note: Regardless of operating system, forward slashes must be used when designating the path of the JAAS login configuration file.

   Option 1. Specify a login configuration file directly in your application with the `java.security.auth.login.config` system property. For example:

   ```
   System.setProperty("java.security.auth.login.config","install_dir/lib/JDBCDriverLogin.conf");
   ```

   Option 2. Set up a default configuration. Modify the Java security properties file to indicate the URL of the login configuration file with the `login.config.url.`*n* property where *n* is an integer connoting separate, consecutive login configuration files. When more than one login configuration file is specified, then the files are read and concatenated into a single configuration.

   a) Open the Java security properties file. The security properties file is the `java.security` file in the `/jre/lib/security` directory of your Java installation.

   b) Find the line `# Default login configuration file` in the security properties file.

   c) Below the `# Default login configuration file` line, add the URL of the login configuration file as the value for a `login.config.url.`*n* property. For example:

   ```
   # Default login configuration file
   login.config.url.1=file:${user.home}/.java.login.config
   login.config.url.2=file:install_dir/lib/JDBCDriverLogin.conf
   ```

2. Ensure your JAAS login configuration file includes an entry with authentication technology that the driver can use to establish a Kerberos connection. (See "The JAAS login configuration file" for details.)

   Note: The JAAS login configuration file installed with the driver (*`install_dir`*`/lib/JDBCDriverLogin.conf`) includes a default entry with the name `JDBC_DRIVER_01`. This entry specifies the Kerberos authentication technology used with an Oracle JVM.

The following examples show that the authentication technology used in a Kerberos environment depends on your JVM.

Oracle JVM

```
JDBC_DRIVER_01 {
  com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;
};
```

IBM JVM

```
JDBC_DRIVER_01 {
  com.ibm.security.auth.module.Krb5LoginModule required useDefaultCcache=true;
};
```

3. Set the driver's AuthenticationMethod connection property to `auto` or `kerberos`. (See "AuthenticationMethod" for details.)

   Note: If your are configuring your environment for Kerberos Constrained Delegation (also known as impersonation), AuthenticationMethod must be set to `kerberos`.

4. Optionally, set the ServicePrincipalName connection property if the default value built by the driver does not match the service principal name registered with the KDC.

   By default, the driver builds the ServicePrincipalName by concatenating the service name `MSSQLSvc`, the fully qualified domain name (FQDN) as specified with the ServerName property, the port number as specified with the PortNumber property, and the default realm name as specified in the Kerberos configuration file (`krb5.conf`). If this value does not match the service principal name registered with the KDC, then the value of the service principal name registered with the KDC should be specified for the ServicePrincipalName property.

   The ServicePrincipalName takes the following form.

   ```
   Service_Name/Fully_Qualified_Domain_Name:Port_Number@REALM_NAME
   ```

   See "ServicePrincipalName" for details on the composition of the service principal name.

5. Optionally, set the LoginConfigName connection property if the name of the JAAS login configuration file entry is different from the driver default `JDBC_DRIVER_01`. (See "The JAAS login configuration file" and "LoginConfigName" for details.)

   `JDBC_DRIVER_01` is the default entry name for the JAAS login configuration file (`JDBCDriverLogin.conf`) installed with the driver. When configuring your Kerberos environment, your network or system administrator may have used a different entry name. Check with your administrator to verify the correct entry name.

6. Optionally, set the GSSCredential connection property for Kerberos constrained delegation (sometimes referred to as impersonation).

   Constrained delegation is a Kerberos mechanism that allows a client application to delegate authentication to a second service. See "Constrained delegation" for additional steps to configure your environment.

   AuthenticationMethod must be set to `kerberos` to use constrained delegation.

See also

Kerberos authentication requirements on page 83

The JAAS login configuration file on page 83

Constrained delegation on page 85

AuthenticationMethod on page 201
ServicePrincipalName on page 242
LoginConfigName on page 229
GSSCredential on page 223

Kerberos authentication requirements

Verify that your environment meets the requirements listed in the following table before you configure the driver for Kerberos authentication.

Note: For Windows Active Directory, the domain controller must administer both the database server and the client.

Table 13: Kerberos configuration requirements

Component	Requirements
Database server	No restrictions
Kerberos server	The Kerberos server is the machine where the user IDs for authentication are administered. The Kerberos server is also the location of the Kerberos key distribution center (KDC). Network authentication must be provided by one of the following methods. • Windows Active Directory on SQL Server 2005 or higher • MIT Kerberos 1.5 or higher
Client	Java SE 6 or higher must be installed.

See also

Configuring the driver for Kerberos authentication on page 81

The JAAS login configuration file

The Java Authentication and Authorization Service (JAAS) login configuration file contains one or more entries that specify authentication technologies to be used by applications. To establish Kerberos connections with the driver, the JAAS login configuration file must include an entry specifically for the driver. In addition, the login configuration file must be referenced either by setting the `java.security.auth.login.config` system property or by setting up a default configuration using the Java security properties file.

Setting up a default configuration

To set up a default configuration, you must modify the Java security properties file to indicate the URL of the login configuration file with the `login.config.url.`*n* property where *n* is an integer connoting separate, consecutive login configuration files. When more than one login configuration file is specified, then the files are read and concatenated into a single configuration. The following steps summarize how to modify the security properties file.

1. Open the Java security properties file. The security properties file is the `java.security` file in the `/jre/lib/security` directory of your Java installation.
2. Find the line `# Default login configuration file` in the security properties file.
3. Below the `# Default login configuration file` line, add the URL of the login configuration file as the value for a `login.config.url.`*n* property. For example:

```
# Default login configuration file
login.config.url.1=file:${user.home}/.java.login.config
login.config.url.2=file:install_dir/lib/JDBCDriverLogin.conf
```

JAAS login configuration file entry for the driver

You can create your own JAAS login configuration file, or you can use the `JDBCDriverLogin.conf` file installed in the `/lib` directory of the product installation directory. In either case, the login configuration file must include an entry that specifies the Kerberos authentication technology to be used by the driver.

JAAS login configuration file entries begin with an entry name followed by one or more LoginModule items. Each LoginModule item contains information that is passed to the LoginModule. A login configuration file entry takes the following form.

```
entry_name {
  login_module flag_value module_options
};
```

where:

entry_name

is the name of the login configuration file entry. The driver's LoginConfigName connection property can be used to specify the name of this entry. `JDBC_DRIVER_01` is the default entry name for the `JDBCDriverLogin.conf` file installed with the driver.

login_module

is the fully qualified class name of the authentication technology used with the driver.

flag_value

specifies whether the success of the module is `required`, `requisite`, `sufficient`, or `optional`.

module_options

specifies available options for the LoginModule. These options vary depending on the LoginModule being used.

The following examples show that the LoginModule used for a Kerberos implementation depends on your JVM.

Oracle JVM

```
JDBC_DRIVER_01 {
  com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;
};
```

IBM JVM

```
JDBC_DRIVER_01 {
  com.ibm.security.auth.module.Krb5LoginModule required useDefaultCcache=true;
};
```

Refer to Java Authentication and Authorization Service documentation for information about the JAAS login configuration file and implementing authentication technologies.

See also

Configuring the driver for Kerberos authentication on page 81

LoginConfigName on page 229

Constrained delegation

Constrained delegation is a Kerberos mechanism that allows a client application to delegate authentication to a second service. The client application informs the KDC that the second service is authorized to act on behalf of a specified Kerberos security principal, such as a user that has an Active Directory account. The second service can then delegate authentication to a database service principal name. (Refer to the Microsoft TechNet page About Kerberos constrained delegation for further details.)

To enable constrained delegation:

Important: Before you start, in the `[libdefaults]` section of the `krb5.conf` file, set the `forwardable` flag to `true`.

1. Authenticate the service principal and get a subject from the login context. The service principal needs a Kerberos granting ticket to be authenticated. You can use either a ticket cache or keytab file for the authentication step. The section you define in your JAAS login configuration file determines which method is used for authentication.
2. Call the impersonate method to generate the service ticket for the database user on behalf of the service principal identity.
3. Using the driver's GSSCredential property, specify the GSSCredential generated in the previous steps.
4. Call the driver's connect() method using the Properties object. The Properties object must contain the GSSCredential property and any additional properties needed to establish a connection to the database.

The following example code shows how a GSS credential object can be integrated into a client application to support constrained delegation.

```
   Subject serviceSubject;
   GSSCredential creds;

//Authenticate the service principal and get a subject from the login context.
   LoginContext lc = new LoginContext("entry_from_your_jaas_config");
   lc.login();
   serviceSubject = lc.getSubject();

//Call the impersonate method to generate the service ticket for database user
//on behalf of the service principal identity.
   try {
    creds = Subject.doAs(serviceSubject, new
            PrivilegedExceptionAction<GSSCredential>() {
              public GSSCredential run() throws Exception {
                GSSManager manager = GSSManager.getInstance();
                if (serviceCredentials == null) {
             serviceCredentials =
                        manager.createCredential(GSSCredential.INITIATE_ONLY);
               }
               GSSName other = manager.createName("userToImpersonate",
                               GSSName.NT_USER_NAME);
                               return
               ((ExtendedGSSCredential)serviceCredentials).impersonate(other);
```

```
                }
            });
    } catch (PrivilegedActionException pae) {
       throw pae.getException();
    }

    final Properties sqlserverProperties;

    sqlserverProperties = new Properties();
 // Set the driver's GSSCredential property and the rest of the database
 // connection properties
    sqlserverProperties.put("GSSCredential", creds);
    sqlserverProperties.put("ServerName", "yourServer");
    sqlserverProperties.put("portNumber", "1433");
    sqlserverProperties.put("authenticationMethod", "Kerberos");
    sqlserverProperties.put("databaseName", "yourDatabase");


    Connection con = DriverManager.getConnection("jdbc:datadirect:sqlserver:",
                     sqlserverProperties);
    DatabaseMetaData  dbmd = con.getMetaData();

    System.out.println( "\nConnected with " + dbmd.getDriverName() + "\n"
                          + " to " + dbmd.getDatabaseProductName() + "\n"
                          + " " + dbmd.getDatabaseProductVersion() + "\n"
                          + " " + dbmd.getDriverVersion() + "\n");
```

Configuring NTLM authentication

If your environment meets the appropriate requirements, you can configure NTLM authentication by specifying user credentials on either a Windows or UNIX/Linux operating system.

NTLM authentication requirements

Verify that your environment meets the requirements listed in the following table before you configure your environment for NTLM authentication.

Note: The domain controller must administer both the database server and the client.

Table 14: NTLM authentication requirements

Component	Requirements
Database server	The server must be running Microsoft SQL Server 2005 or higher.
Domain controller	Network authentication must be provided by NTLM on Microsoft SQL Server 2005 or higher.
Client	Java SE 6 or higher must be installed.

Configuring NTLM authentication by specifying user credentials

You can configure the driver for NTLM authentication with the specification of user credentials.

To configure the driver:

1. Set the AuthenticationMethod property to either of the following values.
   - `ntlmjava` to use NTLMv1 or NTLMv2 depending on the size of the password. NTLMv1 is used if the password is 14 bytes or less; NTLMv2 is used if the password is more than 14 bytes.
   - `ntlm2java` to use NTLMv2 protocols to connect to a server that is restricted to using NTLMv2 authentication.

   Note: See AuthenticationMethod on page 201 for more information on setting the AuthenticationMethod property.

2. Set the Domain property to provide the name of the domain server that administers the database.

   Note: Alternatively, you can set the domain server name using the User property.

3. Set the User property to provide the user ID.
4. Set the Password property to provide the password.
5. If using NTLM authentication with a Security Manager on a Java Platform, security permissions must be granted to allow the driver to establish connections. See Permissions for establishing connections on page 49 for an example.

Data encryption

The driver supports SSL encryption. Depending on your Microsoft SQL Server configuration, you can choose to encrypt all data, including the login request, or encrypt the login request only. Encrypting login requests, but not data, is useful for the following scenarios:

- When your application needs security, but cannot afford to pay the performance penalty for encrypting data transferred between the driver and server.
- When the server is not configured for SSL, but your application still requires a minimum degree of security.

Note: When SSL is enabled, the driver communicates with database protocol packets set by the server's default packet size. Any value set by the PacketSize property is ignored.

Using SSL with Microsoft SQL Server

If your Microsoft SQL Server database server has been configured with an SSL certificate signed by a trusted CA, the server can be configured so that SSL encryption is either optional or required. When required, connections from clients that do support SSL encryption fail.

Although a signed trusted SSL certificate is recommended for the best degree of security, Microsoft SQL Server can provide limited security protection even if an SSL certificate has not been configured on the server. If a trusted certificate is not installed, the server will use a self-signed certificate to encrypt the login request, but not the data.

The following table shows how the different EncryptionMethod property values behave with different Microsoft SQL Server configurations.

Table 15: EncryptionMethod property values and Microsoft SQL Server configurations

Value	No SSL Certificate	SSL Certificate	
		SSL Optional	SSL Required
noEncryption	Login request and data are not encrypted.	Login request and data are not encrypted.	Connection attempt fails.
SSL	Connection attempt fails.	Login request and data are encrypted.	Login request and data are encrypted.
requestSSL	Login request and data are not encrypted.	Login request and data are encrypted.	Login request and data are encrypted.
loginSSL	Login request is encrypted, but data is not encrypted	Login request is encrypted, but data is not encrypted.	Login request and data are encrypted.

Configuring SSL encryption

Take the following steps to configure SSL encryption.

1. Choose the type of encryption for your application:
   - If you want the driver to encrypt all data, including the login request, set the EncryptionMethod property to SSL or requestSSL.
   - If you want the driver to encrypt only the login request, set the EncryptionMethod property to loginSSL.
2. Use the CryptoProtocolVersion property to specify acceptable cryptographic protocol versions (for example, TLSv1.2) supported by your server. (Only applies when the EncryptionMethod property is set to SSL.)
3. Specify the location and password of the truststore file used for SSL server authentication. Either set the TrustStore and TrustStore properties or their corresponding Java system properties (javax.net.ssl.trustStore and javax.net.ssl.trustStorePassword, respectively).
4. To validate certificates sent by the database server, set the ValidateServerCertificate property to `true`.
5. Optionally, set the HostNameInCertificate property to a host name to be used to validate the certificate. The HostNameInCertificate property provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.

Always Encrypted

Note: Always Encrypted support requires the driver to run on a Java Virtual Machine (JVM) that is Java SE 8 or higher.

Microsoft supports the Always Encrypted feature for Azure SQL Databases and SQL Server databases beginning with SQL Server 2016. Always Encrypted functionality provides improved security by storing sensitive data on the server in an encrypted state. When sensitive data is queried by the application, the driver transparently decrypts data from encrypted columns and returns them to the application. Conversely, when encrypted data needs to be passed to the server, the driver transparently encrypts parameter values before sending them for storage. As a result, sensitive data is visible only to authorized users of the application, not by those who maintain the data. This reduces exposure to a number of potential vulnerabilities, including server-side security breaches and access by database administrators who would not otherwise be authorized to view the data.

Always Encrypted functionality employs a column master key and column encryption key to process encrypted data. The column encryption key is used to encrypt sensitive data in an encrypted column, while the column master key is used to encrypt column encryption keys. To prevent server-side access to encrypted data, the column master key is stored in a keystore that is separate from the server that contains the data. When Always Encrypted is enabled, the driver uses the steps described in this section to retrieve keys and negotiate the decryption of encrypted data.

By design, data stored in encrypted columns cannot be accessed without first being retrieved and decrypted by the driver. Although this restriction improves security, it also prevents literal values within these columns to be referenced when issuing a statement. As a result, statements can only reference encrypted columns using parameter markers.

When the application executes a parameterized statement with Always Encrypted enabled:

1. The driver executes a stored procedure to determine from the server whether there are any encrypted columns referenced by the statement.
2. If any columns are encrypted, the driver retrieves the encrypted column metadata, encrypted column encryption key, and the location of the column master key for each parameter to be encrypted.
3. The driver retrieves the column master key from the keystore; then, uses it to decrypt the column encryption key. After decryption, the column encryption key is cached in a decrypted state for subsequent operations or discarded. See the "Using Keystore Providers" section for more information.

   Note: You can dictate the length of time column encryption keys are persisted in the cache using the AEKeyCacheTTL property. See "Caching column encryption keys" for more information.

4. The driver encrypts the parameters using the decrypted column encryption key.
5. The driver sends the statement with encrypted values to the server for processing.
6. If applicable, the server returns the result set, along with the encryption algorithm information, encrypted column encryption key, and location of the column master keys.
7. If the column encryption key is not cached, the driver retrieves column master key from the keystore; then uses it to decrypt the column encryption key.
8. The driver decrypts the result set using the column encryption key and returns it to the application.

See "Enabling Always Encrypted" for information on configuring Always Encrypted with the driver.

See also

Using keystore providers on page 90

Caching column encryption keys on page 91

Enabling Always Encrypted on page 90

Enabling Always Encrypted

You can configure Always Encrypted behavior for the driver by specifying the following values for the ColumnEncryption connection property:

- If set to `Enabled`, the driver fully supports Always Encrypted functionality. The driver transparently decrypts result sets and returns them to the application. In addition, the driver transparently encrypts parameter values that are associated with encrypted columns.
- If set to `ResultsetOnly`, only decryption is enabled. The driver transparently decrypts result sets and returns them to the application. Queries containing parameters that affect encrypted columns will return an error.
- If set to `Disabled` (the default), Always Encrypted functionality is disabled. The driver does not attempt to decrypt data from encrypted columns and returns the data as binary-formatted cipher text. However, statements containing parameters that reference encrypted columns are not supported and will return an error.

By default, Always Encrypted is disabled (`ColumnEncryption=Disabled`). For more information on configuring the ColumnEncryption property, see "ColumnEncryption."

When Always Encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`), the driver must be configured to use a keystore provider; otherwise, an error will be returned. See "Using keystore providers" for details.

See also

Using keystore providers on page 90

ColumnEncryption on page 207

Using keystore providers

Keystore providers securely store the column master keys used for decrypting the column encryption keys employed by Always Encrypted functionality. The driver requires that a keystore provider be used when always encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`). The following section describes how to configure the driver to use the supported keystore providers.

Azure Key Vault

The Azure Key Vault is a certificate repository hosted on Azure platforms. Using Azure Key Vault offers several advantages, including the ability for applications on any platform to access keys. In addition, since the keys are centrally stored, they do not need to be copied to and cached on a local machine. However, unless your application is running on Azure, calls to the key vault must be made over a WAN, which can negatively impact performance. To use Azure Key Vault, values for the following properties must be provided:

- AEKeystorePrincipalId: Specifies the principal ID for the Azure Key Vault. The principal ID is the Application ID created during Azure App Registration. See "KeyStorePrincipalId" for a detailed description.
- AEKeystoreClientSecret: Specifies the Client Secret used to access the Azure Key Vault. See "AEKeystoreClientSecret" for a detailed description.

Java KeyStore

Java Keystore is a repository of certificates for Java platforms. Similar to Azure Key Vault, the column master key is stored centrally, which means keys do not need to be cached on local machines. However, unlike Azure Key Vault, access to the Java Keystore is limited to applications running on Java platforms. To use Java Keystore, values for the following properties must be provided:

- AEKeystoreLocation: Specifies the absolute path to the Java KeyStore file. See "AEKeystoreLocation" for details.

- AEKeystoreSecret: Specifies the password used to access the Java KeyStore file. See "AEKeystoreSecret" for details.

See also

ColumnEncryption on page 207
AEKeystorePrincipalId on page 196
AEKeystoreClientSecret on page 194
AEKeystoreLocation on page 195
AEKeystoreSecret on page 197

Caching column encryption keys

Caching column encryption keys improves performance by reducing the overhead associated with fetching and decrypting the keys for the same data multiple times. For security purposes, the driver empties keys from the cache when a connection closes; however, for applications that remain connected for long periods of time, you may want to delete the keys before the connection ends. You can determine the length of time the driver caches keys by specifying the following values for the AEKeyCacheTTL property:

- If set to `-1`, the driver caches column encryption keys for the life of the connection. The keys are deleted when the connection is closed or added to the connection pool.
- If set to `0`, the driver does not cache column encryption keys.
- If set to *x*, the driver caches column encryption keys for the specified number of seconds before deleting them. The timer starts for a key when it is first accessed and added to the cache. The timer does not reset if you access it after it has been added to the cache. The keys are deleted when the timer expires, or the connection is closed or added to the connection pool.

By default, the driver caches keys for `7200` seconds. See "AEKeyCacheTTL" for details.

See also

AEKeyCacheTTL on page 193

Enabling parameter metadata discovery

The driver initiates the processing of encrypted parameters by passing the T-SQL for a prepared statement to the sp_describe_parameter_encryption system stored procedure, which is then used to return metadata for the encrypted parameters in statement. To maintain data type integrity, the server requires that the T-SQL data type, length, precision and scale values specified by the driver match those of the underlying native data type referenced by the parameters in the T-SQL statement. However, since some of the native data types do not have a one-to-one mapping to a JDBC type[8], the driver may not be able to communicate the T-SQL type to the server when calling this procedure. When this occurs, the procedure will fail to execute with an `Operand type clash` error.

To correct this issue, functionality was added to allow the driver to automatically discover the underlying type and adjust the T-SQL passed to the the sp_describe_parameter_encryption procedure. This behavior is disabled by default when Always Encrypted functionality is enabled (`ColumnEncryption=Enabled|ResultsetOnly`). To enable data type discovery, configure the following connection properties with the values provided:

```
DescribeInputParameters=DESCRIBEALL
DescribeOutputParameters=DESCRIBEALL
```

[8] For example., because there is no SQL_MONEY type for JDBC, the native Money and Decimal types both map to SQL_DECIMAL

When these values are specified, the driver makes an extra call to the server to retrieve accurate metadata to pass to the sp_describe_column_encryption procedure, which results in the gathering of encryption metadata and allows for encryption and decryption to succeed.

See also

DescribeInputParameters on page 214

DescribeOutputParameters on page 215

Connection string example

The following connection strings configure the driver to use Always Encrypted with the minimum required properties.

For connections using Azure Key Vault:

```
"jdbc:datadirect:sqlserver://MyServer:1433;
      AEKeystorePrincipalId=789f8b4c-7a4a-445d-6oe9-7bec14625645;
      AEKeyStoreClientSecret=ABcdEFg/hiJkLmNOPqR01stUvWxyzYx2wvUTsrQpO=;
      ColumnEncryption=Enabled;"
```

For connections Java KeyStore:

```
"jdbc:datadirect:sqlserver://MyServer:1433;
      AEKeystoreLocation=/usr/java/jre/lib/security/cacerts;AEKeystoreSecret=secret;
      ColumnEncryption=Enabled;"
```

See also

AEKeystorePrincipalId on page 196

AEKeystoreClientSecret on page 194

ColumnEncryption on page 207

AEKeystoreLocation on page 195

AEKeystoreSecret on page 197

Always Encrypted properties on page 69

Using failover

The following levels of failover protection are supported to ensure continuous, uninterrupted access to data.

- Connection failover on page 96 provides failover protection for new connections only. The driver fails over new connections to an alternate, or backup, database server if the primary database server is unavailable, for example, because of a hardware failure or traffic overload. If a connection to the database is lost, or dropped, the driver does not fail over the connection. This failover method is the default.
- Extended connection failover on page 97 provides failover protection for new connections and lost database connections. If a connection to the database is lost, the driver fails over the connection to an alternate server, preserving the state of the connection at the time it was lost, but not any work in progress.
- Select connection failover on page 98 provides failover protection for new connections and lost database connections. In addition, it provides protection for Select statements that have work in progress. If a connection to the database is lost, the driver fails over the connection to an alternate server, preserving the state of the connection at the time it was lost and preserving the state of any work being performed by Select statements.

The method you choose depends on how failure-tolerant your application is. For example, if a communication failure occurs while processing, can your application handle the recovery of transactions and restart them?

When using either extended connection failover mode or select connection failover mode, your application needs the ability to recover and restart transactions. The advantage of select mode is that it preserves the state of any work that was being performed by the Select statement at the time of connection loss. If your application had been iterating through results at the time of the failure, when the connection is reestablished, the driver can reposition on the same row where it stopped so that the application does not have to undo all of its previous result processing. For example, if your application was paging through a list of items on a Web page when a failover occurred, the next page operation would be seamless instead of starting from the beginning. Performance, however, is a factor in selecting a failover mode. Select mode incurs additional overhead when tracking which rows the application has already processed.

You can specify which failover method you want to use by setting the FailoverMode connection property. Regardless of the failover method you choose, you must configure one or multiple alternate servers using the AlternateServers connection property.

See also

Failover properties on page 61

Always On Availability Groups on page 100

Configuring failover on page 93

Connection failover on page 96

Extended connection failover on page 97

Select connection failover on page 98

Configuring failover with Microsoft Cluster Server on page 99

Using client load balancing on page 99

Using connection retry on page 100

Configuring failover

Take the following steps to configure failover.

1. Specify the primary and alternate servers:

   - Specify your primary server using a connection URL or data source.
   - Specify one or multiple alternate servers by setting the AlternateServers property.

   See Specifying primary and alternate servers on page 94.

   Note: To turn off failover, do not specify a value for the AlternateServers property.

   Note: If using failover with Microsoft Cluster Server (MSCS), which determines the alternate server for failover instead of the driver, any alternate server specified must be the same as the primary server. For example:

   ```
   jdbc:datadirect:sqlserver://server1:1433;
     DatabaseName=TEST;User=test;Password=secret;
     AlternateServers=(server1:1433;DatabaseName=TEST)
   ```

2. Choose a failover method by setting the FailoverMode connection property. The default method is connection failover (FailoverMode=`connect`).
3. If FailoverMode=`extended` or FailoverMode=`select`, set the FailoverGranularity property to specify how you want the driver to behave if exceptions occur while trying to reestablish a lost connection. The default behavior of the driver is to continue with the failover process and post any exceptions on the statement on which they occur (FailoverGranularity=`nonAtomic`).

4. Optionally, configure the connection retry feature. See Specifying connection retry on page 96.
5. Optionally, set the FailoverPreconnect property if you want the driver to establish a connection with the primary and an alternate server at the same time. The default behavior is to connect to an alternate server only when failover is caused by an unsuccessful connection attempt or a lost connection (FailoverPreconnect=`false`).

Specifying primary and alternate servers

Connection information for primary and alternate servers can be specified using either one of the following methods:

- Connection URL through the JDBC Driver Manager
- JDBC data source

For example, the following connection URL for the SQL Server driver specifies connection information for the primary and alternate servers using a connection URL:

```
jdbc:datadirect:sqlserver://server1:1433;DatabaseName=TEST;User=test;
Password=secret;AlternateServers=(server2:1433;DatabaseName=TEST2,
server3:1433;DatabaseName=TEST3)
```

In this example:

```
...server1:1433;DatabaseName=TEST...
```

is the part of the connection URL that specifies connection information for the primary server. Alternate servers are specified using the AlternateServers property. For example:

```
...;AlternateServers=(server2:1433;DatabaseName=TEST2,server3:1433;
DatabaseName=TEST3)
```

Similarly, the same connection information for the primary and alternate servers specified using a JDBC data source would look like this:

```
SQLServerDataSource mds = new SQLServerDataSource();
mds.setDescription("My SQLServerDataSource");
mds.setServerName("server1");
mds.setPortNumber(1433);
mds.setDatabaseName("TEST");
mds.setUser("test");
mds.setPassword("secret");
mds.setAlternateServers("server2:1433;DatabaseName=TEST2, server3:1433;
   DatabaseName=TEST3")
```

In this example, connection information for the primary server is specified using the ServerName, PortNumber, and DatabaseName properties. Connection information for alternate servers is specified using the AlternateServers property.

The SQL Server driver also allows you to specify connections to named instances, multiple instances of a Microsoft SQL Server database running concurrently on the same server. If specifying named instances for the primary and alternate servers, the connection URL would look like this:

```
jdbc:datadirect:sqlserver://server1\\instance1;User=test;Password=secret;
AlternateServers=(server2\\instance2:1433;DatabaseName=TEST2,
server3\\instance3:1433;DatabaseName=TEST3)
```

Similarly, the same connection information to named instances for the primary and alternate servers specified using a JDBC data source would look like this:

```
SQLServerDataSource mds = new SQLServerDataSource();
mds.setDescription("My SQLServerDataSource");
mds.setServerName("server1\\instance1");
mds.setPortNumber(1433);
mds.setDatabaseName("TEST");
mds.setUser("test");
mds.setPassword("secret");
mds.setAlternateServers("server2\\instance2:1433;DatabaseName=
   TEST2,server3\\instance3:1433;DatabaseName=TEST3")
```

To connect to a named instance using a data source, you specify the named instance on the primary server using the ServerName property.

See Connecting to named instances on page 77 for more information about connecting to named instances on Microsoft SQL Server.

The value of the AlternateServers property is a string that has the format:

```
(servername1[:port1][;property=value][,servername2[:port2]
[;property=value]]...)
```

or, if connecting to named instances:

```
(servername1\\instance1[;property=value][,servername2\\instance2
[;property=value]]
```

where:

servername1

is the IP address or server name of the first alternate database server, *servername2* is the IP address or server name of the second alternate database server, and so on. The IP address or server name is required for each alternate server entry.

instance1

is the named instance on the first alternate database server, *servername2* is the named instance on the second alternate database server, and so on. If connecting to named instances, the named instance is required for each alternate server entry.

port1

is the port number on which the first alternate database server is listening, *port2* is the port number on which the second alternate database server is listening, and so on. The port number is optional for each alternate server entry. If unspecified, the port number specified for the primary server is used. If a port number is unspecified for the primary server, a default port number of `1433` is used.

property=*value*

is the DatabaseName connection property. This property is optional for each alternate server entry. For example:

```
jdbc:datadirect:sqlserver://server1:1433;DatabaseName=TEST;User=test;
Password=secret;AlternateServers=(server2:1433;DatabaseName=TEST2,
server3:1433;DatabaseName=TEST3)
```

or, if connecting to named instances:

```
jdbc:datadirect:sqlserver://server1\\instance1:1433;DatabaseName=TEST;
User=test;Password=secret;AlternateServers=(server2\\instance2:1433;
DatabaseName=TEST2,server3\\instance3:1433;DatabaseName=TEST3)
```

If you do not specify the DatabaseName connection property in an alternate server entry, the connection to that alternate server uses the property specified in the URL for the primary server. For example, if you specify `DatabaseName=TEST` for the primary server, but do not specify a database name in the alternate server entry as shown in the following URL, the driver tries to connect to the TEST database on the alternate server:

```
jdbc:datadirect:sqlserver://server1:1433;DatabaseName=TEST;User=test;
Password=secret;AlternateServers=(server2:1433,server3:1433)
```

Specifying connection retry

Connection retry allows the SQL Server driver to retry connections to the primary database server, and if specified, alternate servers until a successful connection is established. You use the ConnectionRetryCount and ConnectionRetryDelay properties to enable and control how connection retry works. For example:

```
jdbc:datadirect:sqlserver://server1:1433;DatabaseName=TEST;User=test;
Password=secret;AlternateServers=(server2:1433;DatabaseName=TEST2,
server3:1433;DatabaseName=TEST3);ConnectionRetryCount=2;
ConnectionRetryDelay=5
```

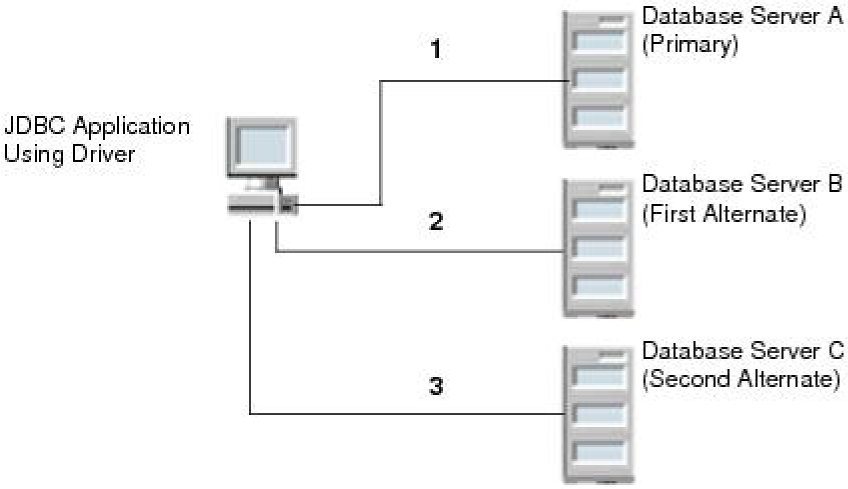
In this example, if a successful connection is not established on the SQL Server driver’s first pass through the list of database servers (primary and alternate), the driver retries the list of servers in the same sequence twice (`ConnectionRetryCount=2`). Because the connection retry delay has been set to five seconds (`ConnectionRetryDelay=5`), the driver waits five seconds between retry passes.

Connection failover

Connection failover allows an application to connect to an alternate, or backup, database server if the primary database server is unavailable, for example, because of a hardware failure or traffic overload. Connection failover provides failover protection for new connections only and does not provide protection for lost connections to the database, nor does it preserve states for transactions or queries.

You can customize the drivers for connection failover by configuring a list of alternate database servers that are tried if the primary server is not accepting connections. Connection attempts continue until a connection is successfully established or until all the alternate database servers have been tried the specified number of times.

For example, suppose you have the environment with multiple database servers as shown in the following figure. Database Server A is designated as the primary database server, Database Server B is the first alternate server, and Database Server C is the second alternate server.



First, the application attempts to connect to the primary database server, Database Server A (**1**). If connection failover is enabled and Database Server A fails to accept the connection, the application attempts to connect to Database Server B (**2**). If that connection attempt also fails, the application attempts to connect to Database Server C (**3**).

In this scenario, it is probable that at least one connection attempt would succeed, but if no connection attempt succeeds, the driver can retry each alternate database server (primary and alternate) for a specified number of attempts. You can specify the number of attempts that are made through the *connection retry* feature. You can also specify the number of seconds of delay, if any, between attempts through the *connection delay* feature. See Using connection retry on page 100 for more information about connection retry.

A driver fails over to the next alternate database server only if a successful connection cannot be established with the current alternate server. If the driver successfully establishes communication with a database server and the connection request is rejected by the database server because, for example, the login information is invalid, then the driver generates an exception.

Extended connection failover

Extended connection failover provides failover protection for the following types of connections:

- New connections (in the same way as described in Connection failover on page 96)
- Lost connections

When a connection to the database is lost, the driver fails over the connection to an alternate server, restoring the same state of the connection at the time it was lost. For example, when reestablishing a lost connection on the alternate database server, the driver performs the following actions:

- Restores the connection using the same connection properties specified by the lost connection
- Reallocates statement handles and attributes
- Logs in the user to the database with the same user credentials
- Restores any prepared statements associated with the connection
- Restores manual commit mode if the connection was in manual commit mode at the time of the failover

The driver does not preserve work in progress. For example, if the database server experienced a hardware failure while processing a query, partial rows processed by the database and returned to the client would be lost.

You can choose how you want the driver to behave if exceptions occur during failover by setting the FailoverGranularity connection property. If an exception occurs while the driver is reestablishing a lost connection, the driver can react in either of the following ways:

- It can fail the entire failover process. The driver stops trying to connect to an alternative server and returns an exception indicating that the connection was lost.
- It can proceed with the failover process as far as it is able. For example, suppose an exception occurred while reestablishing the connection because the driver was unable to log the user into the database. In this case, you may want the driver to notify your application of the exception and proceed with the failover process.

During the failover process, your application may experience a short pause while the driver establishes a new connection or reestablishes a lost connection on an alternate server. If your application is time-sensitive (a real-time customer order application, for example) and cannot absorb this wait, you can set the FailoverPreconnect property to `true`. Setting the FailoverPreconnect property to `true` instructs the driver to establish connections to the primary server and an alternate server at the same time. Your application uses the first connection that is successfully established. As a bonus, if this connection to the database is lost at a later time, the driver saves time in reestablishing the connection on the server it fails over to because it can use the spare connection in its failover process.

Select connection failover

Select connection failover provides failover protection for the following types of connections:

- New connections (in the same way as described in Connection failover on page 96)
- Lost connections (in the same way as described in Extended connection failover on page 97)

In addition, the driver can recover work in progress because it keeps track of the last Select statement the application executed on each Statement handle, including how many rows were fetched to the client. For example, if the database had only processed 500 of 1,000 rows requested by a Select statement when the connection was lost, the driver would reestablish the connection to an alternate server, re-execute the Select statement, and position the cursor on the next row so that the driver can continue fetching the balance of rows as if nothing had happened.

Performance, however, is a factor when considering whether to use Select mode. Select mode incurs additional overhead when tracking what rows the application has already processed.

Note: The driver only recovers work requested by Select statements. You must explicitly restart the following types of statements after a failover occurs:

- Insert, Update, or Delete statements
- Statements that modify the connection state, for example, SET or ALTER SESSION statements
- Objects stored in a temporary tablespace or global temporary table
- Partially executed stored procedures and batch statements

When in manual transaction mode, no statements are rerun if any of the operations in the transaction were Insert, Update, or Delete. This is true even if the statement in process at the time of failover was a Select statement.

By default, the driver verifies that the rows that are restored match the rows that were originally fetched and, if they do not match, generates an exception warning your application that the Select statement must be reissued. By setting the FailoverGranularity connection property, you can configure the driver to fail the entire failover process if the rows do not match.

Configuring failover with Microsoft Cluster Server

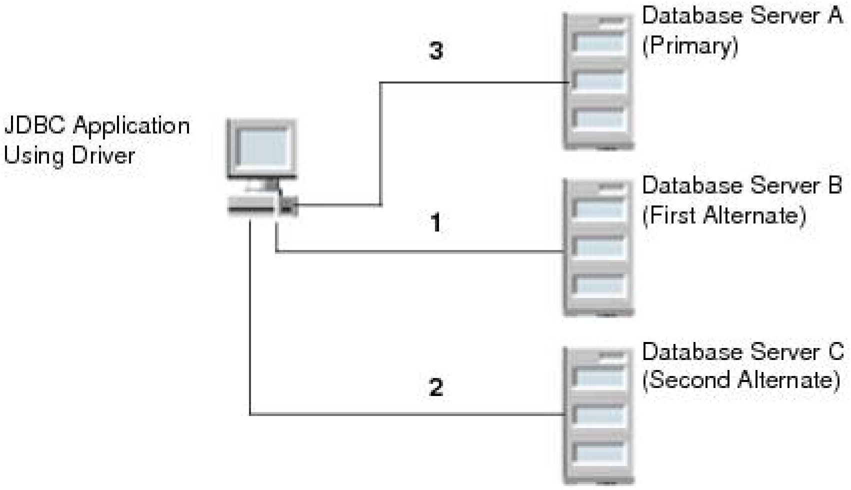
Microsoft SQL Server provides Microsoft Cluster Server (MSCS), an advanced database replication technology. The failover functionality provided by the driver does not require this technology, but can work with MSCS to provide comprehensive failover protection. If using failover with MSCS, which determines the alternate server for failover instead of the driver, any alternate server specified must be the same as the primary server. For example:

```
jdbc:datadirect:sqlserver://server1:1433;DatabaseName=TEST;
  User=test;Password=secret;AlternateServers=(server1:1433;
  DatabaseName=TEST)
```

In addition, alternate servers must mirror data on the primary server or be part of a configuration where multiple database nodes share the same physical data.

Using client load balancing

Client load balancing helps distribute new connections in your environment so that no one server is overwhelmed with connection requests. When client load balancing is enabled, the order in which primary and alternate database servers are tried is random. For example, suppose that client load balancing is enabled as shown in the following figure.



First, Database Server B is tried (**1**). Then, Database Server C may be tried (**2**), followed by a connection attempt to Database Server A (**3**). In contrast, if client load balancing were not enabled in this scenario, each database server would be tried in sequential order, primary server first, then each alternate server based on its entry order in the alternate servers list.

Client load balancing is controlled by the LoadBalancing connection property. For details on configuring client load balancing, see the individual driver chapter in this book.

Using connection retry

Connection retry defines the number of times the driver attempts to connect to the primary server and, if configured, alternate database servers after the initial unsuccessful connection attempt. It can be used with connection failover, extended connection failover, and select failover. Connection retry can be an important strategy for system recovery. For example, suppose you have a power failure in which both the client and the server fails. When the power is restored and all computers are restarted, the client may be ready to attempt a connection before the server has completed its startup routines. If connection retry is enabled, the client application can continue to retry the connection until a connection is successfully accepted by the server.

Connection retry can be used in environments that have only one server or can be used as a complementary feature with failover in environments with multiple servers.

Using the ConnectionRetryCount and ConnectionRetryDelay properties, you can specify the number of times the driver attempts to connect and the time in seconds between connection attempts. For details on configuring connection retry, see the individual driver chapter in this book.

Always On Availability Groups

The driver supports Always On Availability Groups. Introduced in SQL Server 2012, Always On Availability Groups is a replica-database environment that provides a high-level of data availability, protection, and recovery. Follow the proceeding guidelines to use the driver with Always On Availability Groups.

- You must specify the virtual network name (VNN) of the availability group listener with the ServerName property to connect to an Always On Availability group.
- Set the ApplicationIntent property to `ReadOnly`. By setting applicationIntent to `ReadOnly` and querying read-only database replicas when possible, you can improve efficiency by reducing the workload on read-write nodes.
- Set the MultiSubnetFailover property to `true`. This allows the driver to attempt parallel connections to all the IP addresses associated with an Availability Group. This offers improved response time over traditional failover, which attempts connections to alternate servers one at a time.

See also

ServerName on page 241

ApplicationIntent on page 199

MultiSubnetFailover on page 233

Using failover on page 92

Returning and inserting/updating XML data

The driver supports the xml data type. Which JDBC data type the xml data type is mapped to depends on whether the JDBCBehavior and XMLDescribeType properties are set.

- If XMLDescribeType=`longvarchar` or XMLDescribeType=`longvarbinary`, the driver maps the XML data type to the JDBC LONGVARCHAR or LONGVARBINARY data type, respectively, regardless of the setting of the JDBCBehavior property.
- If JDBCBehavior=`1` (default) and the XMLDescribeType property is not set, the driver maps XML data to the JDBC LONGVARCHAR data type.

- If JDBCBehavior=0 and the XMLDescribeType property is not set, XML data is mapped to SQLXML or LONGVARCHAR, depending on which JVM your application is using. The driver maps the XML data type to the JDBC SQLXML data type if your application is using Java SE 6 or higher. If your application is using an earlier JVM, the driver maps the XML data type to the JDBC LONGVARCHAR data type.

Returning XML data

You can specify whether XML data is returned as character or binary data by setting the XMLDescribeType property. For example, consider a database table defined as:

```
CREATE TABLE xmlTable (id int, xmlCol xml NOT NULL)
```

and the following code:

```
String sql="SELECT xmlCol FROM xmlTable";
ResultSet rs=stmt.executeQuery(sql);
```

If your application uses the following connection URL, which specifies that the XML data type be mapped to the LONGVARBINARY data type, the driver would return XML data as binary data:

```
jdbc:datadirect:sqlserver://server1:1433;DatabaseName=jdbc;User=test;
Password=secret;XMLDescribeType=longvarbinary
```

Returning XML data as character data

When XMLDescribeType=longvarchar, the driver returns XML data as character data. The result set column is described with a column type of LONGVARCHAR and the column type name is xml.

When XMLDescribeType=longvarchar, your application can use the following methods to return data stored in XML columns as character data.

- ResultSet.getString()
- ResultSet.getCharacterStream()
- ResultSet.getClob()
- CallableStatement.getString()
- CallableStatement.getClob()

The driver converts the XML data returned from the database server from the UTF-8 encoding used by the database server to the UTF-16 Java String encoding.

Your application can use the following method to return data stored in XML columns as ASCII data.

- ResultSet.getAsciiStream()

The driver converts the XML data returned from the database server from the UTF-8 encoding to the ISO-8859-1 (latin1) encoding.

Note: This conversion caused by using the getAsciiStream() method may create XML that is not well-formed because the content encoding is not the default encoding and does not contain an XML declaration specifying the content encoding. Do not use the getAsciiStream() method if your application requires well-formed XML.

If XMLDescribeType=`longvarbinary`, your application should not use any of the methods for returning character data described in this section. In this case, the driver applies the standard JDBC character-to-binary conversion to the data, which returns the hexadecimal representation of the character data.

Returning XML data as binary data

When XMLDescribeType=`longvarbinary`, the driver returns XML data as binary data. The result set column is described with a column type of LONGVARBINARY and the column type name is xml.

Your application can use the following methods to return XML data as binary data.

- ResultSet.getBytes()
- ResultSet.getBinaryStream()
- ResultSet.getBlob()
- ResultSet.getObject()
- CallableStatement.getBytes()
- CallableStatement.getBlob()
- CallableStatement.getObject()

The driver does not apply any data conversions to the XML data returned from the database server. These methods return a byte array or binary stream that contains the XML data encoded as UTF-8.

If XMLDescribeType=`longvarchar`, your application should not use any of the methods for returning binary data described in this section. In this case, the driver applies the standard JDBC binary-to-character conversion to the data, which returns the hexadecimal representation of the binary data.

Inserting/updating XML data

The driver can insert or update XML data as character or binary data.

Inserting/updating XML as character data

Your application can use the following methods to insert or update XML data as character data:

- PreparedStatement.setString()
- PreparedStatement.setCharacterStream()
- PreparedStatement.setClob()
- PreparedStatement.setObject()
- ResultSet.updateString()
- ResultSet.updateCharacterStream()
- ResultSet.updateClob()
- ReultSet.updateObject()

The driver converts the character representation of the data to the XML character set used by the database server and sends the converted XML data to the server. The driver does not parse or remove any XML processing instructions.

Your application can update XML data as ASCII data using the following methods:

- PreparedStatement.setAsciiStream()
- ResultSet.updateAsciiStream()

The driver interprets the data returned by these methods using the ISO-8859-1 (latin 1) encoding. The driver converts the data from ISO-8859-1 to the XML character set used by the database server and sends the converted XML data to the server.

Inserting/updating XML as binary data

Your application can use the following methods to insert or update XML data as binary data:

- PreparedStatement.setBytes()
- PreparedStatement.setBinaryStream()
- PreparedStatement.setBlob()
- PreparedStatement.setObject()
- ResultSet.updateBytes()
- ResultSet.updateBinaryStream()
- ResultSet.updateBlob()
- ResultSet.updateObject()

The driver does not apply any data conversions when sending XML data to the database server.

DML with results

The driver supports the Microsoft SQL Server Output clause for Insert, Update, and Delete statements. For example, suppose you created a table with the following statement:

```
CREATE TABLE table1(id int, name varchar(30))
```

The following Update statement updates the values in the id column of table1 and returns a result set that includes the old ID (replaced by the new ID), the new ID, and the name associated with these IDs:

```
UPDATE table1 SET id=id*10 OUTPUT deleted.id as oldId, inserted.id as
   newId, inserted.name
```

The driver returns the results of Insert, Update, or Delete statements and the update count in separate result sets. The output result set is returned first, followed by the update count for the Insert, Update, or Delete statement. To execute DML with Results statements in an application, use the Statement.execute() or PreparedStatement.execute() method. Then, use Statement.getMoreResults () to obtain the output result set and the update count. For example:

```
String sql = "UPDATE table1 SET id=id*10 OUTPUT deleted.id as oldId, " +
   "inserted.id as newId, inserted.name";
boolean isResultSet = stmt.execute(sql);
int   updateCount = 0;
while (true) {
   if (isResultSet) {
         resultSet = stmt.getResultSet();
         while (resultSet.next()) {
            System.out.println("oldId: " + resultSet.getInt(1) +
                               "newId: " + resultSet.getInt(2) +
                               "name: " + resultSet.getString(3));
         }
         resultSet.close();
   }
   else {
         updateCount = stmt.getUpdateCount();
         if (updateCount == -1) {
            break;
         }
         System.out.println("Update Count: " + updateCount);
   }
   isResultSet = stmt.getMoreResults();
}
```

Using client information

Many databases allow applications to store client information associated with a connection, which can be useful for database administration and monitoring purposes. The driver allows applications to store and return the following types of client information.

- Name of the application currently using the connection.
- User ID for whom the application using the connection is performing work. The user ID may be different than the user ID that was used to establish the connection.
- Host name of the client on which the application using the connection is running.
- Product name and version of the driver on the client.
- Additional information that may be used for accounting or troubleshooting purposes, such as an accounting ID.

How databases store client information

Typically, databases that support storing client information do so by providing a register, a variable, or a column in a system table in which the information is stored. If an application attempts to store information and the database does not provide a mechanism for storing that information, the driver caches the information locally. Similarly, if an application returns client information and the database does not provide a mechanism for storing that information, the driver returns the locally cached value.

For example, let’s assume that the following code returns a pooled connection to a database and sets a client application name for that connection. In this example, the application sets the application name SALES157 using the driver property ApplicationName.

```
// Get Database Connection
Connection con = DriverManager.getConnection(
   "jdbc:datadirect:sqlserver://MyServer:1433;DatabaseName=MyDB;
   ApplicationName=SALES157","TEST","secret");
   ...
```

The application name SALES157 is stored locally by the database. When the connection to the database is closed, the client information on the connection is reset to an empty string.

Storing client information

Your application can store client information associated with a connection using any of the following methods:

- Using the driver connection properties listed in "Client information properties."
- Using the following JDBC methods:
  - Connection.setClientInfo(*properties*)
  - Connection.setClientInfo(*property_name*, *value*)
- Using the JDBC extension methods provided in the com.ddtek.jdbc.extensions package.

See also

Client information properties on page 68
JDBC support on page 271
JDBC extensions on page 329

Returning client information

Your application can return client information in the following ways:

- Using the following JDBC methods:
  - Connection.getClientInfo()
  - Connection.getClientInfo(*property_name*)
  - DatabaseMetaData.getClientInfoProperties()
- Using the JDBC extension methods provided in the com.ddtek.jdbc.extensions package.

See also

JDBC support on page 271
JDBC extensions on page 329

Returning metadata about client information locations

You may want to return metadata about the register, variable, or column in which the database stores client information. For example, you may want to determine the maximum length allowed for a client information value before you store that information. If your application attempts to set a client information value that exceeds the maximum length allowed by the database, that value is truncated and the driver generates a warning. Determining the maximum length of the value beforehand can avoid this situation.

To return metadata about client information, call the DatabaseMetaData.getClientInfoProperties() method.

```
// Get Database Connection
Connection con = DriverManager.getConnection(
   "jdbc:datadirect:sqlserver://MyServer:1433;DatabaseName=jdbc", "test", "secret");
DatabaseMetaData metaData = con.getMetaData();
ResultSet rs = metaData.getClientInfoProperties();
...
```

The driver returns a result set that provides the following information for each client information property supported by the database.

- Property name
- Maximum length of the property value
- Default property value
- Property description

Using IP addresses

The driver supports Internet Protocol (IP) addresses in IPv4 and IPv6 formats.

The server name specified in a connection URL, or data source, can resolve to an IPv4 or IPv6 address. In the following example, the server name MyServer can resolve to either type of address:

```
jdbc:datadirect:sqlserver://MyServer:1433;
   DatabaseName=Test;User=admin;Password=secret
```

Alternately, you can specify addresses using IPv4 or IPv6 format in the server portion of the connection URL. For example, the following connection URL specifies the server using an IPv4 address:

```
jdbc:datadirect:sqlserver://123.456.78.90:1433;
   DatabaseName=MyDB;User=admin;Password=secret
```

You also can specify addresses in either format using the ServerName data source property. The following example shows a data source definition that specifies the server name using an IPv6 address:

```
SQLServerDataSource mds = new SQLServerDataSource();
mds.setDescription("My SQLServer DataSource");
mds.setServerName("2001:DB8:0:0:8:800:200C:417A");
...
```

Note: When specifying IPv6 addresses in a connection URL or data source property, the address must be enclosed by brackets.

In addition to the normal IPv6 format, the drivers support IPv6 alternative formats for compressed and IPv4/IPv6 combination addresses. For example, the following connection URL specifies the server using IPv6 format, but uses the compressed syntax for strings of zero bits:

```
jdbc:datadirect:sqlserver://[2001:DB8:0:0:8:800:200C:417A]:1433;
   DatabaseName=MyDB;User=admin;Password=secret
```

Similarly, the following connection URL specifies the server using a combination of IPv4 and IPv6:

```
jdbc:datadirect:sqlserver://[0000:0000:0000:0000:0000:FFFF:123.456.78.90]:1433;
   DatabaseName=MyDB;User=admin;Password=secret
```

For complete information about IPv6, go to the following URL:

http://tools.ietf.org/html/rfc4291#section-2.2

Parameter metadata support

The SQL Server driver supports returning parameter metadata as described in this section.

Insert, Update, and Delete statements

The SQL Server driver supports returning parameter metadata for the following forms of Insert, Update, and Delete statements:

- `INSERT INTO foo VALUES (?, ?, ?)`
- `INSERT INTO foo (col1, col2, col3) VALUES (?, ?, ?)`
- `UPDATE foo SET col1=?, col2=?, col3=? WHERE col1` *`operator`* `? [{AND | OR} col2` *`operator`* `?]`
- `DELETE FROM foo WHERE col1 operator ?`

where *`operator`* is any of the following SQL operators: =, <, >, <=, >=, and <>.

Select statements

The SQL Server driver supports returning parameter metadata for Select statements that contain parameters in ANSI SQL 92 entry-level predicates, for example, such as COMPARISON, BETWEEN, IN, LIKE, and EXISTS predicate constructs. Refer to the ANSI SQL reference for detailed syntax.

Parameter metadata can be returned for a Select statement if one of the following conditions is true:

- The statement contains a predicate value expression that can be targeted against the source tables in the associated FROM clause. For example:

  `SELECT * FROM foo WHERE bar > ?`

  In this case, the value expression "bar" can be targeted against the table "foo" to determine the appropriate metadata for the parameter.

- The statement contains a predicate value expression part that is a nested query. The nested query's metadata must describe a single column. For example:

  `SELECT * FROM foo WHERE (SELECT x FROM y WHERE z = 1) < ?`

The following Select statements show further examples for which parameter metadata can be returned:

```
SELECT col1, col2 FROM foo WHERE col1 = ? and col2 > ?
SELECT ... WHERE colname = (SELECT col2 FROM t2 WHERE col3 = ?)
SELECT ... WHERE colname LIKE ?
SELECT ... WHERE colname BETWEEN ? and ?
SELECT ... WHERE colname IN (?, ?, ?)
SELECT ... WHERE EXISTS(SELECT ... FROM T2 WHERE col1 < ?)
```

ANSI SQL 92 entry-level predicates in a WHERE clause containing GROUP BY, HAVING, or ORDER BY statements are supported. For example:

```
SELECT * FROM t1 WHERE col = ? ORDER BY 1
```

Joins are supported. For example:

```
SELECT * FROM t1,t2 WHERE t1.col1 = ?
```

Fully qualified names and aliases are supported. For example:

```
SELECT a, b, c, d FROM T1 AS A, T2 AS B WHERE A.a = ? and B.b = ?"
```

Stored procedures

The SQL Server driver does not support returning parameter metadata for stored procedure arguments.

ResultSet metadata support

If your application requires table name information, the SQL Server driver can return table name information in ResultSet metadata for Select statements. By setting the ResultSetMetaDataOptions property to 1, the SQL Server driver performs additional processing to determine the correct table name for each column in the result set when the ResultSetMetaData.getTableName() method is called. Otherwise, the getTableName() method may return an empty string for each column in the result set.

When the ResultSetMetaDataOptions property is set to 1 and the ResultSetMetaData.getTableName() method is called, the table name information that is returned by the SQL Server driver depends on whether the column in a result set maps to a column in a table in the database. For each column in a result set that maps to a column in a table in the database, the SQL Server driver returns the table name associated with that column. For columns in a result set that do not map to a column in a table (for example, aggregates and literals), the SQL Server driver returns an empty string.

The Select statements for which ResultSet metadata is returned may contain aliases, joins, and fully qualified names. The following queries are examples of Select statements for which the ResultSetMetaData.getTableName() method returns the correct table name for columns in the Select list:

```
SELECT id, name FROM Employee
SELECT E.id, E.name FROM Employee E
SELECT E.id, E.name AS EmployeeName FROM Employee E
SELECT E.id, E.name, I.location, I.phone FROM Employee E, EmployeeInfo I
   WHERE E.id = I.id
SELECT id, name, location, phone FROM Employee, EmployeeInfo WHERE id = empId
SELECT Employee.id, Employee.name, EmployeeInfo.location, EmployeeInfo.phone
   FROM Employee, EmployeeInfo WHERE Employee.id = EmployeeInfo.id
```

The table name returned by the driver for generated columns is an empty string. The following query is an example of a Select statement that returns a result set that contains a generated column (the column named "upper").

```
SELECT E.id, E.name as EmployeeName, {fn UCASE(E.name)} AS upper FROM Employee E
```

The SQL Server driver also can return schema name and catalog name information when the ResultSetMetaData.getSchemaName() and ResultSetMetaData.getCatalogName() methods are called if the driver can determine that information. For example, for the following statement, the SQL Server driver returns "test" for the catalog name, "test1" for the schema name, and "foo" for the table name:

```
SELECT * FROM test.test1.foo
```

The additional processing required to return table name, schema name, and catalog name information is only performed if the ResultSetMetaData.getTableName(), ResultSetMetaData.getSchemaName(), or ResultSetMetaData.getCatalogName() methods are called.

Isolation levels

The SQL Server driver supports the following isolation levels for Microsoft SQL Server.

Note: For Microsoft Azure Synapse Analytics and Microsoft Analytics Platform System, Read Uncommitted is the only supported isolation level.

- Read Committed with Locks or Read Committed
- Read Committed with Snapshots
- Read Uncommitted
- Repeatable Read
- Serializable
- Snapshot

For Microsoft SQL Server, the default is Read Committed with Locks or Read Committed.

Using the Snapshot isolation level

The driver supports snapshot isolation level.

Note: Snapshot isolation level is not supported for Microsoft Azure Synapse Analytics or Microsoft Analytics Platform System.

You can use snapshot isolation level in either of the following ways:

- Setting the SnapshotSerializable property changes the behavior of the Serializable isolation level to use the Snapshot isolation level. This allows an application to use the Snapshot isolation level with no or minimum code changes. See SnapshotSerializable on page 243 for more information.
- Importing the ExtConstants class allows you to specify the TRANSACTION_SNAPSHOT or TRANSACTION_SERIALIZABLE isolation levels for an individual statement in the same application. The ExtConstants class in the com.ddtek.jdbc.extensions package defines the TRANSACTION_SNAPSHOT

constant. For example, the following code imports the ExtConstants class and sets the TRANSACTION_SNAPSHOT isolation level.

```
import com.ddtek.jdbc.extensions.ExtConstants;
Connection.setTransactionIsolation(
   ExtConstants.TRANSACTION_SNAPSHOT)
```

Using scrollable cursors

The SQL Server driver supports scroll-sensitive result sets, scroll-insensitive result sets, and updatable result sets.

Note: When the SQL Server driver cannot support the requested result set type or concurrency, it automatically downgrades the cursor and generates one or more SQLWarnings with detailed information.

Server-side updatable cursors

The driver can use client-side cursors or server-side cursors to support updatable result sets.

Note: Server-side updatable cursors are not supported for Microsoft Azure Synapse Analytics or Microsoft Analytics Platform System.

By default, the driver uses client-side cursors because this type of cursor can work with any result set type. Using server-side cursors typically can improve performance, but server-side cursors cannot be used with scroll-insensitive result sets or with scroll-sensitive result sets that are not generated from a database table that contains a primary key. To use server-side cursors, set the UseServerSideUpdatableCursors property to `true`.

When the UseServerSideUpdatableCursors property is set to `true` and a scroll-insensitive updatable result set is requested, the driver downgrades the request to a scroll-insensitive read-only result set. Similarly, when a scroll-sensitive updatable result set is requested and the table from which the result set was generated does not contain a primary key, the driver downgrades the request to a scroll-sensitive read-only result set. In both cases, a warning is generated.

When server-side updatable cursors are used with sensitive result sets that are generated from a database table that contains a primary key, the following changes you make to the result set are visible:

- Own Inserts are visible. Others Inserts are not visible.
- Own and Others Updates are visible.
- Own and Others Deletes are visible.

Using the default behavior of the driver (UseServerSideUpdatableCursors=`false`), those changes are not visible.

See also

UseServerSideUpdatableCursors on page 251

JTA support: installing stored procedures

The driver supports distributed transactions through JTA.

Note: Distributed transactions through JTA are not supported for Microsoft Azure, Microsoft Azure Synapse Analytics, or Microsoft Analytics Platform System.

To use JDBC distributed transactions through JTA, use the following procedure to install Microsoft SQL Server JDBC XA procedures. Repeat this procedure for any Microsoft SQL Server installation that uses distributed transactions.

If you have multiple instances of Microsoft SQL Server on the same machine, you can edit the .sql script file with a text editor to specify a fully qualified path to the sqljdbc.dll file for a particular instance. You will run one of two available script files depending on the version of SQL Server you are using.

- For SQL Server 2008 or higher, the instjdbc.sql script should be used.
- For SQL Server 2005, the instjdbc_2005.sql script should be used.

For example, if you want to install XA Procedures for an instance named "MSSQL.2," modify the .sql script file as shown and run it as described in the following procedure.

```
/*
**  add references for the stored procedures
*/
print 'creating JDBC XA procedures'
go
sp_addextendedproc 'xp_jdbc_open',
   'C:\Program Files\Microsoft SQL Server\MSSQL.2\MSSQL\Binn\sqljdbc.dll'
go
sp_addextendedproc 'xp_jdbc_open2',
   'C:\Program Files\Microsoft SQL Server\MSSQL.2\MSSQL\Binn\sqljdbc.dll'
go
sp_addextendedproc 'xp_jdbc_close',
   'C:\Program Files\Microsoft SQL Server\MSSQL.2\MSSQL\Binn\sqljdbc.dll'
go
sp_addextendedproc 'xp_jdbc_close2',
   'C:\Program Files\Microsoft SQL Server\MSSQL.2\MSSQL\Binn\sqljdbc.dll'
go
sp_addextendedproc 'xp_jdbc_start',
   'C:\Program Files\Microsoft SQL Server\MSSQL.2\MSSQL\Binn\sqljdbc.dll'
...
```

Note: You can use the Microsoft SQL Server Configuration Manager tool to view Microsoft SQL Server services and determine the fully qualified path to the \Binn subdirectory of each Microsoft SQL Server instance on a machine. Using the Configuration Manager, right-click on a service and select Properties. Select the Service tab. The path is shown as a value of the Binary Path attribute. Refer to your Microsoft SQL Server documentation for details.

To install stored procedures for JTA:

1. Stop the Microsoft SQL Server instance.
2. Copy the appropriate 32-bit or 64-bit sqljdbc.dll file to the `*SQL_Server_Root*/bin` directory of the Microsoft SQL Server database server:

sqljdbc.dll Version	File Location
32-bit	*install_dir*/SQLServer JTA/32-bit
64-bit Itanium	*install_dir*/SQLServer JTA/64-bit
64-bit AMD64 and Intel EM64T	*install_dir*/SQLServer JTA/x64-bit

where:

install_dir is your product installation directory.

SQL_Server_Root is your Microsoft SQL Server installation directory.

3. Start the Microsoft SQL Server instance.
4. From the database server, use the ISQL utility to run the .sql script. As a precaution, have your system administrator back up the master database before running the script.

   At a command prompt, run the script. For example:

   ```
   ISQL -Usa -Psa_password -Sserver_name -ilocation\instjdbc.sql
   ```

   where:

   sa_password is the password of the system administrator.

   server_name is the name of the server on which the Microsoft SQL Server database resides.

   location is the full path to instjdbc.sql. This script is located in the *install_dir*/SQLServer JTA directory, where *install_dir* is your product installation directory.

5. The script generates many messages. In general, these messages can be ignored; however, the system administrator should scan the output for any messages that may indicate an execution error. The last message should indicate that the script ran successfully. The script fails when there is insufficient space available in the master database to store the JDBC XA procedures or to log changes to existing procedures.

See also

Distributed transaction cleanup on page 112

Distributed transaction cleanup

Connections associated with distributed transactions can become orphaned if the connection to the server is lost before the transaction has completed. When connections associated with distributed transactions are orphaned, any locks held by the database for that transaction are maintained, which can cause data to become unavailable. By cleaning up distributed transactions, connections associated with those transactions are freed and any locks held by the database are released.

You can use the XAResource.recover() method to clean up distributed transactions that have been prepared, but not committed or rolled back. Calling this method returns a list of active distributed transactions that have been prepared, but not committed or rolled back. An application can use the list returned by the XAResource.recover method to clean up those transactions by explicitly committing them or rolling them back. The list of transactions returned by the XAResource.recover method does not include transactions that are active and have not been prepared.

In addition, the SQL Server driver supports the following methods of distributed transaction cleanup.

- Transaction timeout sets a timeout value that is used to audit active transactions. Any active transactions that have a life span greater than the specified timeout value are rolled back. Setting a transaction timeout allows distributed transactions to be cleaned up automatically based on the timeout value.
- Explicit transaction cleanup allows you to explicitly roll back any transactions left in an unprepared state based on a transaction group identifier. Explicit transaction cleanup provides more control than transaction timeout over when distributed transactions are cleaned up.

See also

JTA support: installing stored procedures on page 111

Transaction timeout

To set a timeout value for transaction cleanup, you use the XAResource.setTransactionTimeout method. Setting this value causes sqljdbc.dll on the server side to maintain a list of active transactions. Distributed transactions are placed in the list of active transactions when they are started and removed from this list when they are prepared, rolled back, committed, or forgotten using the appropriate XAResource methods.

When a timeout value is set for transaction cleanup using the XAResource.setTransactionTimeout method, sqljdbc.dll periodically audits the list of active transactions for expired transactions. Any active transactions that have a life span greater than the timeout value are rolled back. If an exception is generated when rolling back a transaction, the exception is written to the sqljdbc.log file, which is located in the same directory as the sqljdbc.dll file.

Setting the transaction timeout value too low means running the risk of rolling back a transaction that otherwise would have completed successfully. As a general guideline, set the timeout value to allow sufficient time for a transaction to complete under heavy traffic load.

Setting a value of `0` (default) disables transaction timeout cleanup.

Explicit transaction cleanup

The SQL Server driver allows you to associate an identifier with a group of transactions using the XATransactionGroup connection property. When you specify a transaction group ID, all distributed transactions initiated by the connection are identified by this ID.

Setting this value causes sqljdbc.dll on the server side to maintain a list of active transactions. Distributed transactions are placed in the list of active transactions when they are started and removed from this list when they are prepared, rolled back, committed, or forgotten using the appropriate XAResource methods.

You can use the XAResource.recover method to roll back any transactions left in an unprepared state that match the transaction group ID on the connection used to call XAResource.recover. For example, if you specified XATransactionGroup=ACCT200 and called the XAResource.recover method on the same connection, any transactions left in an unprepared state with a transaction group ID of ACCT200 would be rolled back.

If an exception is generated when rolling back a transaction, the exception is written to the sqljdbc.log file, which is located in the same directory as the sqljdbc.dll file.

When using explicit transaction cleanup, distributed transactions associated with orphaned connections, and the locks held by those connections, will persist until the application explicitly invokes them. As a general rule, applications should clean up orphaned connections at startup and when the application is notified that a connection to the server was lost.

Unicode support

Multilingual JDBC applications can be developed on any operating system using the driver to access both Unicode and non-Unicode enabled databases. Internally, Java applications use UTF-16 Unicode encoding for string data. When fetching data, the driver automatically performs the conversion from the character encoding used by the database to UTF-16. Similarly, when inserting or updating data in the database, the driver automatically converts UTF-16 encoding to the character encoding used by the database.

The JDBC API provides mechanisms for retrieving and storing character data encoded as Unicode (UTF-16) or ASCII. Additionally, the Java String object contains methods for converting UTF-16 encoding of string data to or from many popular character encodings.

Error handling

SQLExceptions

The driver reports errors to the application by throwing SQLExceptions. Each SQLException contains the following information:

- Description of the probable cause of the error, prefixed by the component that generated the error
- Native error code (if applicable)
- String containing the XOPEN SQLstate

Driver Errors

An error generated by the driver has the format shown in the following example:

```
[DataDirect][SQL Server JDBC Driver]Timeout expired.
```

You may need to check the last JDBC call your application made and refer to the JDBC specification for the recommended action.

Database Errors

An error generated by the database has the format shown in the following example:

```
[DataDirect][SQL Server JDBC Driver][SQL Server]Invalid Object Name.
```

If you need additional information, use the native error code to look up details in your database documentation.

Large object (LOB) support

Although Microsoft SQL Server does not define a Blob or Clob data type, the SQL Server driver allows you to return and update long data, specifically LONGVARBINARY and LONGVARCHAR data, using JDBC methods designed for Blobs and Clobs. When using these methods to update long data as Blobs or Clobs, the updates are made to the local copy of the data contained in the Blob or Clob object.

Retrieving and updating long data using JDBC methods designed for Blobs and Clobs provides some of the same advantages as retrieving and updating Blobs and Clobs. For example, using Blobs and Clobs:

- Provides random access to data
- Allows searching for patterns in the data, such as returning long data that begins with a specific character string

To provide these advantages of Blobs and Clobs, data must be cached. Because data is cached, you will incur a performance penalty, particularly if the data is read once sequentially. This performance penalty can be severe if the size of the long data is larger than available memory.

Batch Inserts and Updates

The SQL Server driver implementation for batch Inserts and Updates is JDBC 3.0 compliant. When the SQL Server driver detects an error in a statement or parameter set in a batch Insert or Update, it generates a BatchUpdateException and continues to execute the remaining statements or parameter sets in the batch. The array of update counts contained in the BatchUpdateException contain one entry for each statement or parameter set. Any entries for statements or parameter sets that failed contain the value Statement.EXECUTE_FAILED.

Rowset support

The SQL Server driver supports any JSR 114 implementation of the RowSet interface, including:

- CachedRowSets
- FilteredRowSets
- WebRowSets
- JoinRowSets
- JDBCRowSets

See https://www.jcp.org/en/jsr/detail?id=114 for more information about JSR 114.

Auto-generated keys support

The driver supports retrieving the values of auto-generated keys. An auto-generated key returned by the driver is the value of an identity column.

Note: Auto-generated keys are not supported for Microsoft Azure Synapse Analytics or Microsoft Analytics Platform System.

An application can return values of auto-generated keys when it executes an Insert statement. How you return these values depends on whether you are using an Insert statement with a Statement object or with a PreparedStatement object, as outlined in the following scenarios:

- When using an Insert statement with a Statement object, the driver supports the following form of the Statement.execute and Statement.executeUpdate methods to instruct the driver to return values of auto-generated keys:
  - `Statement.execute(String` *`sql`*`, int` *`autoGeneratedKeys`*`)`
  - `Statement.execute(String` *`sql`*`, int[]` *`columnIndexes`*`)`

- `Statement.execute(String sql, String[] columnNames)`
- `Statement.executeUpdate(String sql, int autoGeneratedKeys)`
- `Statement.executeUpdate(String sql, int[] columnIndexes)`
- `Statement.executeUpdate(String sql, String[] columnNames)`

- When using an Insert statement with a PreparedStatement object, the driver supports the following form of the Connection.prepareStatement method to instruct the driver to return values of auto-generated keys:
  - `Connection.prepareStatement(String sql, int autoGeneratedKeys)`
  - `Connection.prepareStatement(String sql, int[] columnIndexes)`
  - `Connection.prepareStatement(String sql, String[] columnNames)`

An application can retrieve values of auto-generated keys using the Statement.getGeneratedKeys() method. This method returns a ResultSet object with a column for each auto-generated key.

See Designing JDBC applications for performance optimization on page 345 for information about how auto-generated keys can improve performance.

Null values

When the driver establishes a connection, the driver sets the Microsoft SQL Server database option ANSI_NULLS to on. This action ensures that the driver is compliant with the ANSI SQL standard, which makes developing cross-database applications easier.

By default, Microsoft SQL Server does not evaluate null values in SQL equality (=) or inequality (<>) comparisons or aggregate functions in an ANSI SQL-compliant manner. For example, the ANSI SQL specification defines that `col1=null` as shown in the following Select statement always evaluates to `false`:

```
SELECT * FROM table WHERE col1 = NULL
```

Using the default database setting (ANSI_NULLS=off, the same comparison evaluates to true instead of `false`.

Setting ANSI_NULLS to on changes how the database handles null values and forces the use of `IS NULL` instead of `=NULL`. For example, if the value of col1 in the following Select statement is null, the comparison evaluates to true:

```
SELECT * FROM table WHERE col1 IS NULL
```

In your application, you can restore the default Microsoft SQL Server behavior for a connection in the following ways:

Note: Setting ANSI_NULLS to off is not supported for Microsoft Azure Synapse Analytics or Microsoft Analytics Platform System.

- Use the InitializationString property to specify the SQL command `set ANSI_NULLS off`. For example, the following URL ensures that the handling of null values is restored to the Microsoft SQL Server default for the current connection:

  ```
  jdbc:datadirect:sqlserver://server1:1433;InitializationString=
  set ANSI_NULLS off;DatabaseName=test
  ```

- Explicitly execute the following statement after the connection is established:

```
SET ANSI_NULLS OFF
```

Timeouts

The driver allows you to impose limits on the duration of active sessions through the use of the EnableCancelTimeout and QueryTimeout connection properties. With the LoginTimeout connection property, you can specify how long the driver waits for a connection to be established before timing out the connection request.

See also

Timeout properties on page 66

Using connection properties on page 59

Connection Pool Manager

The DataDirect Connection Pool Manager allows you to pool connections when accessing databases. When your applications use connection pooling, connections are reused rather than created each time a connection is requested. Because establishing a connection is among the most costly operations an application may perform, using Connection Pool Manager to implement connection pooling can significantly improve performance.

How connection pooling works

Typically, creating a connection is the most expensive operation an application performs. Connection pooling allows you to reuse connections rather than create a new one every time an application needs to connect to the database. Connection pooling manages connection sharing across different user requests to maintain performance and reduce the number of new connections that must be created. For example, compare the following transaction sequences.

Example A: Without connection pooling

1. The application creates a connection.
2. The application sends a query to the database.
3. The application obtains the result set of the query.
4. The application displays the result to the end user.
5. The application ends the connection.

Example B: With connection pooling

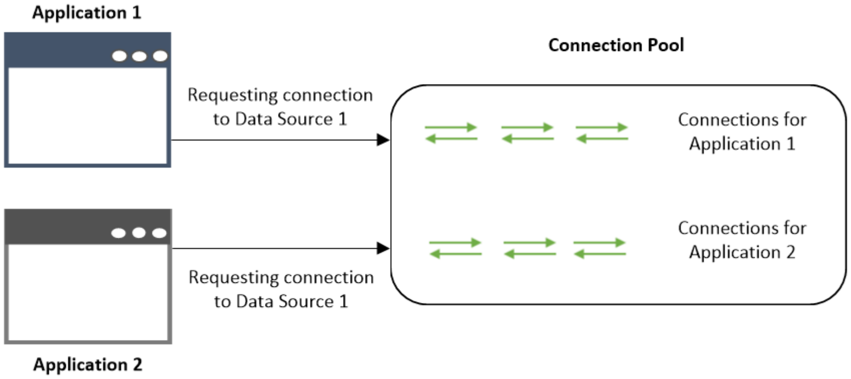
1. The application requests a connection from the connection pool.
2. If an unused connection exists, it is returned by the pool; otherwise, the pool creates a new connection.
3. The application sends a query to the database.
4. The application obtains the result set of the query.

5. The application displays the result to the end user.
6. The application closes the connection, which returns the connection to the pool.

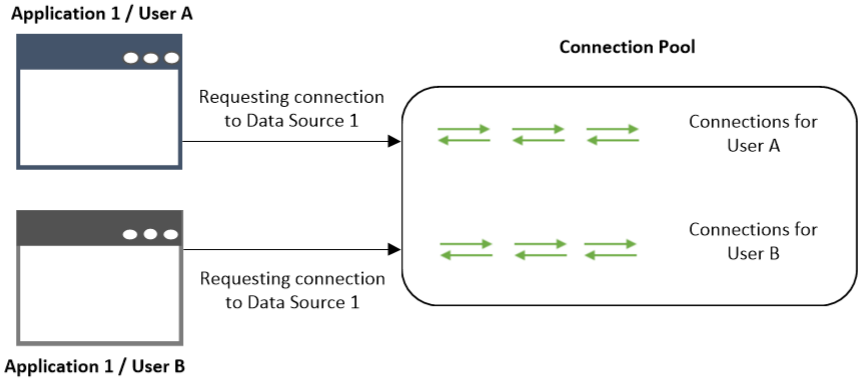
Note: The application calls the close() method, but the connection remains open and the pool is notified of the close request.

The connection pool environment

There is a one-to-one relationship between a JDBC connection pool and a data source, so the number of connection pools used by an application depends on the number of data sources configured to use connection pooling. If multiple applications are configured to use the same data source, those applications share the same connection pool as shown in the following figure.


Application 1
Connection Pool
Requesting connection to Data Source 1
Connections for Application 1
Connections for Application 2
Requesting connection to Data Source 1
Application 2

An application may use only one data source, but allow multiple users, each with their own set of login credentials. The connection pool contains connections for all unique users using the same data source as shown in the following figure.



Connections are one of the following types:

- *Active connection* is a connection that is in use by the application.
- *Idle connection* is a connection in the connection pool that is available for use.

The DataDirect Connection Pool Manager

Connection pooling is performed in the background and does not affect how an application is coded. To use connection pooling, an application must use a DataSource object (an object implementing the DataSource interface) to obtain a connection instead of using the DriverManager class. A DataSource object registers with a JNDI naming service. Once a DataSource object is registered, the application retrieves it from the JNDI naming service in the standard way.

Connection pool implementations, such as the DataDirect Connection Pool Manager, use objects that implement the javax.sql.ConnectionPoolDataSource interface to create the connections managed in a connection pool. All Progress DataDirect data source objects implement the ConnectionPoolDataSource interface.

The DataDirect Connection Pool Manager creates database connections, referred to as PooledConnections, by using the getPooledConnection() method of the ConnectionPoolDataSource interface. Then, the Pool Manager registers itself as a listener to the PooledConnection. When a client application requests a connection, the Pool Manager assigns an available connection. If a connection is unavailable, the Pool Manager establishes a new connection and assigns it to that application.

When the application closes the connection, the driver uses the ConnectionEventListener interface to notify the Pool Manager that the connection is free and available for reuse. The driver also uses the ConnectionEventListener interface to notify the Pool Manager when a connection is corrupted so that the Pool Manager can remove that connection from the pool.

Using a connection pool DataSource object

Once a PooledConnectionDataSource object has been created and registered with JNDI, it can be used by your JDBC application as shown in the following example:

```
Context ctx = new InitialContext();
ConnectionPoolDataSource ds =
(ConnectionPoolDataSource)ctx.lookup("EmployeeDB");
Connection conn = ds.getConnection("domino", "spark");
```

The example begins with the intialization of the JNDI environment. Then, the initial naming context is used to find the logical name of the JDBC DataSource (EmployeeDB). The Context.lookup method returns a reference to a Java object, which is narrowed to a javax.sql.ConnectionPoolDataSource object. Next, the ConnectionPoolDataSource.getPooledConnection() method is called to establish a connection with the underlying database. Then, the application obtains a connection from the ConnectionPoolDataSource.

Implementing DataDirect connection pooling

To use connection pooling, an application must use a DataSource object (an object implementing the DataSource interface) to obtain a connection instead of using the DriverManager class. A DataSource object registers with a JNDI naming service. Once a DataSource object is registered, the application retrieves it from the JNDI naming service in the standard way.

To implement DataDirect connection pooling, perform the following steps.

1. Create and register with JNDI a Progress DataDirect data source object. Once created, the `DataSource` object can be used by a connection pool (`PooledConnectionDataSource` object created in "Creating a driver DataSource object") to create connections for one or multiple connection pools.
2. To create a connection pool, you must create and register with JNDI a `PooledConnectionDataSource` object. A `PooledConnectionDataSource` creates and manages one or multiple connection pools. The `PooledConnectionDataSource` uses the driver `DataSource` object created in "Creating the connection pool" to create the connections for the connection pool.

Creating a driver DataSource object

The following Java code example creates a Progress DataDirect `DataSource` object and registers it with a JNDI naming service.

Note: The `DataSource` class implements the `ConnectionPoolDataSource` interface for pooling in addition to the `DataSource` interface for non-pooling.

```
    SQLServerDataSource mds = new SQLServerDataSource();
    mds.setDescription("My SQL Server Datasource");
    mds.setServerName("MyServer");
    mds.setPortNumber(1433);
    mds.setUser("User123");
    mds.setPassword("secret");
    mds.setDatabaseName("myDB");
```

```
//************************************************************************
// This code creates a Progress DataDirect for JDBC data source and
// registers it to a JNDI naming service. This JDBC data source uses the
// DataSource implementation provided by DataDirect Connect Series
// for JDBC Drivers.
//
// This data source registers its name as <jdbc/ConnectSQLServer>.
//// NOTE: To connect using a data source, the driver needs to access a JNDI data
// store to persist the data source information. To download the JNDI File
// System Service Provider, go to:
//
// http://www.oracle.com/technetwork/java/javasebusiness/downloads/
// java-archive-downloads-java-plat-419418.html#7110-jndi-1.2.1-oth-JPR
////
// Make sure that the fscontext.jar and providerutil.jar files from the
// download are on your classpath.
//************************************************************************
// From DataDirect Connect Series for JDBC:
import java.util.Hashtable;

import javax.naming.Context;
import javax.naming.InitialContext;

import com.ddtek.jdbcx.sqlserver.SQLServerDataSource;
public class SQLServerDataSourceRegisterJNDI {
public static void main(String argv[]) {
  try {

   // Set up data source reference data for naming context:
   // ----------------------------------------------------
   // Create a class instance that implements the interface
   // ConnectionPoolDataSource
   SQLServerDataSource ds = new SQLServerDataSource();
   ds.setDescription("SQL Server DataSource");
            ds.setServerName("MyServer");
            ds.setPortNumber(1433);
            ds.setUser("User123");
            ds.setPassword("secret");
            ds.setDatabaseName("MyDB");
   // Set up environment for creating initial context
   Hashtable env = new Hashtable();
   env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.fscontext.RefFSContextFactory");
   env.put(Context.PROVIDER_URL, "file:C:\\JNDI_Test_Dir");

         Context ctx = new InitialContext(env);
   // Register the data source to JNDI naming service
   ctx.bind("jdbc/ConnectSQLServer", ds);
  } catch (Exception e) {
   System.out.println(e.getStackTrace());
   e.printStackTrace();
   return;
  }
 }// Main
}
// class SQLServerDataSourceRegisterJNDI
```

Creating the connection pool

To create a connection pool, you must create and register with JNDI a `PooledConnectionDataSource` object. The following Java code creates a `PooledConnectionDataSource` object and registers it with a JNDI naming service.

To specify the driver `DataSource` object to be used by the connection pool to create pooled connections, set the parameter of the `DataSourceName` method to the JNDI name of a registered driver `DataSource` object. For example, the following code sets the parameter of the `DataSourceName` method to the JNDI name of the driver `DataSource` object created in "Creating a driver DataSource object."

The `PooledConnectionDataSource` class is provided by the `DataDirect com.ddtek.pool` package. See "PooledConnectionDataSource" for a description of the methods supported by the `PooledConnectionDataSource` class.

```
//************************************************************************
// This code creates a data source and registers it to a JNDI naming service.
// This data source uses the PooledConnectionDataSource
// implementation provided by the DataDirect com.ddtek.pool package.
//
// This data source refers to a registered
// DataDirect Connect Series for JDBC driver DataSource object.
//
// This data source registers its name as <jdbc/ConnectSQLServer>.
//
// NOTE: To connect using a data source, the driver needs to access a JNDI data
// store to persist the data source information. To download the JNDI File
// System Service Provider, go to:
//
// http://www.oracle.com/technetwork/java/javasebusiness/downloads/
// java-archive-downloads-java-plat-419418.html#7110-jndi-1.2.1-oth-JPR
//
// Make sure that the fscontext.jar and providerutil.jar files from the
// download are on your classpath.
//************************************************************************
// From the DataDirect connection pooling package:
import com.ddtek.pool.PooledConnectionDataSource;

import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import javax.naming.directory.*;
import java.util.Hashtable;

public class PoolMgrDataSourceRegisterJNDI
{
   public static void main(String argv[])
   {
      try {
         // Set up data source reference data for naming context:
```

```
         // ----------------------------------------------------
         // Create a pooling manager's class instance that implements
         // the interface DataSource
         PooledConnectionDataSource ds = new PooledConnectionDataSource();

         ds.setDescription("SQL Server DataSource");

         // Specify a registered driver DataSource object to be used
         // by this data source to create pooled connections
         ds.setDataSourceName("jdbc/ConnectSQLServer");

         // The pool manager will be initiated with 5 physical connections
         ds.setInitialPoolSize(5);

         // The pool maintenance thread will make sure that there are 5
         // physical connections available
         ds.setMinPoolSize(5);

         // The pool maintenance thread will check that there are no more
         // than 10 physical connections available
         ds.setMaxPoolSize(10);

         // The pool maintenance thread will wake up and check the pool
         // every 20 seconds
         ds.setPropertyCycle(20);

         // The pool maintenance thread will remove physical connections
         // that are inactive for more than 300 seconds
         ds.setMaxIdleTime(300);

         // Set tracing off because we choose not to see an output listing
         // of activities on a connection
         ds.setTracing(false);

         // Set up environment for creating initial context
         Hashtable env = new Hashtable();
         env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
         env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");
         Context ctx = new InitialContext(env);

         // Register this data source to the JNDI naming service
         ctx.bind("jdbc/SparkyOracle", ds);

         catch (Exception e) {
         System.out.println(e);
         return;
      }
   }
}
```

See also

Creating a driver DataSource object on page 120

PooledConnectionDataSource on page 127

Configuring the connection pool

You can configure attributes of a connection pool for optimal performance and scalability using the methods provided by the DataDirect Connection Pool Manager classes.

Some commonly set connection pool attributes include:

- Minimum pool size, which is the minimum number of connections that will be kept in the pool for each user
- Maximum pool size, which is the maximum number of connections in the pool for each user

- Initial pool size, which is the number of connections created for each user when the connection pool is initialized
- Maximum idle time, which is the amount of time a pooled connection remains idle before it is removed from the connection pool

See also

Connection Pool Manager interfaces on page 126

Configuring the maximum pool size

You set the maximum pool size using the `PooledConnectionDataSource.setMaxPoolSize()` method. For example, the following code sets the maximum pool size to 10:

```
ds.setMaxPoolSize(10);
```

You can control how the Pool Manager implements the maximum pool size by setting the `PooledConnectionDataSource.setMaxPoolSizeBehavior()` method:

- If `setMaxPoolSizeBehavior(softCap)`, the number of active connections can exceed the maximum pool size, but the number of idle connections for each user in the pool cannot exceed this limit. If a user requests a connection and an idle connection is unavailable, the Pool Manager creates a new connection for that user. When the connection is no longer needed, it is returned to the pool. If the number of idle connections exceeds the maximum pool size, the Pool Manager closes idle connections to enforce the pool size limit. This is the default behavior.
- If `setMaxPoolSizeBehavior(hardCap)`, the total number of active and idle connections cannot exceed the maximum pool size. Instead of creating a new connection for a connection request if an idle connection is unavailable, the Pool Manager queues the connection request until a connection is available or the request times out. This behavior is useful if your client or application server has memory limitations or if your database server is licensed for only a certain number of connections.

See also

PooledConnectionDataSource on page 127

Connecting using a connection pool

Because an application uses connection pooling by referencing the JNDI name of a registered PooledConnectionDataSource object, code changes are not required for an application to use connection pooling.

The following example shows Java code that looks up and uses the JNDI-registered PooledConnectionDataSource object created in "Creating the connection pool."

```
//********************************************************************
// Test program to look up and use a JNDI-registered data source.
//
// To run the program, specify the JNDI lookup name for the
// command-line argument, for example:
//
//     java  TestDataSourceApp  <jdbc/ConnectSQLServer>
//********************************************************************
import javax.sql.*;
import java.sql.*;
import javax.naming.*;
import java.util.Hashtable;
public class TestDataSourceApp
{   public static void main(String argv[])
```

```
      {
         String strJNDILookupName = "";
         // Get the JNDI lookup name for a data source
         int nArgv = argv.length;
         if (nArgv != 1) {
            // User does not specify a JNDI lookup name for a data source,
            System.out.println(
               "Please specify a JNDI name for your data source");
            System.exit(0);
            else {
            strJNDILookupName = argv[0];
         }
         DataSource ds = null;
         Connection con = null;
         Context ctx = null;
         Hashtable env = null;
         long nStartTime, nStopTime, nElapsedTime;
         // Set up environment for creating InitialContext object
         env = new Hashtable();
         env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
         env.put(Context.PROVIDER_URL, "file:c:\\JDBCDataSource");
         try {
            // Retrieve the DataSource object that is bound to the logical
            // lookup JNDI name
            ctx = new InitialContext(env);
            ds = (DataSource) ctx.lookup(strJNDILookupName);
            catch (NamingException eName) {
            System.out.println("Error looking up " +
               strJNDILookupName + ": " +eName);
            System.exit(0);
         }
         int numOfTest = 4;
         int [] nCount = {100, 100, 1000, 3000};
         for (int i = 0; i < numOfTest; i ++) {
            // Log the start time
            nStartTime = System.currentTimeMillis();
            for (int j = 1; j <= nCount[i]; j++) {
               // Get Database Connection
               try {
                   con = ds.getConnection("scott", "tiger");
                   // Do something with the connection
                   // ...
                   // Close Database Connection
                   if (con != null) con.close();
                   } catch (SQLException eCon) {
                   System.out.println("Error getting a connection: " + eCon);
                   System.exit(0);
                   } // try getConnection
                } // for j loop
            // Log the end time
            nStopTime = System.currentTimeMillis();
            // Compute elapsed time
            nElapsedTime = nStopTime - nStartTime;
            System.out.println("Test number " + i + ": looping " +
               nCount[i] + " times");
            System.out.println("Elapsed Time: " + nElapsedTime + "\n");
         } // for i loop
         // All done
         System.exit(0);
         // Main
   } // TestDataSourceApp
```

Note: To use non-pooled connections, specify the JNDI name of a registered driver DataSource object as the command-line argument when you run the preceding application. For example, the following command specifies the driver DataSource object created in "Creating a driver DataSource object": `java TestDataSourceApp jdbc/ConnectSQLServer`.

See also

Creating a driver DataSource object on page 120

Creating the connection pool on page 121

Closing the connection pool

The `PooledConnectionDataSource.close()` method can be used to explicitly close the connection pool while the application is running. For example, if changes are made to the pool configuration using a pool management tool, the `PooledConnectionDataSource.close()` method can be used to force the connection pool to close and re-create the pool using the new configuration values.

Using reauthentication

The driver supports reauthentication for Microsoft SQL Server.

Note: Reauthentication is not supported for SQL Server Azure, Azure Synapse Analytics, or Analytics Platform System (Parallel Data Warehouse).

You can configure a connection pool to provide scalability for connections. In addition, to help minimize the number of connections required in a connection pool, you can switch the user associated with a connection to another user, a process known as *reauthentication*.

For example, suppose you are using Kerberos authentication to authenticate users using their operating system user name and password. To reduce the number of connections that must be created and managed, you can use reauthentication to switch the user associated with a connection to multiple users. For example, suppose your connection pool contains a connection, Conn, which was established by the user ALLUSERS. That connection can service multiple users (User A, B, and C) by switching the user associated with the connection Conn to User A, B, and C.

The user performing the switch must have been granted the SQL Server database permission IMPERSONATE. In addition, before performing reauthentication, applications must ensure that any statements or result sets created as one user are closed before switching the connection to another user.

Your application can use the setCurrentUser() method in the ExtConnection interface located in the com.ddtek.jdbc.extensions package to switch a user on a connection. The setCurrentUser() method accepts driver-specific reauthentication options. The reauthentication options supported for the SQL Server driver are:

- CURRENT_DATABASE specifies the name of the current database. The value must be a valid Microsoft SQL Server database name.

  If the setCurrentUser() method is called and this option is specified as an empty string or is not specified, only the user is switched; the database is not switched.

- REVERT_USER determines whether the driver reverts the current user to the initial user before setting the user to a new user for connections that have already reauthenticated.

  - If set to `true` and the setCurrentUser() method is called, the driver reverts the current user to the initial user before setting the connection to the new user. For example, consider a connection that was initially created by User A and was later switched to User B. Before the connection could be further switched to User C, the driver reverts the connection back to User A and then sets it to User C.
  - If set to `false` and the setCurrentUser() method is called, the driver does not revert the current user to the initial user before performing the switch. For example, if the connection was initially created by User

A, switched to User B, and then switched to User C, the driver does not revert the user to User A before switching to User C.

Checking the Pool Manager version

To check the version of your DataDirect Connection Pool Manager, navigate to the directory containing the DataDirect Connection Pool Manager (*install_dir*/pool manager where *install_dir* is your product installation directory). At a command prompt, enter the command:

On Windows:

```
java -classpath poolmgr_dir\pool.jar com.ddtek.pool.PoolManagerInfo
```

On UNIX:

```
java -classpath poolmgr_dir/pool.jar com.ddtek.pool.PoolManagerInfo
```

where:

poolmgr_dir

is the directory containing the DataDirect Connection Pool Manager.

Alternatively, you can obtain the name and version of the DataDirect Connection Pool Manager programmatically by invoking the following static methods:

- `com.ddtek.pool.PoolManagerInfo.getPoolManagerName()`
- `com.ddtek.pool.PoolManagerInfo.getPoolManagerVersion()`

Enabling Pool Manager tracing

You can enable Pool Manager tracing by calling `setTracing(true)` on the `PooledConnectionDataSource` connection. To disable logging, call `setTracing(false)`.

By default, the DataDirect Connection Pool Manager logs its pool activities to the standard output `System.out`. You can change where the Pool Manager trace information is written by calling the `setLogWriter()` method on the `PooledConnectionDataSource` connection.

See "Troubleshooting connection pooling" for information about using a Pool Manager trace file for troubleshooting.

See also

Troubleshooting connection pooling on page 258

Connection Pool Manager interfaces

This section describes the methods used by the DataDirect Connection Pool Manager interfaces: `PooledConnectionDataSourceFactory`, `PooledConnectionDataSource`, and `ConnectionPoolMonitor`.

PooledConnectionDataSourceFactory

The `PooledConnectionDataSourceFactory` interface is used to create a `PooledConnectionDataSource` object from a Reference object that is stored in a naming or directory service. These methods are typically invoked by a JNDI service provider; they are not usually invoked by a user application.

`PooledConnectionDataSourceFactory` Methods	Description
`static Object getObjectInstance(Object` *`refObj`*`, Name` *`name`*`, Context` *`nameCtx`*`, Hashtable` *`env`*`)`	Creates a `PooledConnectionDataSource` object from a Reference object that is stored in a naming or directory service. This is an implementation of the method of the same name defined in the `javax.naming.spi.ObjectFactory` interface. Refer to the Javadoc for this interface for a description.

PooledConnectionDataSource

The `PooledConnectionDataSource` interface is used to create a `PooledConnectionDataSource` object for use with the DataDirect Connection Pool Manager.

`PooledConnectionDataSource` Methods	Description
`void close()`	Closes the connection pool. All physical connections in the pool are closed. Any subsequent connection request re-initializes the connection pool.
`Connection getConnection()`	Obtains a physical connection from the connection pool.
`Connection getConnection(String` *`user`*`, String` *`password`*`)`	Obtains a physical connection from the connection pool, where *`user`* is the user requesting the connection and *`password`* is the password for the connection.
`String getDataSourceName()`	Returns the JNDI name that is used to look up the `DataSource` object referenced by this `PooledConnectionDataSource`.
`String getDescription()`	Returns the description of this `PooledConnectionDataSource`.
`int getInitialPoolSize()`	Returns the value of the initial pool size, which is the number of physical connections created when the connection pool is initialized.
`int getLoginTimeout()`	Returns the value of the login timeout, which is the time allowed for the database login to be validated.
`PrintWriter getLogWriter()`	Returns the writer to which the Pool Manager sends trace information about its activities.
`int getMaxIdleTime()`	Returns the value of the maximum idle time, which is the time a physical connection can remain idle in the connection pool before it is removed from the connection pool.

`PooledConnectionDataSource` Methods	Description
`int getMaxPoolSize()`	Returns the value of the maximum pool size. See "Configuring the maximum pool size" for more information about how the Pool Manager implements the maximum pool size.
`int getMaxPoolSizeBehavior()`	Returns the value of the maximum pool size behavior. See "Configuring the maximum pool size" for more information about how the Pool Manager implements the maximum pool size.
`int getMinPoolSize()`	Returns the value of the minimum pool size, which is the minimum number of idle connections to be kept in the pool.
`int getPropertyCycle()`	Returns the value of the property cycle, which specifies how often the pool maintenance thread wakes up and checks the connection pool.
`Reference getReference()`	Obtains a `javax.naming`. Reference object for this `PooledConnectionDataSource`. The Reference object contains all the state information needed to recreate an instance of this data source using the `PooledConnectionDataSourceFactory` object. This method is typically called by a JNDI service provider when this `PooledConnectionDataSource` is bound to a JNDI naming service.
`public static ConnectionPoolMonitor[ ] getMonitor()`	Returns an array of Connection Pool Monitors, one for each connection pool managed by the Pool Manager.
`public static ConnectionPoolMonitor getMonitor(String` *`name`*`)`	Returns the name of the Connection Pool Monitor for the connection pool specified by *`name`*. If a pool with the specified name cannot be found, this method returns null. The connection pool name has the form: `jndi_name-user_id` where: *`jndi_name`* is the name used for the JNDI lookup of the driver `DataSource` object from which the pooled connection was obtained and *`user_id`* is the user ID used to establish the connections contained in the pool.
`boolean isTracing()`	Determines whether tracing is enabled. If enabled, tracing information is sent to the `PrintWriter` that is passed to the `setLogWriter()` method or the standard output `System.out` if the `setLogWriter()` method is not called.

`PooledConnectionDataSource` Methods	Description
`void setDataSourceName(String` *`dataSourceName`*`)`	Sets the JNDI name, which is used to look up the driver `DataSource` object referenced by this `PooledConnectionDataSource`. The driver `DataSource` object bound to this `PooledConnectionDataSource`, specified by *`dataSourceName`*, is not persisted. Any changes made to the `PooledConnectionDataSource` bound to the specified driver `DataSource` object affect this `PooledConnectionDataSource`.
`void setDataSourceName(String` *`dataSourceName`*`, ConnectionPoolDataSource` *`dataSource`*`)`	Sets the JNDI name associated with this `PooledConnectionDataSource`, specified by *`dataSourceName`*, and the driver `DataSource` object, specified by *`dataSource`*, referenced by this `PooledConnectionDataSource`. The driver `DataSource` object, specified by *`dataSource`*, is persisted with this `PooledConnectionDataSource`. Changes made to the specified driver `DataSource` object after this `PooledConnectionDataSource` is persisted do not affect this `PooledConnectionDataSource`.
`void setDataSourceName(String` *`dataSourceName`*`, Context` *`ctx`*`)`	Sets the JNDI name, specified by *`dataSourceName`*, and context, specified by *`ctx`*, to be used to look up the driver `DataSource` referenced by this `PooledConnectionDataSource`. The JNDI name, specified by *`dataSourceName`*, and context, specified by *`ctx`*, are used to look up a driver `DataSource` object. The driver `DataSource` object is persisted with this `PooledConnectionDataSource`. Changes made to the driver `DataSource` after this `PooledConnectionDataSource` is persisted do not affect this `PooledConnectionDataSource`.
`void setDescription(String` *`description`*`)`	Sets the description of the `PooledConnectionDataSource`, where *`description`* is the description.
`void setInitialPoolSize(int` *`initialPoolSize`*`)`	Sets the value of the initial pool size, which is the number of connections created when the connection pool is initialized.
`void setLoginTimeout(int` *`i`*`)`	Sets the value of the login timeout, where *`i`* is the login timeout, which is the time allowed for the database login to be validated.
`void setLogTimestamp(boolean` *`value`*`)`	If set to `true`, the timestamp is logged when DataDirect Spy logging is enabled. If set to `false`, the timestamp is not logged.
`void setLogTname(boolean` *`value`*`)`	If set to `true`, the thread name is logged when DataDirect Spy logging is enabled. If set to `false`, the thread name is not logged.
`void setLogWriter(PrintWriter` *`printWriter`*`)`	Sets the writer, where *`printWriter`* is the writer to which the stream will be printed.
`void setMaxIdleTime(int` *`maxIdleTime`*`)`	Sets the value in seconds of the maximum idle time, which is the time a connection can remain unused in the connection pool before it is closed and removed from the pool. Zero (0) indicates no limit.

`PooledConnectionDataSource` Methods	Description
`void setMaxPoolSize(int` *`maxPoolSize`*`)`	Sets the value of the maximum pool size, which is the maximum number of connections for each user allowed in the pool. See "Configuring the maximum pool size" for more information about how the Pool Manager implements the maximum pool size.
`void setMaxPoolSizeBehavior(String` *`value`*`)`	Sets the value of the maximum pool size behavior, which is either `softCap` or `hardCap`. If `setMaxPoolSizeBehavior(softCap)`, the number of active connections may exceed the maximum pool size, but the number of idle connections in the connection pool for each user cannot exceed this limit. If a user requests a connection and an idle connection is unavailable, the Pool Manager creates a new connection for that user. When the connection is no longer needed, it is returned to the pool. If the number of idle connections exceeds the maximum pool size, the Pool Manager closes idle connections to enforce the maximum pool size limit. This is the default behavior. If `setMaxPoolSizeBehavior(hardCap)`, the total number of active and idle connections cannot exceed the maximum pool size. Instead of creating a new connection for a connection request if an idle connection is unavailable, the Pool Manager queues the connection request until a connection is available or the request times out. This behavior is useful if your database server has memory limitations or is licensed for only a specific number of connections.The timeout is set using the LoginTimeout connection property. If the connection request times out, the driver throws an exception. See "Configuring the maximum pool size" for more information about how the Pool Manager implements the maximum pool size.
`void setMinPoolSize(int` *`minPoolSize`*`)`	Sets the value of the minimum pool size, which is the minimum number of idle connections to be kept in the connection pool.
`void setPropertyCycle(int` *`propertyCycle`*`)`	Sets the value in seconds of the property cycle, which specifies how often the pool maintenance thread wakes up and checks the connection pool.
`void setTracing(boolean` *`value`*`)`	Enables or disables tracing. If set to `true`, tracing is enabled; if `false`, it is disabled. If enabled, tracing information is sent to the `PrintWriter` that is passed to the `setLogWriter()` method or the standard output `System.out` if the `setLogWriter()` method is not called.

See also

Configuring the maximum pool size on page 123

ConnectionPoolMonitor

The `ConnectionPoolMonitor` interface is used to return information that is useful for monitoring the status of your connection pools.

ConnectionPoolMonitor Methods	Description
`String getName()`	Returns the name of the connection pool associated with the monitor. The connection pool name has the form: *`jndi_name-user_id`* where: *`jndi_name`* is the name used for the JNDI lookup of the `PooledConnectionDataSource` object from which the pooled connection was obtained *`user_id`* is the user ID used to establish the connections contained in the pool.
`int getNumActive()`	Returns the number of connections that have been checked out of the pool and are currently in use.
`int getNumAvailable()`	Returns the number of connections that are idle in the pool (available connections).
`int getInitialPoolSize()`	Returns the initial size of the connection pool (the number of available connections in the pool when the pool was first created).
`int getMaxPoolSize()`	Returns the maximum number of available connection in the connection pool. If the number of available connections exceeds this value, the Pool Manager removes one or multiple available connections from the pool.
`int getMinPoolSize()`	Returns the minimum number of available connections in the connection pool. When the number of available connections is lower than this value, the Pool Manager creates additional connections and makes them available.
`int getPoolSize()`	Returns the current size of the connection pool, which is the total of active connections and available connections.

Statement Pool Monitor

The driver supports the DataDirect Statement Pool Monitor. You can use the Statement Pool Monitor to load statements into and remove statements from the statement pool as well as generate information to help you troubleshoot statement pooling performance. The Statement Pool Monitor is an integrated component of the driver, and you can manage statement pooling directly with DataDirect-specific methods. In addition, the Statement Pool Monitor can be enabled as a Java Management Extensions (JMX) MBean. When enabled as a JMX MBean, the Statement Pool Monitor can be used to manage statement pooling with standard JMX API calls, and it can easily be used by JMX-compliant tools, such as JConsole.

Using DataDirect-specific methods to access the Statement Pool Monitor

To access the Statement Pool Monitor using DataDirect-specific methods, you should first enable statement pooling. You can enable statement pooling by setting the "MaxPooledStatements" connection property to a value greater than zero (0).

The ExtConnection.getStatementPoolMonitor() method returns an ExtStatementPoolMonitor object for the statement pool associated with the connection. This method is provided by the ExtConnection interface in the com.ddtek.jdbc.extensions package. If the connection does not have a statement pool, the method returns null.

Once you have an ExtStatementPoolMonitor object, you can use the poolEntries() method of the ExtStatementPoolMonitorMBean interface implemented by the ExtStatementPoolMonitor to return a list of statements in the statement pool as an array.

See also

MaxPooledStatements on page 232

Using the poolEntries method

Using the `poolEntries()` method, your application can return all statements in the pool or filter the list based on the following criteria:

- Statement type (prepared statement or callable statement)
- Result set type (forward only, scroll insensitive, or scroll sensitive)
- Concurrency type of the result set (read only and updateable)

The following table lists the parameters and the valid values supported by the poolEntries() method.

Table 16: `poolEntries()` Parameters

Parameter	Value	Description
`statementType`	`ExtStatementPoolMonitor.TYPE_PREPARED_STATEMENT`	Returns only prepared statements
	`ExtStatementPoolMonitor.TYPE_CALLABLE_STATEMENT`	Returns only callable statements
	`-1`	Returns all statements regardless of statement type
`resultSetType`	`ResultSet.TYPE_FORWARD_ONLY`	Returns only statements with forward-only result sets
	`ResultSet.TYPE_SCROLL_INSENSITIVE`	Returns only statements with scroll insensitive result sets
	`ResultSet.TYPE_SCROLL_SENSITIVE`	Returns only statements with scroll sensitive result sets
	`-1`	Returns statements regardless of result set type

Parameter	Value	Description
resultSetConcurrency	ResultSet.CONCUR_READ_ONLY	Returns only statements with a read-only result set concurrency
	ResultSet.CONCUR_UPDATABLE	Returns only statements with an updateable result set concurrency
	-1	Returns statements regardless of result set concurrency type

The result of the `poolEntries()` method is an array that contains a String entry for each statement in the statement pool using the format:

```
SQL_TEXT=[SQL_text];STATEMENT_TYPE=TYPE_PREPARED_STATEMENT|
TYPE_CALLABLE_STATEMENT;RESULTSET_TYPE=TYPE_FORWARD_ONLY|
TYPE_SCROLL_INSENSITIVE|TYPE_SCROLL_SENSITIVE;
RESULTSET_CONCURRENCY=CONCUR_READ_ONLY|CONCUR_UPDATABLE;
AUTOGENERATEDKEYSREQUESTED=true|false;
REQUESTEDKEYCOLUMNS=comma-separated_list
```

where *`SQL_text`* is the SQL text of the statement and *`comma-separated_list`* is a list of column names that will be returned as generated keys.

For example:

```
SQL_TEXT=[INSERT INTO emp(id, name) VALUES(99, ?)];
STATEMENT_TYPE=Prepared Statement;RESULTSET_TYPE=Forward Only;
RESULTSET_CONCURRENCY=ReadOnly;AUTOGENERATEDKEYSREQUESTED=false;
REQUESTEDKEYCOLUMNS=id,name
```

Generating a list of statements in the statement pool

The following code shows how to return an ExtStatementPoolMonitor object using a connection and how to generate a list of statements in the statement pool associated with the connection.

```
private void run(String[] args) {
   Connection con = null;
   PreparedStatement prepStmt = null;
   String sql = null;
   try {
      // Create the connection and enable statement pooling
      Class.forName("com.ddtek.jdbc.sqlserver.SQLServerDriver");
      con = DriverManager.getConnection(
         "jdbc:datadirect:sqlserver://MyServer:1433;" +
         "RegisterStatementPoolMonitorMBean=true",
         "maxPooledStatements=10",
         "test", "test");
      // Prepare a couple of statements
      sql = "INSERT INTO employees (id, name) VALUES(?, ?)";
      prepStmt = con.prepareStatement(sql);
      prepStmt.close();
      sql = "SELECT name FROM employees WHERE id = ?";
      prepStmt = con.prepareStatement(sql);
      prepStmt.close();
      ExtStatementPoolMonitor monitor =
         ((ExtConnection) con).getStatementPoolMonitor();
      System.out.println("Statement Pool - " + monitor.getName());
      System.out.println("Max Size:     " + monitor.getMaxSize());
      System.out.println("Current Size: " + monitor.getCurrentSize());
      System.out.println("Hit Count:    " + monitor.getHitCount());
      System.out.println("Miss Count:   " + monitor.getMissCount());
      System.out.println("Statements:");
      ArrayList statements = monitor.poolEntries(-1, -1, -1);
      Iterator itr = statements.iterator();
      while (itr.hasNext()) {
         String entry = (String)itr.next();
         System.out.println(entry);
      }
}
catch (Throwable except) {
   System.out.println("ERROR: " + except);
   }
   finally {
      if (con != null) {
         try {
            con.close();
         }
         catch (SQLException except) {}
         }
      }
   }
```

In the previous code example, the PoolEntries() method returns all statements in the statement pool regardless of statement type, result set cursor type, and concurrency type by specifying the value `-1` for each parameter as shown in the following code:

```
ArrayList statements = monitor.poolEntries(-1, -1, -1);
```

We could have easily filtered the list of statements to return only prepared statements that have a forward-only result set with a concurrency type of updateable using the following code:

```
ArrayList statements = monitor.poolEntries(
   ExtStatementPoolMonitor.TYPE_PREPARED_STATEMENT,
   ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
```

Using JMX to access the Statement Pool Monitor

Your application cannot access the Statement Pool Monitor using JMX unless the driver registers the Statement Pool Monitor as a JMX MBean. To enable the Statement Pool Monitor as an MBean, statement pooling must be enabled with the MaxPooledStatements connection property, and the Statement Pool Monitor MBean must be registered using the RegisterStatementPoolMonitorMBean connection property.

When the Statement Pool Monitor is enabled, the driver registers a single MBean for each statement pool. The registered MBean name has the following form, where *`monitor_name`* is the string returned by the ExtStatementPoolMonitor.getName() method:

`com.ddtek.jdbc.type=StatementPoolMonitor,name=`*`monitor_name`*

Note: Registering the MBean exports a reference to the Statement Pool Monitor. The exported reference can prevent garbage collection on connections if the connections are not properly closed. When garbage collection does not take place on these connections, out of memory errors can occur.

To return information about the statement pool, retrieve the names of all MBeans that are registered with the com.ddtek.jdbc domain and search through the list for the StatementPoolMonitor type attribute. The following code shows how to use the standard JMX API calls to return the state of all active statement pools in the JVM:

```
private void run(String[] args) {
   if (args.length < 2) {
      System.out.println("Not enough arguments supplied");
      System.out.println("Usage: " + "ShowStatementPoolInfo hostname port");
   }
   String hostname = args[0];
   String port = args[1];
   JMXServiceURL           url = null;
   JMXConnector            connector = null;
   MBeanServerConnection   server = null;
   try {
      url = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://" +
                              hostname +":" + port +  "/jmxrmi");
      connector = JMXConnectorFactory.connect(url);
      server = connector.getMBeanServerConnection();
      System.out.println("Connected to JMX MBean Server at " +
                        args[0] + ":" + args[1]);
      // Get the MBeans that have been registered with the
      // com.ddtek.jdbc domain.
      ObjectName ddMBeans = new ObjectName("com.ddtek.jdbc:*");
      Set<ObjectName> mbeans = server.queryNames(ddMBeans, null);
      // For each statement pool monitor MBean, display statistics and
      // contents of the statement pool monitored by that MBean
      for (ObjectName name: mbeans) {
         if (name.getDomain().equals("com.ddtek.jdbc") &&
             name.getKeyProperty("type")
                 .equals("StatementPoolMonitor")) {
            System.out.println("Statement Pool - " +
               server.getAttribute(name, "Name"));
            System.out.println("Max Size:     " +
               server.getAttribute(name, "MaxSize"));
            System.out.println("Current Size: " +
               server.getAttribute(name, "CurrentSize"));
            System.out.println("Hit Count:    " +
               server.getAttribute(name, "HitCount"));
            System.out.println("Miss Count:   " +
               server.getAttribute(name, "MissCount"));
            System.out.println("Statements:");
            Object[] params = new Object[3];
            params[0] = new Integer(-1);
            params[1] = new Integer(-1);
            params[2] = new Integer(-1);
            String[] types = new String[3];
            types[0] = "int";
            types[1] = "int";
            types[2] = "int";
            ArrayList<String>statements = (ArrayList<String>)
               server.invoke(name,
                             "poolEntries",
                              params,
                              types);
            for (String stmt : statements) {
               int index = stmt.indexOf(";");
               System.out.println("   " + stmt.substring(0, index));
            }
         }
      }
   }
   catch (Throwable except) {
      System.out.println("ERROR: " + except);
   }
}
```

See also

MaxPooledStatements on page 232

RegisterStatementPoolMonitorMBean on page 238

Importing statements into a statement pool

When importing statements into a statement pool, for each statement entry in the export file, a statement is added to the statement pool provided a statement with the same SQL text and statement attributes does not already exist in the statement pool. Existing statements in the pool that correspond to a statement entry are kept in the pool unless the addition of new statements causes the number of statements to exceed the maximum pool size. In this case, the driver closes and discards some statements until the pool size is shrunk to the maximum pool size.

For example, if the maximum number of statements allowed for a statement pool is 10 and the number of statements to be imported is 20, only the last 10 imported statements are placed in the statement pool. The other statements are created, closed, and discarded. Importing more statements than the maximum number of statements allowed in the statement pool can negatively affect performance because the driver unnecessarily creates some statements that are never placed in the pool.

To import statements into a statement pool:

1. Create a statement pool export file. See "Statement pool export file example" for an example of a statement pool export file.

   Note: The easiest way to create a statement pool export file is to generate an export file from the statement pool associated with the connection as described in "Generating a statement pool export file."

2. Edit the export file to contain statements to be added to the statement pool.
3. Import the contents of the export file to the statement pool using either of the following methods to specify the path and file name of the export file:
   - Use the ImportStatementPool property. For example:

     ```
     jdbc:datadirect:sqlserver://MyServer:1433;
       User=User123;Password=secret;
       DatabaseName=MyDB;
       ImportStatementPool=C:\\statement_pooling\\stmt_export.txt
     ```

   - Use the importStatements() method of the ExtStatementPoolMonitorMBean interface. For example:

     ```
     ExtStatementPoolMonitor monitor =
        ((ExtConnection)
     con).getStatementPoolMonitor().importStatements
        ("C:\\statement_pooling\\stmt_export.txt");
     ```

See also

Statement pool export file example on page 263

Generating a statement pool export file on page 137

Clearing all statements in a statement pool

To close and discard all statements in a statement pool, use the `emptyPool()` method of the `ExtStatementPoolMonitorMBean` interface. For example:

```
ExtStatementPoolMonitor monitor =
   ((ExtConnection) con).getStatementPoolMonitor().emptyPool();
```

Freezing and unfreezing the statement pool

Freezing the statement pool restricts the statements in the pool to those that were in the pool at the time the pool was frozen. For example, perhaps you have a core set of statements that you do not want replaced by new statements when your core statements are closed. You can freeze the pool using the `setFrozen()` method:

```
ExtStatementPoolMonitor monitor =
   ((ExtConnection) con).getStatementPoolMonitor().setFrozen(true);
```

Similarly, you can use the same method to unfreeze the statement pool:

```
ExtStatementPoolMonitor monitor =
   ((ExtConnection) con).getStatementPoolMonitor().setFrozen(false);
```

When the statement pool is frozen, your application can still clear the pool and import statements into the pool. In addition, you can use the `Statement.setPoolable()` method to add or remove single statements from the pool regardless of the pool's frozen state, assuming the pool is not full. If the pool is frozen and the number of statements in the pool is the maximum, no statements can be added to the pool.

To determine if a pool is frozen, use the `isFrozen()` method.

Generating a statement pool export file

You may want to generate an export file in the following circumstances:

- To import statements to the statement pool, you can create an export file, edit its contents, and import the file into the statement pool to import statements to the pool.
- To examine the characteristics of the statements in the statement pool to help you troubleshoot statement pool performance.

To generate a statement pool export file, use the `exportStatements()` method of the `ExtStatementPoolMonitorMBean` interface. For example, the following code exports the contents of the statement pool associated with the connection to a file named `stmt_export.txt`:

```
ExtStatementPoolMonitor monitor =
   ((ExtConnection) con).getStatementPoolMonitor().exportStatements
   ("stmt_export.txt");
```

See the "Statement pool export file example" topic for information on interpreting the contents of an export file.

See also

Statement pool export file example on page 263

DataDirect Statement Pool Monitor interfaces and classes

This section describes the methods used by the DataDirect Statement Pool Monitor interfaces and classes.

ExtStatementPoolMonitor class

This class is used to control and monitor a single statement pool. This class implements the `ExtStatementPoolMonitorMBean` interface.

ExtStatementPoolMonitorMBean interface

`ExtStatementPoolMonitorMBean` Methods	Description
`String getName()`	Returns the name of a Statement Pool Monitor instance associated with the connection. The name is comprised of the name of the driver that established the connection, and the name and port of the server to which the Statement Pool Monitor is connected, and the MBean ID of the connection.
`int getCurrentSize()`	Returns the total number of statements cached in the statement pool.
`long getHitCount()`	Returns the hit count for the statement pool. The hit count is the number of times a lookup is performed for a statement that results in a cache hit. A cache hit occurs when the Statement Pool Monitor successfully finds a statement in the pool with the same SQL text, statement type, result set type, result set concurrency, and requested generated key information. This method is useful to determine if your workload is using the statement pool effectively. For example, if the hit count is low, the statement pool is probably not being used to its best advantage.
`long getMissCount()`	Returns the miss count for the statement pool. The miss count is the number of times a lookup is performed for a statement that fails to result in a cache hit. A cache hit occurs when the Statement Pool Monitor successfully finds a statement in the pool with the same SQL text, statement type, result set type, result set concurrency, and requested generated key information. This method is useful to determine if your workload is using the statement pool effectively. For example, if the miss count is high, the statement pool is probably not being used to its best advantage.
`int getMaxSize()`	Returns the maximum number of statements that can be stored in the statement pool.
`int setMaxSize(int` *`value`*`)`	Changes the maximum number of statements that can be stored in the statement pool to the specified value.
`void emptyPool()`	Closes and discards all the statements in the statement pool.
`void resetCounts()`	Resets the hit and miss counts to zero (0). See long `getHitCount()` and long `getMissCount()` for more information.

`ExtStatementPoolMonitorMBean` Methods	Description
`ArrayList poolEntries(int` *`statementType`*`, int` *`resultSetType`*`, int` *`resultSetConcurrency`*`)`	Returns a list of statements in the pool. The list is an array that contains a String entry for each statement in the statement pool.
`void exportStatements(File` *`file_object`*`)`	Exports statements from the statement pool into the specified file. The file format contains an entry for each statement in the statement pool.
`void exportStatements(String` *`file_name`*`)`	Exports statements from statement pool into the specified file. The file format contains an entry for each statement in the statement pool.
`void importStatements(File` *`file_object`*`)`	Imports statements from the specified File object into the statement pool.
`void importStatements(String` *`file_name`*`)`	Imports statements from the specified file into the statement pool.
`boolean isFrozen()`	Returns whether the state of the statement pool is frozen. When the statement pool is frozen, the statements that can be stored in the pool are restricted to those that were in the pool at the time the pool was frozen. Freezing a pool is useful if you have a core set of statements that you do not want replaced by other statements when the core statements are closed.
`void setFrozen(boolean)`	`setFrozen(true)` freezes the statement pool. `setFrozen(false)` unfreezes the statement pool. When the statement pool is frozen, the statements that can be stored in the pool are restricted to those that were in the pool at the time the pool was frozen. Freezing a pool is useful if you have a core set of statements that you do not want replaced by other statements when the core statements are closed.

DataDirect Bulk Load

In addition to supporting the native bulk protocols in SQL Server databases, the driver supports DataDirect Bulk Load. DataDirect Bulk Load allows you to perform bulk load operations by creating a `DDBulkLoad` object and using the methods provided by the `DDBulkLoad` interface in the `com.ddtek.jdbc.extensions` package for bulk load. You may want to use this method if you are developing a new application that needs to perform bulk load operations.

Important: Because a bulk load operation may bypass data integrity checks, your application must ensure that the data it is transferring does not violate integrity constraints in the database. For example, suppose you are bulk loading data into a database table and some of that data duplicates data stored as a primary key, which must be unique. The driver will not throw an exception to alert you to the error; your application must provide its own data integrity checks.

See also

Bulk load properties on page 63

Using a DDBulkLoad object

The first step in performing a bulk load operation using a DDBulkLoad object is to create a DDBulkLoad object. A DDBulkLoad must be created with the getInstance method using the DDBulkLoadFactory class as shown in the following example.

```
import com.ddtek.jdbc.extensions.*
// Get SQLServer Connection
Connection con = DriverManager.getConnection(
   "jdbc:datadirect:sqlserver://MyServer:1433;User=User123;
    Password=secret;DatabaseName=MyDB");

// Get a DDBulkLoad object
DDBulkLoad bulkLoad = DDBulkLoadFactory.getInstance(con);
```

Once the DDBulkLoad object has been created, DDBulkLoad methods can be used to instruct the driver to obtain data from one location and load it to another location. The driver can obtain data either from a ResultSet object or from a CSV file.

Migrating data using a ResultSet object

The following steps would need to be taken to migrate data from Oracle to SQL Server using a ResultSet object.

Important: This scenario assumes that you are using the DataDirect Oracle driver to query the Oracle database and the DataDirect SQL Server driver to insert data to the SQL Server database.

1. The application creates a DDBulkLoad object.
2. The application executes a standard query on the Oracle database, and the Oracle driver retrieves the results as a ResultSet object.
3. The application instructs the SQL Server driver to load the data from the ResultSet object into SQL Server. (See "Loading data from a ResultSet object".)

Migrating data using a CSV file

The following steps would need to be taken to migrate data from Oracle to SQL Server using a CSV file.

1. The application creates a DDBulkLoad object.
2. The application instructs the Oracle driver to export the data from the Oracle database into a CSV file. (See "Exporting data to a CSV file.")
3. The application instructs the SQL Server driver to load data from the CSV file into SQL Server. (See "Loading data from a CSV file.")

See also

CSV files on page 143

JDBC extensions on page 329

Permissions for bulk load from a CSV file on page 50

Exporting data to a CSV file

Using a BulkLoad object, data can be exported either as a table or ResultSet object.

Exporting Data as a Table

To export data as a table, the application must specify the table name with the setTableName method and then export the data to a CSV file using the export method. For example, the following code snippet specifies the GBMAXTABLE table and exports it to a CSV file called tmp.csv.

```
bulkLoad.setTableName("GBMAXTABLE");
bulkLoad.export("tmp.csv");
```

Alternatively, you can create a file reference to the CSV file and use the export method to specify the file reference. For example:

```
File csvFile = new File ("tmp.csv");
bulkLoad.export(csvFile);
```

Exporting Data as a ResultSet object

To export data as a ResultSet object, the application must first create a file reference to a CSV file, and then, using the export method, specify the ResultSet object and the file reference. For example, the following code snippet creates the tmp.csv file reference and then specifies MyResults (the ResultSet object be exported).

```
File csvFile = new File ("tmp.csv");
bulkLoad.export(MyResults, csvFile);
```

If the CSV file does not already exist, the driver creates it when the export method is executed. The driver also creates a bulk load configuration file, which describes the structure of the CSV file. For more information about using CSV files, see "CSV Files." See "JDBC Extensions" for more information about bulk load methods.

See also

CSV files on page 143

JDBC extensions on page 329

Permissions for bulk load from a CSV file on page 50

Loading data from a ResultSet object

The setTableName method should be used to specify the table into which you want to load the data. Then, the load method is used to specify the ResultSet object that contains the data you are loading. For example, the following code snippet loads data from a ResultSet object named MyResults into a table named GBMAXTABLE

```
bulkLoad.setTableName("GBMAXTABLE");
bulkLoad.load(MyResults);
```

This example loads the first column in the result set to the ColName1 column, the second column in the result set to the ColName2 column, and so on.

You can use the BulkLoadBatchSize property to specify the number of rows the driver loads at a time when bulk loading data. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Be aware that increasing the number of rows that are loaded also causes the driver to consume more memory on the client.

See also

BulkLoadBatchSize on page 202

JDBC extensions on page 329

Loading data from a CSV file

The `setTableName` method should be used to specify the table into which you want to load the data. Then, the `load` method is used to specify the CSV file that contains the data you are loading. For example:

```
bulkLoad.setTableName("GBMAXTABLE");
bulkLoad.load("tmp.csv");
```

Alternatively, you can create a file reference to the CSV file, and use the load() method to specify the file reference:

```
File csvFile = new File("tmp.csv");
bulkLoad.load(csvFile);
```

For the Salesforce driver, you also can specify the columns to which you want to load the data. This example loads the first column in the CSV file to the `ColName1` column, the second column in the CSV file to the `ColName2` column, and the third column in the CSV file to the `ColName3` column:

```
bulkLoad.setTableName("GBMAXTABLE(ColName1, ColName2, ColName3)");
bulkLoad.load("tmp.csv");
```

Use the BulkLoadBatchSize property to specify the number of rows the driver loads at a time when bulk loading data. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Be aware that increasing the number of rows that are loaded also causes the driver to consume more memory on the client. See "JDBC Extensions" for more information about bulk load methods.

See also

CSV files on page 143

BulkLoadBatchSize on page 202

JDBC extensions on page 329

Specifying the bulk load operation

You can specify which type of bulk load operation will be performed when a load method is called by setting the operation property using the `setProperties` method of the `DDBulkLoad` interface. The operation property accepts the following values: `insert`, `update`, `delete` and `upsert`. The default value is `insert`.

The following example changes the type of bulk load operation to update.

```
DDBulkLoad bulkLoad =
com.ddtek.jdbc.extensions.DDBulkLoadFactory.getInstance(connection);
Properties props = new Properties();
props.put("operation", "update");
bulkLoad.setProperties(props);
```

See also

JDBC extensions on page 329

Logging

If logging is enabled for bulk load, a log file records information for each bulk load operation. Logging is enabled by specifying a file name and location for the log file using the `setLogFile` method.

The log file records the following types of information about each bulk load operation.

- Total number of rows that were read
- Total number of rows that successfully loaded

  Note: The total number of rows that successfully loaded is not provided for Microsoft Azure Synapse Analytics or Microsoft Analytics Platform System.

- Total number of rows that failed to load

For example, the following log file shows that the 11 rows read were all successfully loaded.

```
/*----- Load Started: <Feb 25, 2009 11:20:09 AM EST>---------------------*/
Total number of rows read 11
Total number of rows successfully loaded 11
Total number of rows that failed to load 0
```

Enabling Logging on Windows

To enable logging using a log file named `bulk_load.log` located in the `C:\temp` directory, you would specify:

```
bulkLoad.setLogFile(C:\\temp\\bulk_load.log)
```

Note: If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `C:\\temp\\bulk_load.log`.

Enabling Logging on UNIX/Linux

To enable logging using a log file named `bulk_load.log` located in the `/tmp` directory, you would specify:

```
bulkLoad.setLogFile(/tmp/bulk_load.log)
```

CSV files

As described in "Exporting data to a CSV file," the driver can create a CSV file by exporting data from a table or `ResultSet` object. For example, suppose you want to export data from a 4-column table named `GBMAXTABLE` into a CSV file. The contents of the CSV file, named `GBMAXTABLE.csv`, might look like the following example:

```
1,0x6263,"bc","bc"
2,0x636465,"cde","cde"
3,0x64656667,"defg","defg"
4,0x6566676869,"efghi","efghi"
5,0x666768696a6b,"fghijk","fghijk"
6,0x6768696a6b6c6d,"ghijklm","ghijklm"
7,0x68696a6b6c6d6e6f,"hijklmno","hijklmno"
8,0x696a6b6c6d6e6f7071,"ijklmnopq","ijklmnopq"
9,0x6a6b6c6d6e6f70717273,"jklmnopqrs","jklmnopqrs"
10,0x6b,"k","k"
```

See also

Exporting data to a CSV file on page 141

Bulk load configuration file

Each time data is exported to a CSV file, a bulk load configuration file is created. This file has the same name as the CSV file, but with an `.xml` extension (for example, `GBMAXTABLE.xml`).

In its metadata, the bulk load configuration file defines the names and data types of the columns in the CSV file based on those in the table or `ResultSet` object. It also defines other data properties, such as the source code page for character data types, precision and scale for numeric data types, and nullability for all data types. The format of `GBMAXTABLE.xml` might look like the following example.

Note: If the driver cannot read a bulk load configuration file (for example, because it was inadvertently deleted), the driver reads all data as character data. The character set used by the driver is UTF-8.

```
<?xml version="1.0" encoding="utf-8"?>
<table codepage="UTF-8" xsi:noNamespaceSchemaLocation=
"https://documentation.progress.com/output/DataDirect/collateral/BulkData.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row>
   <column datatype="DECIMAL" precision="38" scale="0" nullable="false">
      INTEGERCOL</column>
   <column datatype="VARBINARY" length="10"
nullable="true">VARBINCOL</column>
   <column datatype="VARCHAR" length="10" sourcecodepage="Windows-1252"
      externalfilecodepage="Windows-1252"
nullable="true">VCHARCOL</column>
   <column datatype="VARCHAR" length="10" sourcecodepage="Windows-1252"
      externalfilecodepage="Windows-1252"
nullable="true">UNIVCHARCOL</column>
</row>
</table>
```

Bulk load configuration file schema

The bulk load configuration file must conform to the bulk load configuration XML schema defined at the following Web site.

https://documentation.progress.com/output/DataDirect/collateral/BulkData.xsd

The driver throws an exception if either of the following circumstances occur.

- If the driver performs a data export and the CSV file cannot be created
- If the driver performs a bulk load operation and the driver detects that the CSV file does not comply with the XML schema described in the bulk load configuration file

Character set conversions

When you export data from a database to a CSV file, the CSV file uses the same code page as the table from which the data was exported. If the CSV file and the target table use different code pages, performance for bulk load operations can suffer because the driver must perform a character set conversion.

To avoid character set conversions, your application can specify which code page to use for the CSV file when exporting data. For example, if the source database table uses a SHIFT-JIS code page and the target table uses a EUC-JP code page, specify `setCodePage("EUC_JP")` to ensure that the CSV file will use the same code page as the target table.

You can specify any of the following code pages.

Note: If the code page you need to use is not listed, contact Customer Support to request support for that code page.

US_ASCII

ISO_8859_1

ISO_8859_2

ISO_8859_3

ISO_8859_4

ISO_8859_5

ISO_8859_6

ISO_8859_7

ISO_8859_8

ISO_8859_9

JIS_Encoding

Shift_JIS

EUC_JP

KS_C_5601

ISO_2022_KR

EUC_KR

ISO_2022_JP

GB2312

ISO_8859_13

ISO_8859_15

GBK

IBM850

IBM852

IBM437

IBM862

Big5

MACINTOSH

IBM037

IBM273

IBM277

IBM278

IBM280

IBM284

IBM285

IBM290

IBM297

IBM420

IBM424

IBM500

IBM851

IBM855

IBM857

IBM860

IBM861

IBM863

IBM864

IBM865

IBM869

IBM870

IBM871

IBM1026

KOI8_R

HZ_GB_2312

IBM866

IBM775

IBM00858

IBM01140

IBM01141

IBM01142

IBM01143

IBM01144

IBM01145

IBM01146

IBM01147

IBM01148

IBM01149

WINDOWS-1250

WINDOWS-1251

WINDOWS-1252

WINDOWS-1253

WINDOWS-1254

WINDOWS-1255

WINDOWS-1256

WINDOWS-1257

WINDOWS-1258

WINDOWS-854

IBM-939

IBM-943_P14A-2000

IBM-4396

IBM-5026

IBM-5035

UTF-8

UTF-16LE

UTF-16BE

External overflow files

When you export data into a CSV file, you can choose to create one large file or multiple smaller files. For example, if you are exporting BLOB data that is a total of several GB, you may want to break the data that into multiple smaller files of 100 MB each.

If the values set by the `setCharacterThreshold` or `setBinaryThreshold` methods are exceeded, separate files are generated to store character or binary data, respectively. Overflow files are located in the same directory as the CSV file.

The format for overflow file names is:

`CSV_file_name``.`*`xxxxxx`*`.lob`

where:

`CSV_file_name` is the name of the CSV file.

`xxxxxx` is a 6-digit number that increments an overflow file.

For example, if multiple overflow files are created for a CSV file named `CSV1`, the file names would look like this:

```
CSV1.000001.lob
```

```
CSV1.000002.lob
```

```
CSV1.000003.lob
```

```
...
```

If the overflow file contains character data, the code page used by the file is the code page specified in the bulk load configuration file for the CSV file.

Discard file

If the driver was unable to load rows into the database for a bulk load operation from a CSV file, it can record all the rows that were not loaded in a discard file. The contents of the discard file is in the same format as that of the CSV file. After fixing reported issues in the discard file, the bulk load can be reissued, using the discard file as the CSV file.

A discard file is created by specifying a file name and location for the discard file using the `setDiscardFile` method.

Creating a discard file on Windows

To create a discard file named `discard.csv` located in the `C:\temp` directory, you would specify:

```
bulkLoad.setDiscardFile(C:\\temp\\discard.csv)
```

Note: If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `C:\\temp\\discard.csv`.

Creating a discard file on UNIX/Linux

To create a discard file named `discard.csv` located in the `/tmp` directory, you would specify:

```
bulkLoad.setDiscardFile(/tmp/discard.csv)
```

DataDirect Test

Use DataDirect Test to test your JDBC applications and learn the JDBC API. DataDirect Test contains menu selections that correspond to specific JDBC functions, for example, connecting to a database or passing a SQL statement. DataDirect Test allows you to perform the following tasks:

- Execute a single JDBC method or execute multiple JDBC methods simultaneously, so that you can easily perform some common tasks, such as returning result sets
- Display the results of all JDBC function calls in one window, while displaying fully commented, JDBC code in an alternate window

DataDirect Test works only with JDBC drivers from Progress DataDirect.

DataDirect Test tutorial

This DataDirect Test tutorial explains how to use the most important features of DataDirect Test (and the JDBC API) and assumes that you can connect to a database with the standard available demo table or fine-tune the sample SQL statements shown in this example as appropriate for your environment.

Note: The tutorial describes functionality across a spectrum of data stores. In some cases, the functionality described may not apply to the driver or data store you are using. Additionally, examples are drawn from a variety of drivers and data stores.

Note: The step-by-step examples used in this tutorial do not show typical clean-up routines (for example, closing result sets and connections). These steps have been omitted to simplify the examples. Do not forget to add these steps when you use equivalent code in your applications.

Configuring DataDirect Test

The default DataDirect Test configuration file is:

`install_dir/testforjdbc/Config.txt`

where:

`install_dir`

is your product installation directory.

The DataDirect Test configuration file can be edited as appropriate for your environment using any text editor. All parameters are configurable, but the most commonly configured parameters are:

Drivers	A list of colon-separated JDBC driver classes.
DefaultDriver	The default JDBC driver that appears in the **Get Driver URL** window.
Databases	A list of comma-separated JDBC URLs. The first item in the list appears as the default in the **Database Selection** window. You can use one of these URLs as a template when you make a JDBC connection. The default Config.txt file contains example URLs for most databases.

InitialContextFactory

Set to `com.sun.jndi.fscontext.RefFSContextFactory` if you are using file system data sources, or `com.sun.jndi.ldap.LdapCtxFactory` if you are using LDAP.

ContextProviderURL

The location of the .bindings file if you are using file system data sources, or your LDAP Provider URL if you are using LDAP.

Datasources

A list of comma-separated JDBC data sources. The first item in the list appears as the default in the **Data Source Selection** window.

To connect using a data source, DataDirect Test needs to access a JNDI data store to persist the data source information. By default, DataDirect Test is configured to use the JNDI File System Service Provider to persist the data source. You can download the JNDI File System Service Provider from the Oracle Java Platform Technology Downloads page.

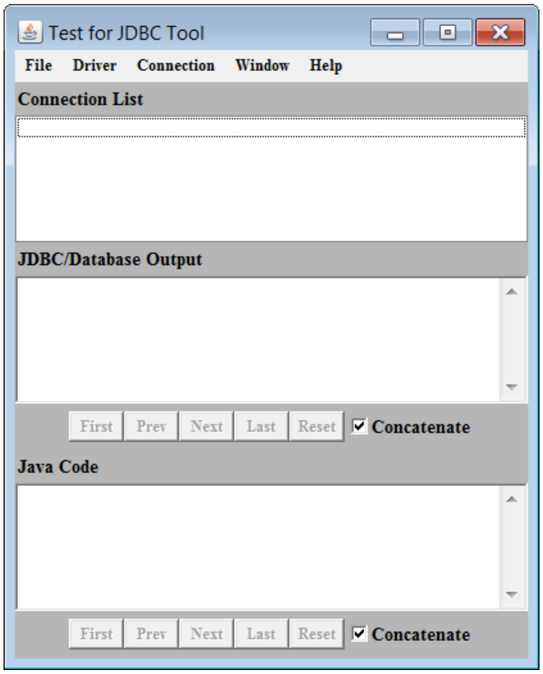
Make sure that the `fscontext.jar` and `providerutil.jar` files from the download are on your classpath.

Starting DataDirect Test

How you start DataDirect Test depends on your platform:

- **As a Java application on Windows**. Run the `testforjdbc.bat` file located in the `testforjdbc` subdirectory of your product installation directory.
- **As a Java application on Linux/UNIX**. Run the `testforjdbc.sh` shell script located in the `testforjdbc` subdirectory in the installation directory.

After you start DataDirect Test, the **Test for JDBC Tool** window appears.



The main **Test for JDBC Tool** window shows the following information:

- In the **Connection List** box, a list of available connections.
- In the **JDBC/Database Output** scroll box, a report indicating whether the last action succeeded or failed.
- In the **Java Code** scroll box, the actual Java code used to implement the last action.

Tip: DataDirect Test windows contain two **Concatenate** check boxes. Select a **Concatenate** check box to see a cumulative record of previous actions; otherwise, only the last action is shown. Selecting **Concatenate** can degrade performance, particularly when displaying large result sets.

Connecting using DataDirect Test

You can use either of the following methods to connect using DataDirect Test:

- Using a data source
- Using a driver/database selection

Connecting using a data source

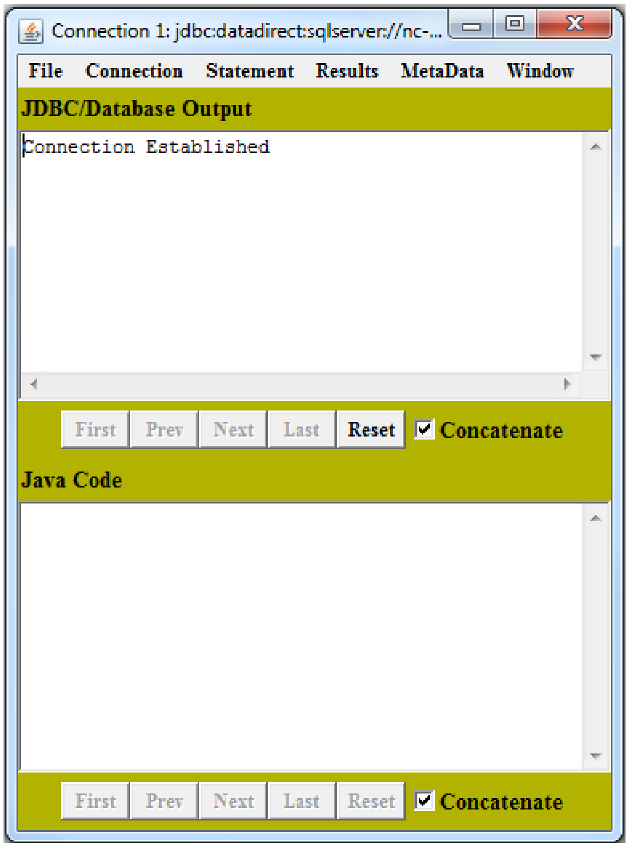
To connect using a data source, DataDirect Test needs to access a JNDI data store to persist the data source information. By default, DataDirect Test is configured to use the JNDI File System Service Provider to persist the data source. You can download the JNDI File System Service Provider from the Oracle Java Platform Technology Downloads page.

Make sure that the `fscontext.jar` and `providerutil.jar` files from the download are on your classpath.

To connect using a data source:

1. From the main **Test for JDBC Tool** window menu, select **Connection / Connect to DB via Data Source**. The **Select A Datasource** window appears.
2. Select a data source from the **Defined Datasources** pane. In the User Name and Password fields, type values for the User and Password connection properties; then, click **Connect**. For information about JDBC connection properties, refer to your driver's connection property descriptions.
3. If the connection was successful, the **Connection** window appears and shows the `Connection Established` message in the JDBC/Database Output scroll box.

   Important: For the Autonomous REST Connector: REST API data sources do not connect in the same manner as database servers; therefore; the `Connection Established` notification does not guarantee that the driver is properly configured. To confirm that the driver is correctly configured, you will need to retrieve data from an endpoint using the methods described in "Executing a simple database selection."



```
Connection Established
```

See also

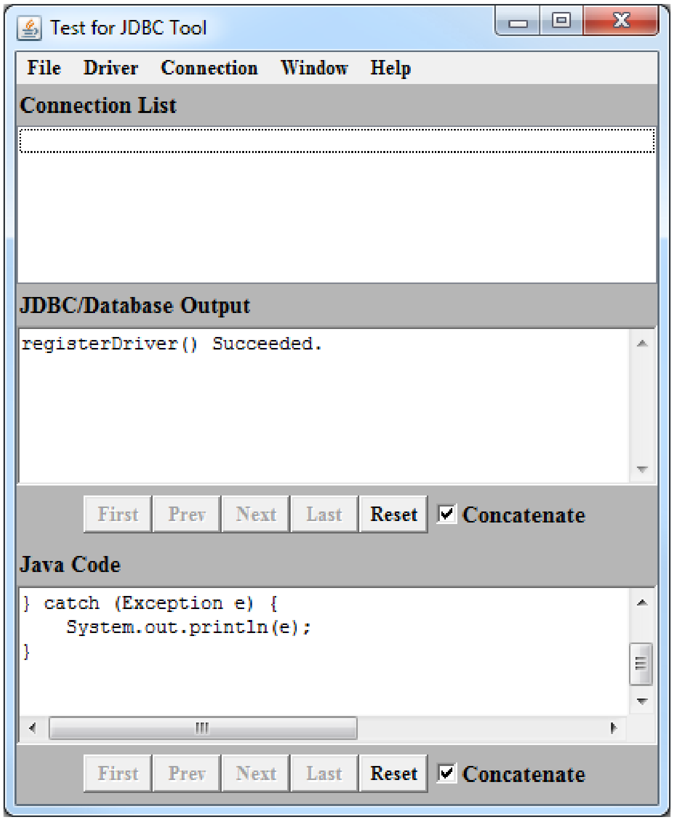
Executing a simple Select statement on page 152

Connecting using database selection

To connect using database selection:

1. From the **Test for JDBC Tool** window menu, select **Driver / Register Driver**. DataDirect Test prompts for a JDBC driver name.
2. In the Please Supply a Driver URL field, specify a driver (for example `com.ddtek.jdbc.sqlserver.SQLServerDriver`); then, click **OK**.

   If the driver was registered successfully, the **Test for JDBC Tool** window appears with a confirmation in the JDBC/Database Output scroll box.



```
registerDriver() Succeeded.
```

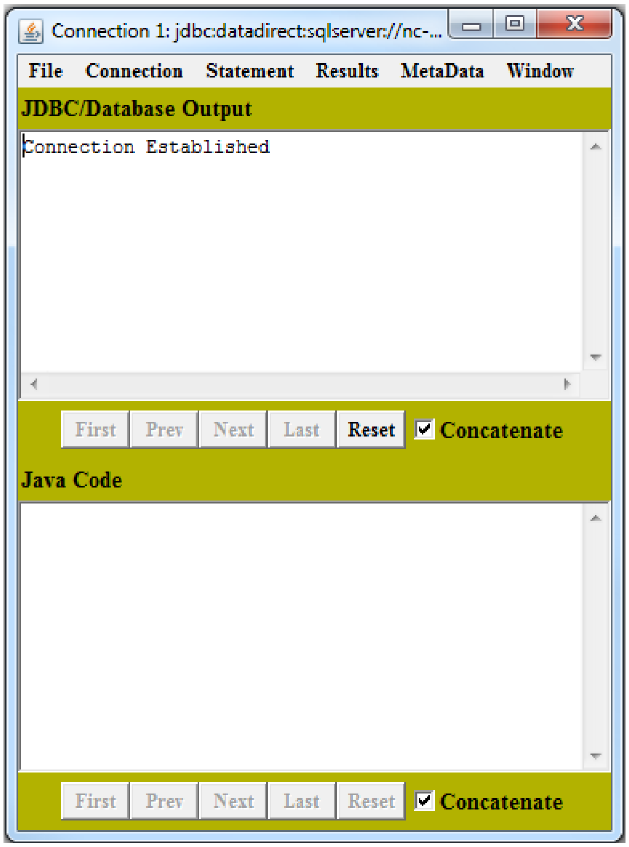
```
} catch (Exception e) {
   System.out.println(e);
}
```

3. From the **Test for JDBC Tool** window, select **Connection / Connect to DB**. The **Select A Database** window appears with a list of default connection URLs.
4. Select one of the default driver connection URLs. In the Database field, modify the default values of the connection URL appropriately for your environment.

Note: There are two entries for DB2: one with locationName and another with databaseName. If you are connecting to DB2 for Linux/UNIX/Windows, select the entry containing `databaseName`. If you are connecting to DB2 for z/OS or DB2 for i, select the entry containing `locationName`.

5. In the User Name and Password fields, type the values for the User and Password connection properties; then, click **Connect**. For information about JDBC connection properties, refer to your driver's connection property descriptions.
6. If the connection was successful, the **Connection** window appears and shows the `Connection Established` message in the JDBC/Database Output scroll box.

Important: For the Autonomous REST Connector: REST API data sources do not connect in the same manner as database servers; therefore; the `Connection Established` notification does not guarantee that the driver is properly configured. To confirm that the driver is correctly configured, you will need to retrieve data from an endpoint using the methods described in "Executing a simple database selection."



```
Connection Established
```

See also

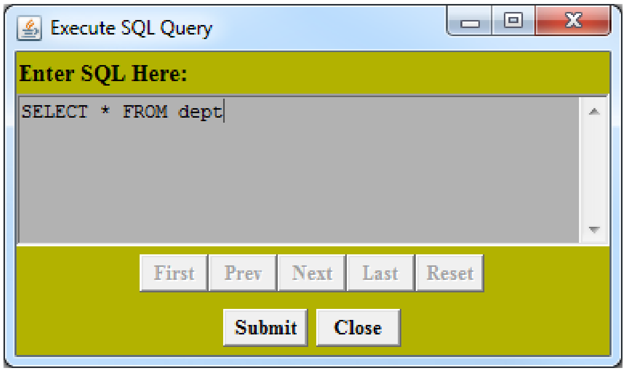
Executing a simple Select statement on page 152

Executing a simple Select statement

This example explains how to execute a simple Select statement and return the results.

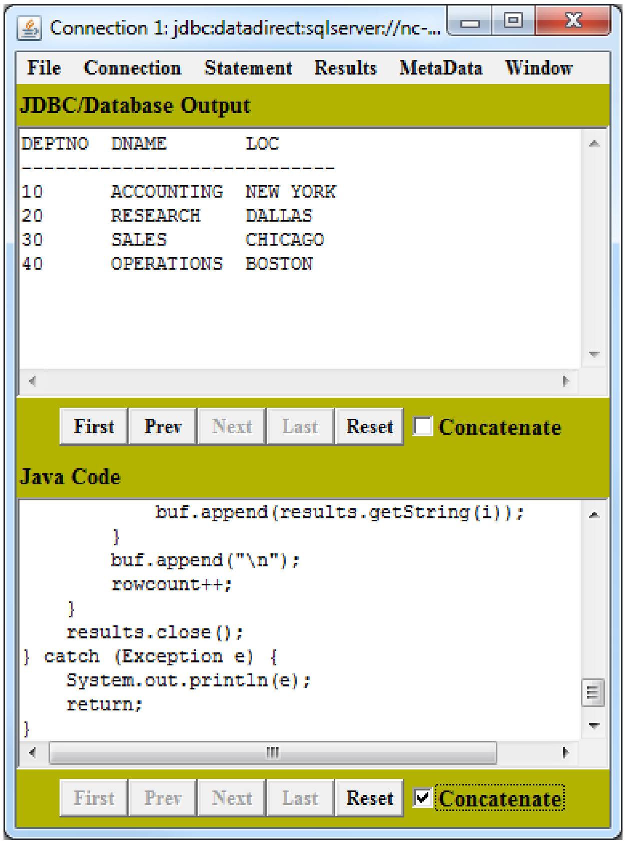
To Execute a Simple Select Statement:

1. From the **Connection** window menu, select **Connection / Create Statement**. The **Connection** window indicates that the creation of the statement was successful.
2. Select **Statement / Execute Stmt Query**. DataDirect Test displays a dialog box that prompts for a SQL statement.



```
SELECT * FROM dept
```

3. Type a Select statement and click **Submit**. Then, click **Close**.
4. Select **Results / Show All Results**. The data from your result set displays in the JDBC/Database Output scroll box.



```
DEPTNO  DNAME       LOC
----------------------------
10      ACCOUNTING  NEW YORK
20      RESEARCH    DALLAS
30      SALES       CHICAGO
40      OPERATIONS  BOSTON
```

```
            buf.append(results.getString(i));
        }
        buf.append("\n");
        rowcount++;
    }
    results.close();
} catch (Exception e) {
    System.out.println(e);
    return;
}
```

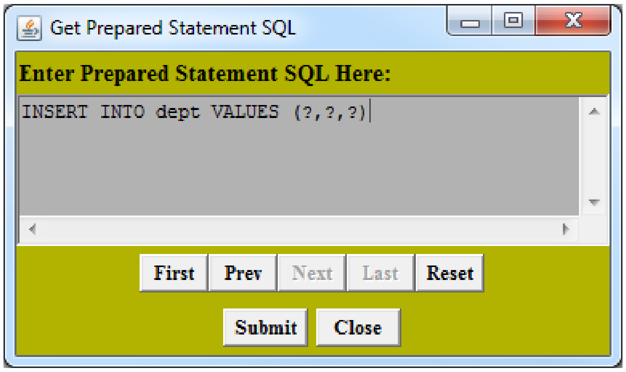
5. Scroll through the code in the Java Code scroll box to see which JDBC calls have been implemented by DataDirect Test.

Executing a prepared statement

This example explains how to execute a parameterized statement multiple times.

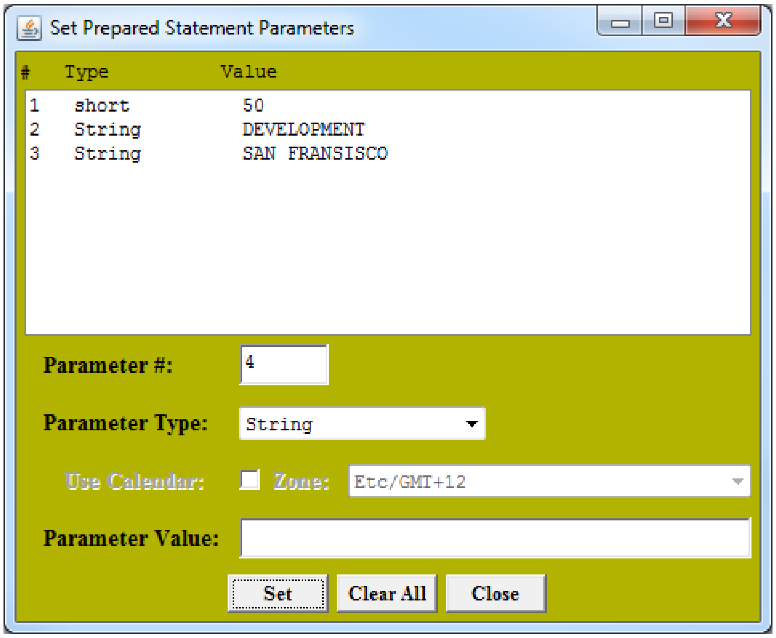
To execute a prepared statement:

1. From the **Connection** window menu, select **Connection / Create Prepared Statement**. DataDirect Test prompts you for a SQL statement.
2. Type an Insert statement and click **Submit**. Then, click **Close**.


Get Prepared Statement SQL
Enter Prepared Statement SQL Here:
First
Prev
Next
Last
Reset
Submit
Close

```
INSERT INTO dept VALUES (?,?,?)
```

3. Select **Statement / Set Prepared Parameters**. To set the value and type for each parameter:
   a) Type the parameter number.
   b) Select the parameter type.
   c) Type the parameter value.
   d) Click **Set** to pass this information to the JDBC driver.



#	Type	Value
1	short	50
2	String	DEVELOPMENT
3	String	SAN FRANSISCO

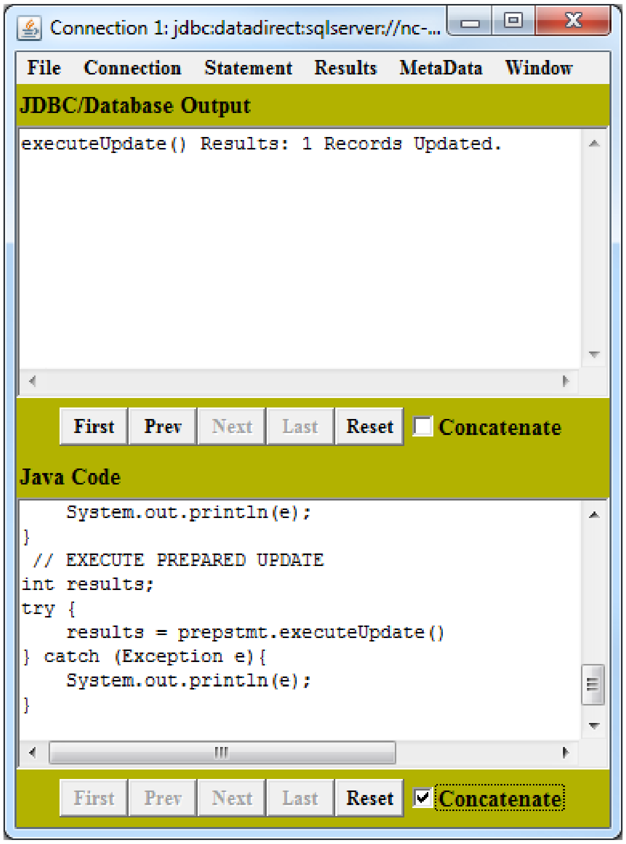
Parameter #: 4

Parameter Type: String

Use Calendar: □ Zone: Etc/GMT+12

Parameter Value:

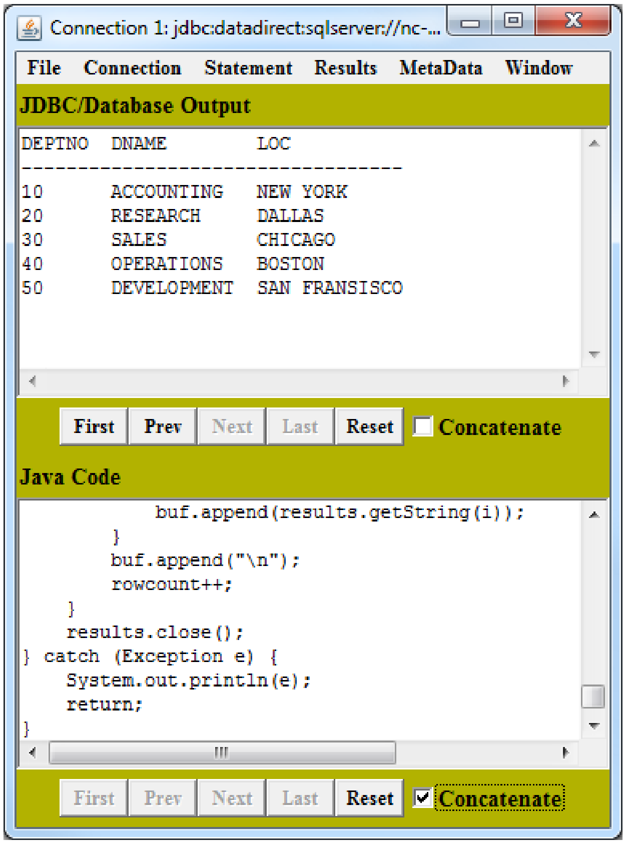
4. When you are finished, click **Close**.
5. Select **Statement / Execute Stmt Update**. The JDBC/Database Output scroll box indicates that one row has been inserted.


Connection 1: jdbc:datadirect:sqlserver://nc-...
File Connection Statement Results MetaData Window
JDBC/Database Output
First Prev Next Last Reset □ Concatenate
Java Code
First Prev Next Last Reset ☑ Concatenate

```
executeUpdate() Results: 1 Records Updated.
```

```
    System.out.println(e);
}
 // EXECUTE PREPARED UPDATE
int results;
try {
    results = prepstmt.executeUpdate()
} catch (Exception e){
    System.out.println(e);
}
```

6. If you want to insert multiple records, repeat Step 3 on page 154 and Step 5 on page 154 for each record.
7. If you repeat the steps described in "Executing a simple Select statement," you will see that the previously inserted records are also returned.



```
DEPTNO  DNAME        LOC
----------------------------------
10      ACCOUNTING   NEW YORK
20      RESEARCH     DALLAS
30      SALES        CHICAGO
40      OPERATIONS   BOSTON
50      DEVELOPMENT  SAN FRANSISCO
```

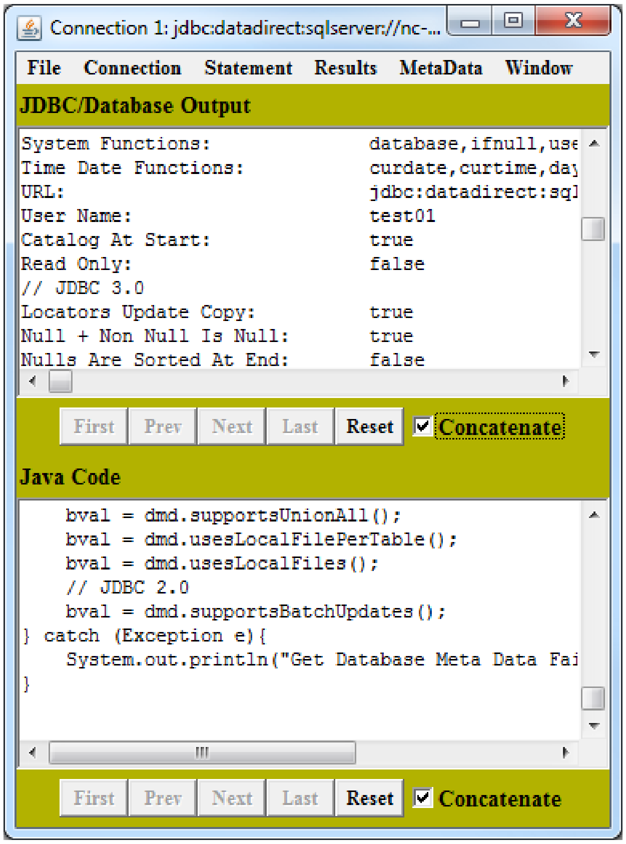
```
            buf.append(results.getString(i));
        }
        buf.append("\n");
        rowcount++;
    }
    results.close();
} catch (Exception e) {
    System.out.println(e);
    return;
}
```

See also

Executing a simple Select statement on page 152

Retrieving database metadata

1. From the **Connection** window menu, select **Connection / Get DB Meta Data**.
2. Select **MetaData / Show Meta Data**. Information about the JDBC driver and the database to which you are connected is returned.


Connection 1: jdbc:datadirect:sqlserver://nc-...
File Connection Statement Results MetaData Window
JDBC/Database Output
First Prev Next Last Reset ☑ Concatenate
Java Code
First Prev Next Last Reset ☑ Concatenate

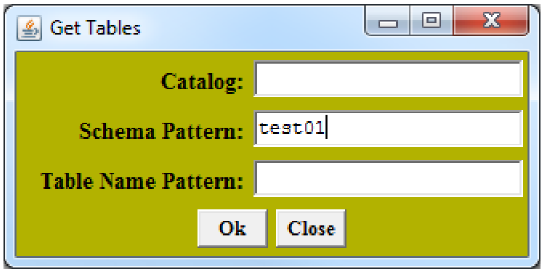
```
System Functions:                database,ifnull,use
Time Date Functions:             curdate,curtime,day
URL:                             jdbc:datadirect:sql
User Name:                       test01
Catalog At Start:                true
Read Only:                       false
// JDBC 3.0
Locators Update Copy:            true
Null + Non Null Is Null:         true
Nulls Are Sorted At End:         false
```

```
    bval = dmd.supportsUnionAll();
    bval = dmd.usesLocalFilePerTable();
    bval = dmd.usesLocalFiles();
    // JDBC 2.0
    bval = dmd.supportsBatchUpdates();
} catch (Exception e){
    System.out.println("Get Database Meta Data Fai
}
```

3. Scroll through the Java code in the Java Code scroll box to find out which JDBC calls have been implemented by DataDirect Test.

   Metadata also allows you to query the database catalog (enumerate the tables in the database, for example). In this example, we will query all tables with the schema pattern `test01`.

4. Select **MetaData / Tables**.
5. In the Schema Pattern field, type `test01`.

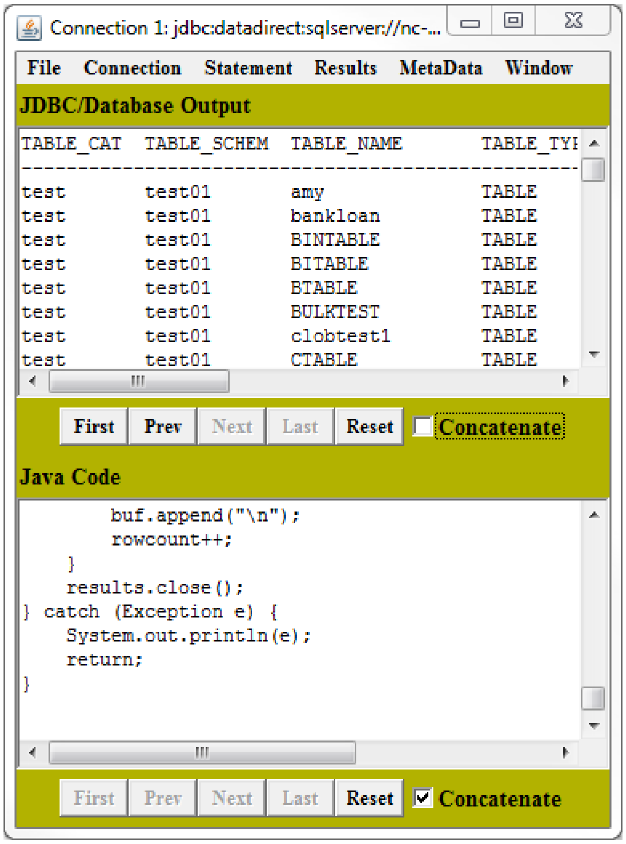


Catalog:

Schema Pattern: test01

Table Name Pattern:

6. Click **Ok**. The **Connection** window indicates that getTables() succeeded.
7. Select **Results / Show All Results**. All tables with a `test01` schema pattern are returned.


Connection 1: jdbc:datadirect:sqlserver://nc-...
File Connection Statement Results MetaData Window
JDBC/Database Output

```
TABLE_CAT  TABLE_SCHEM  TABLE_NAME        TABLE_TYI
--------------------------------------------------
test       test01       amy               TABLE
test       test01       bankloan          TABLE
test       test01       BINTABLE          TABLE
test       test01       BITABLE           TABLE
test       test01       BTABLE            TABLE
test       test01       BULKTEST          TABLE
test       test01       clobtest1         TABLE
test       test01       CTABLE            TABLE
```

First Prev Next Last Reset
□ Concatenate
Java Code

```
        buf.append("\n");
        rowcount++;
    }
    results.close();
} catch (Exception e) {
    System.out.println(e);
    return;
}
```

First Prev Next Last Reset
☑ Concatenate

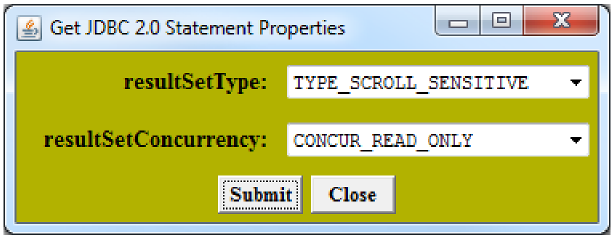
Scrolling through a result set

1. From the **Connection** window menu, select **Connection / Create JDBC 2.0 Statement**. DataDirect Test prompts for a result set type and concurrency.
2. Complete the following fields:

   a) In the resultSetType field, select **TYPE_SCROLL_SENSITIVE**.

   b) In the resultSetConcurrency field, select **CONCUR_READ_ONLY**.

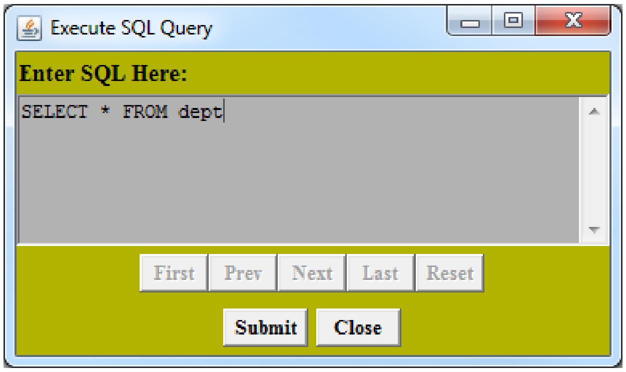
   c) Click **Submit**; then, click **Close**.



resultSetType: TYPE_SCROLL_SENSITIVE

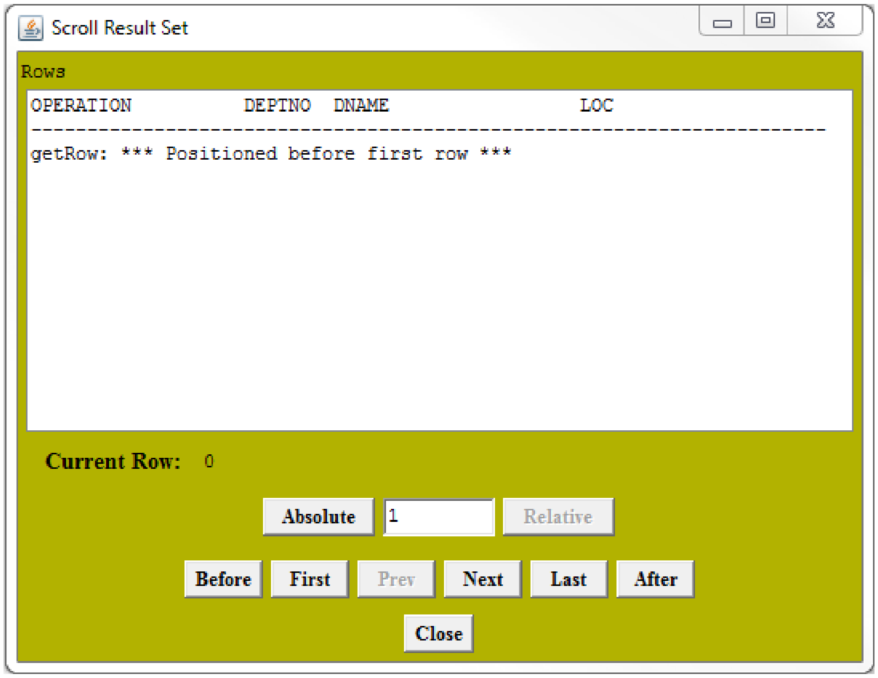
resultSetConcurrency: CONCUR_READ_ONLY

3. Select **Statement / Execute Stmt Query**.
4. Type a Select statement and click **Submit**. Then, click **Close**.



```
SELECT * FROM dept
```

5. Select **Results / Scroll Results**. The **Scroll Result Set** window indicates that the cursor is positioned before the first row.

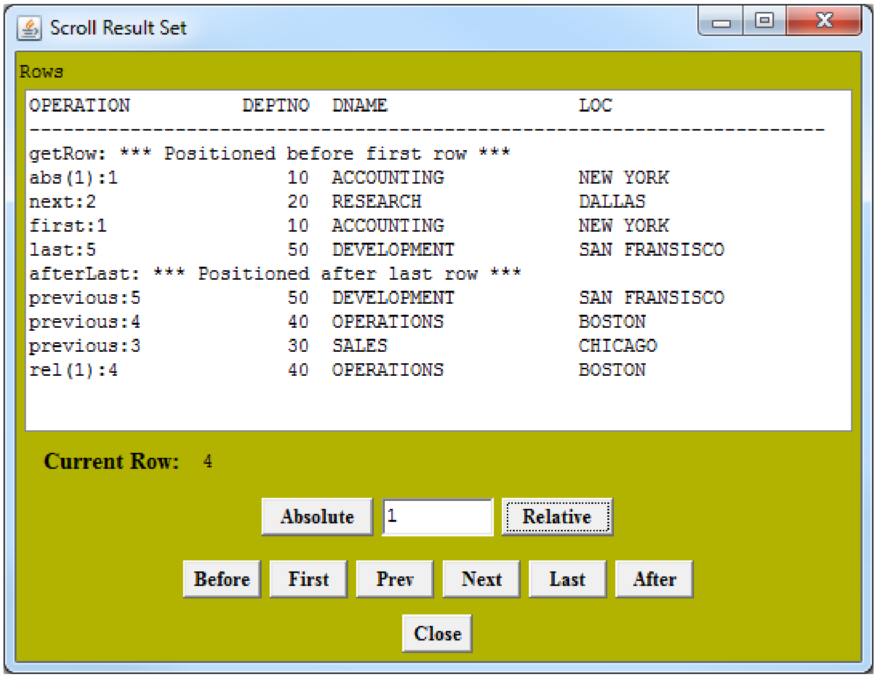


```
OPERATION          DEPTNO  DNAME                    LOC
-----------------------------------------------------------------------
getRow: *** Positioned before first row ***
```

Current Row: 0

Absolute: 1

6. Click the **Absolute**, **Relative**, **Before**, **First**, **Prev**, **Next**, **Last**, and **After** buttons as appropriate to navigate through the result set. After each action, the **Scroll Result Set** window displays the data at the current position of the cursor.



```
OPERATION          DEPTNO  DNAME                 LOC
------------------------------------------------------------------------
getRow: *** Positioned before first row ***
abs(1):1               10  ACCOUNTING            NEW YORK
next:2                 20  RESEARCH              DALLAS
first:1                10  ACCOUNTING            NEW YORK
last:5                 50  DEVELOPMENT           SAN FRANSISCO
afterLast: *** Positioned after last row ***
previous:5             50  DEVELOPMENT           SAN FRANSISCO
previous:4             40  OPERATIONS            BOSTON
previous:3             30  SALES                 CHICAGO
rel(1):4               40  OPERATIONS            BOSTON
```

Current Row: 4

Relative: 1

7. Click **Close**.

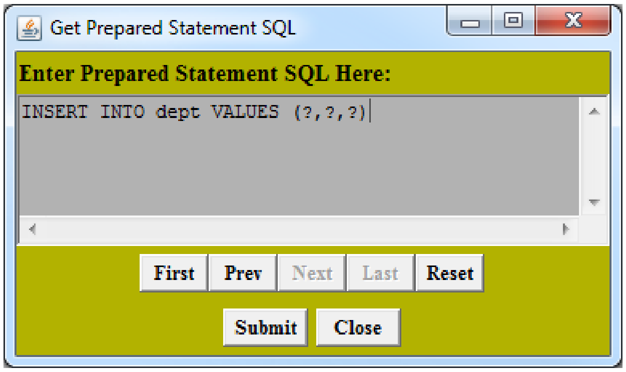
Batch execution on a prepared statement

Batch execution on a prepared statement allows you to update or insert multiple records simultaneously. In some cases, this can significantly improve system performance because fewer round trips to the database are required.

To execute a batch on a prepared statement:

1. From the **Connection** window menu, select **Connection / Create Prepared Statement**.

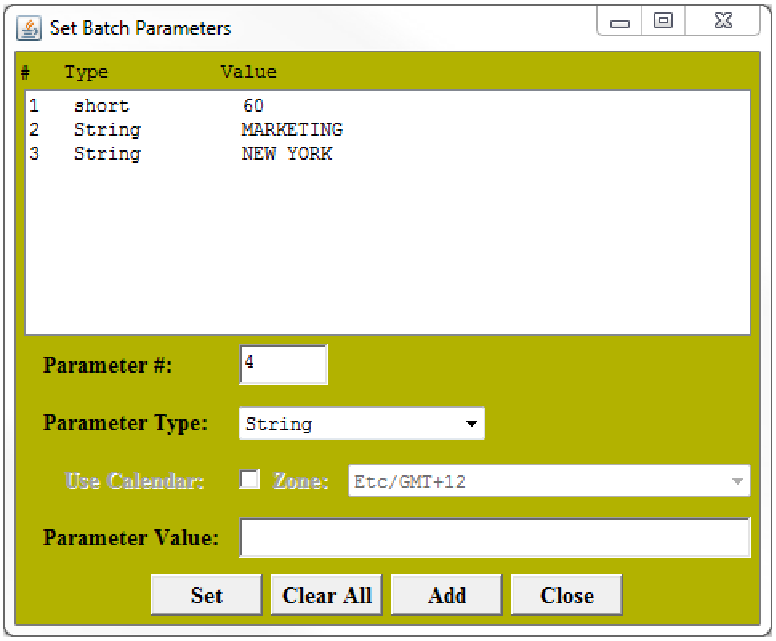
   Type an Insert statement and click **Submit**. Then, click **Close**.



Enter Prepared Statement SQL Here:

```
INSERT INTO dept VALUES (?,?,?)
```

2. Select **Statement / Add Stmt Batch**.
3. For each parameter:
   a) Type the parameter number.
   b) Select the parameter type.
   c) Type the parameter value.
   d) Click **Set**.



#	Type	Value
1	short	60
2	String	MARKETING
3	String	NEW YORK

Parameter #: 4

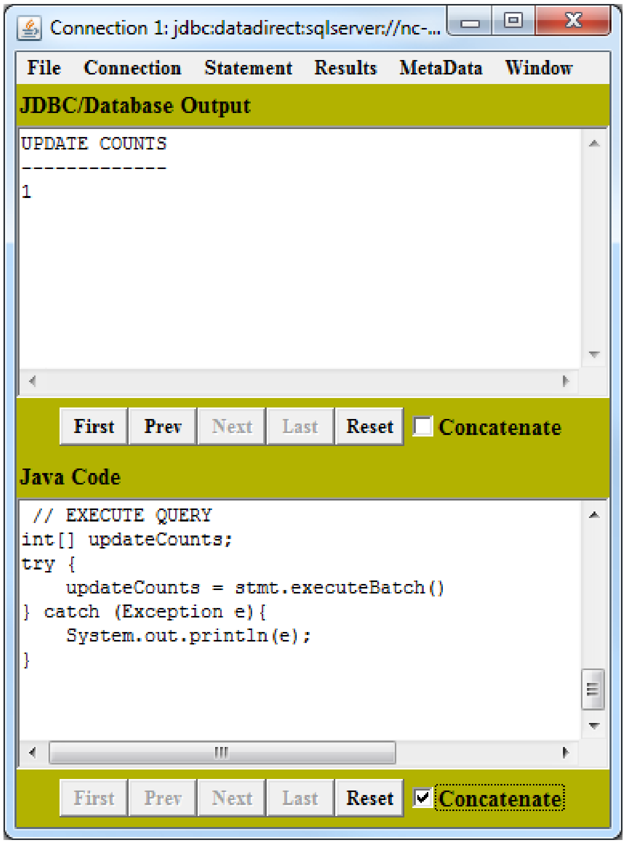
Parameter Type: String

Use Calendar: □ Zone: Etc/GMT+12

Parameter Value:

4. Click **Add** to add the specified set of parameters to the batch. To add multiple parameter sets to the batch, repeat Step 2 on page 160 through Step 4 on page 161 as many times as necessary. When you are finished adding parameter sets to the batch, click **Close**.

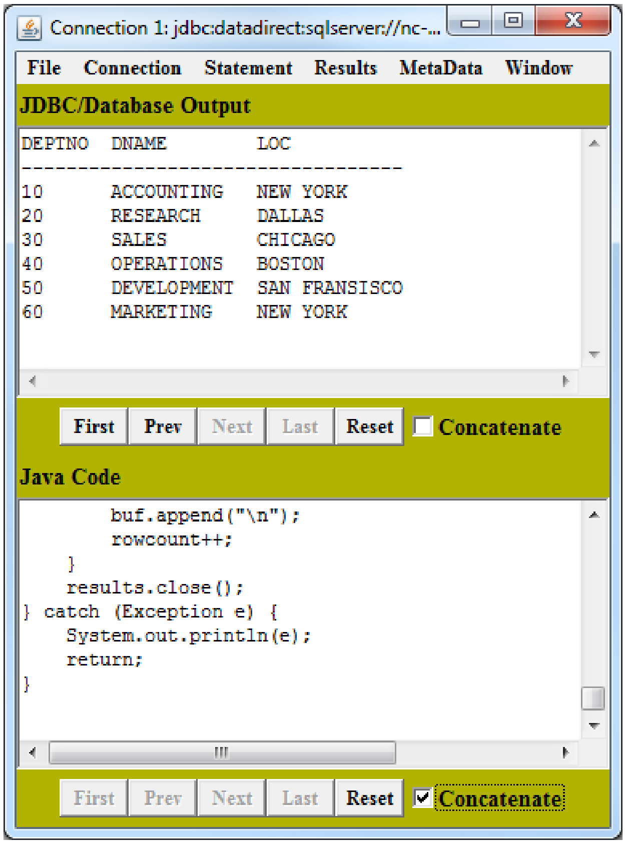
5. Select **Statement** / **Execute Stmt Batch**. DataDirect Test displays the rowcount for each of the elements in the batch.



```
UPDATE COUNTS
-------------
1
```

```
 // EXECUTE QUERY
int[] updateCounts;
try {
    updateCounts = stmt.executeBatch()
} catch (Exception e){
    System.out.println(e);
}
```

6. If you re-execute the Select statement from "Executing a simple Select statement," you see that the previously inserted records are returned.



```
DEPTNO  DNAME        LOC
----------------------------------
10      ACCOUNTING   NEW YORK
20      RESEARCH     DALLAS
30      SALES        CHICAGO
40      OPERATIONS   BOSTON
50      DEVELOPMENT  SAN FRANSISCO
60      MARKETING    NEW YORK
```

```
        buf.append("\n");
        rowcount++;
    }
    results.close();
} catch (Exception e) {
    System.out.println(e);
    return;
}
```

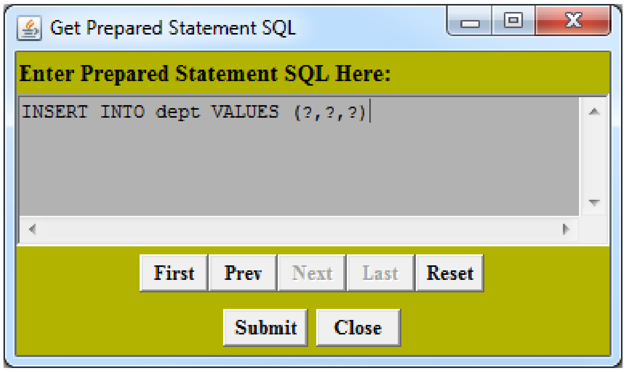
See also

Executing a simple Select statement on page 152

Returning parameter metadata

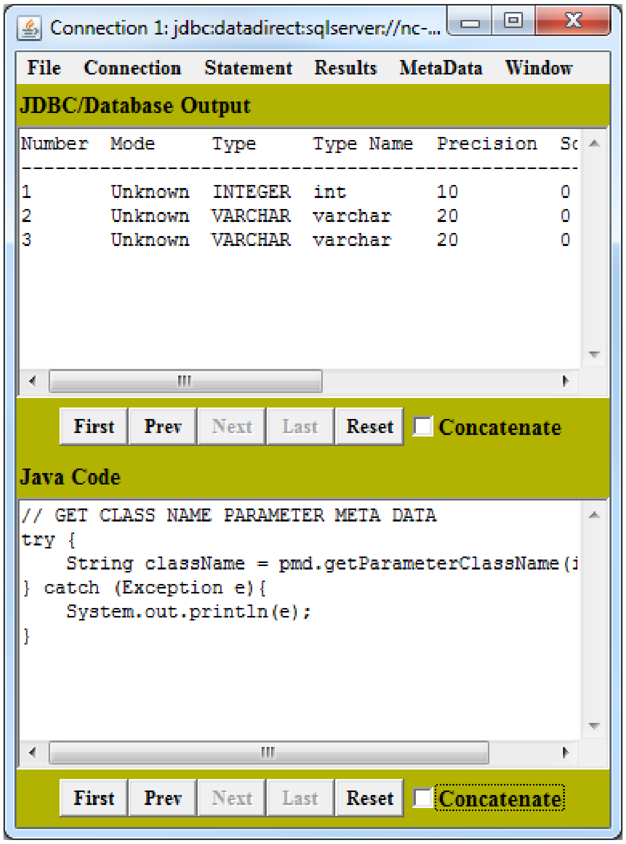
1. From the **Connection** window menu, select **Connection** / **Create Prepared Statement**.

   Type the prepared statement and click **Submit**. Then, click **Close**.


Get Prepared Statement SQL
Enter Prepared Statement SQL Here:
First
Prev
Next
Last
Reset
Submit
Close

```
INSERT INTO dept VALUES (?,?,?)
```

2. Select **Statement** / **Get ParameterMetaData**. The **Connection** window displays parameter metadata.

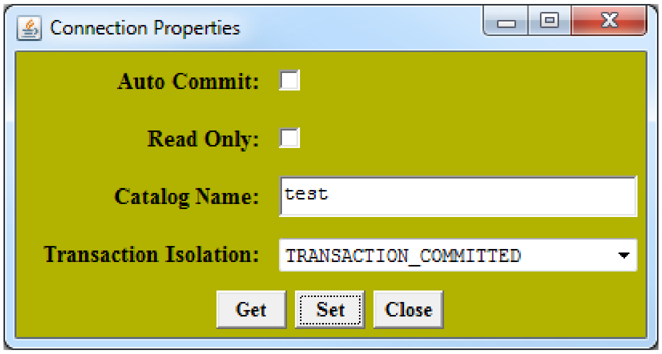


```
Number  Mode     Type     Type Name  Precision  Sc
------------------------------------------------------
1       Unknown  INTEGER  int        10         0
2       Unknown  VARCHAR  varchar    20         0
3       Unknown  VARCHAR  varchar    20         0
```

```
// GET CLASS NAME PARAMETER META DATA
try {
    String className = pmd.getParameterClassName(i
} catch (Exception e){
    System.out.println(e);
}
```

Establishing savepoints

1. From the **Connection** window menu, select **Connection** / **Connection Properties**.
2. Select **TRANSACTION_COMMITTED** from the Transaction Isolation drop-down list. Do not select the Auto Commit check box.



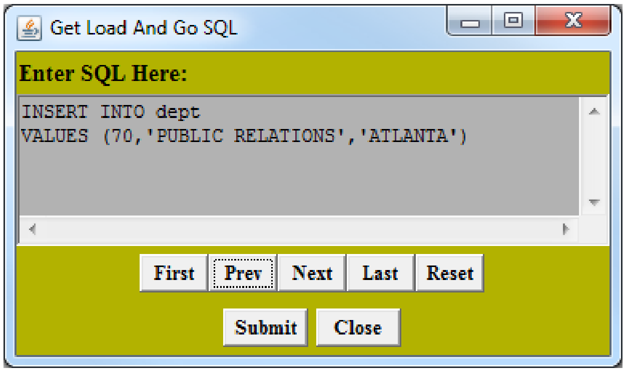
Auto Commit: □

Read Only: □

Catalog Name: test

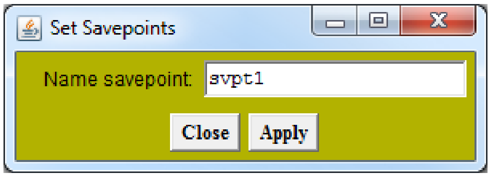
Transaction Isolation: TRANSACTION_COMMITTED

3. Click **Set**; then, click **Close**.
4. From the **Connection** window menu, select **Connection** / **Load and Go**. The **Get Load And Go SQL** window appears.
5. Type a statement and click **Submit**.



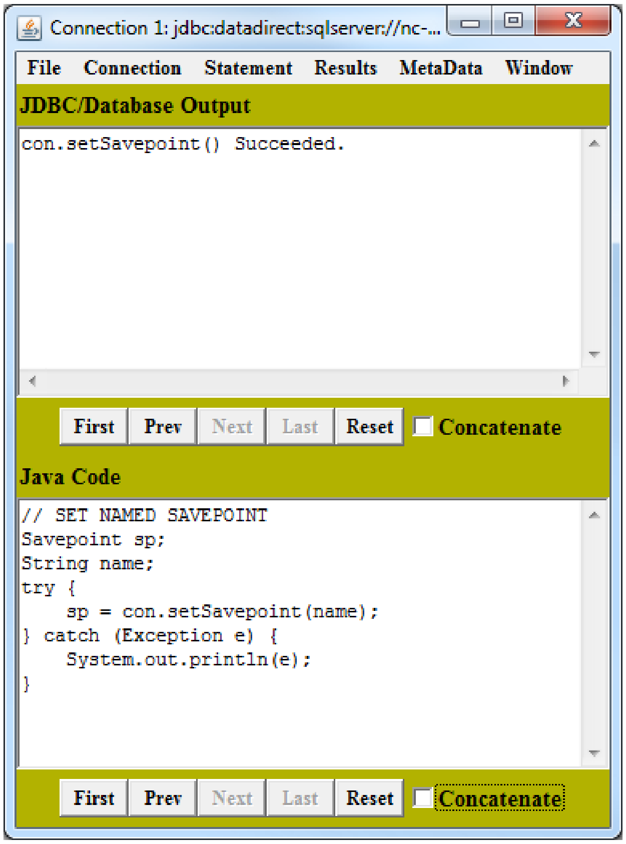
```
INSERT INTO dept
VALUES (70,'PUBLIC RELATIONS','ATLANTA')
```

6. Select **Connection** / **Set Savepoint**.
7. In the **Set Savepoints** window, type a savepoint name.



Name savepoint: svpt1

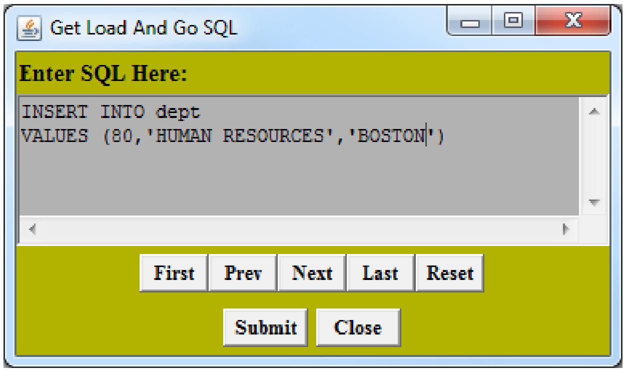
8. Click **Apply**; then, click **Close**. The **Connection** window indicates whether or not the savepoint succeeded.



```
con.setSavepoint() Succeeded.
```

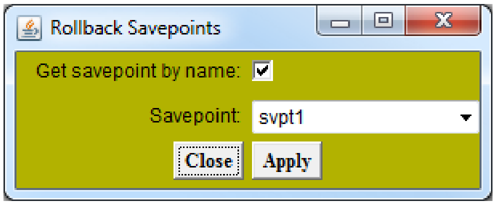
```
// SET NAMED SAVEPOINT
Savepoint sp;
String name;
try {
    sp = con.setSavepoint(name);
} catch (Exception e) {
    System.out.println(e);
}
```

9. Return to the **Get Load And Go SQL** window and specify another statement. Click **Submit**.



```
INSERT INTO dept
VALUES (80,'HUMAN RESOURCES','BOSTON')
```

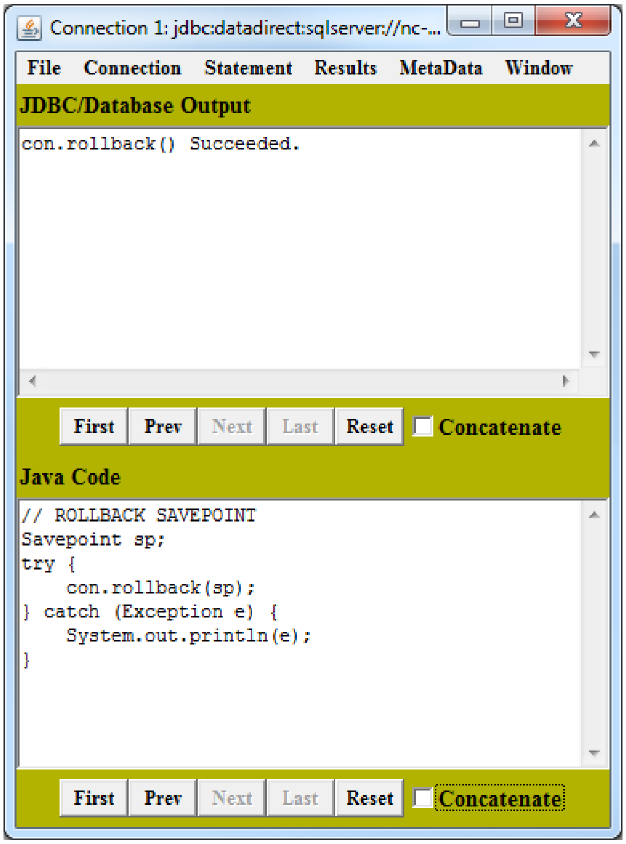
10. Select **Connection** / **Rollback Savepoint**. In the **Rollback Savepoints** window, specify the savepoint name.



Get savepoint by name: ☑

Savepoint: svpt1

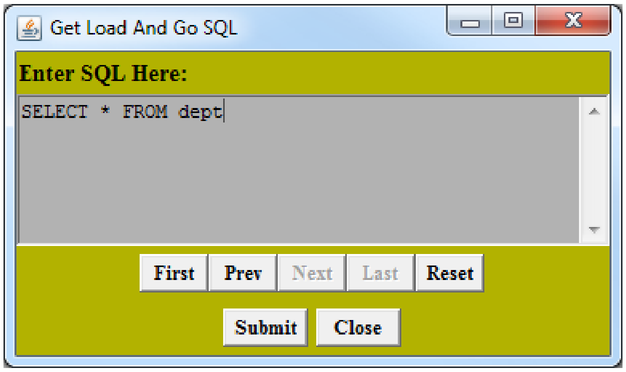
11. Click **Apply**; then, click **Close**. The **Connection** window indicates whether or not the savepoint rollback succeeded.



```
con.rollback() Succeeded.
```

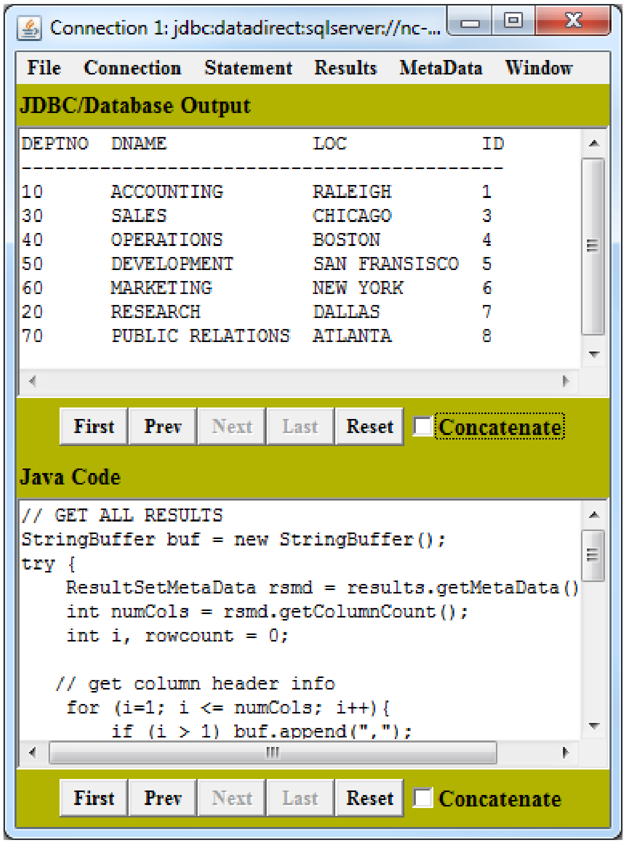
```
// ROLLBACK SAVEPOINT
Savepoint sp;
try {
    con.rollback(sp);
} catch (Exception e) {
    System.out.println(e);
}
```

12. Return to the **Get Load And Go SQL** window and specify another statement.



```
SELECT * FROM dept
```

Click **Submit**; then, click **Close**. The **Connection** window displays the data inserted before the first savepoint. The second insert was rolled back.



```
DEPTNO  DNAME            LOC            ID
-------------------------------------------
10      ACCOUNTING       RALEIGH        1
30      SALES            CHICAGO        3
40      OPERATIONS       BOSTON         4
50      DEVELOPMENT      SAN FRANSISCO  5
60      MARKETING        NEW YORK       6
20      RESEARCH         DALLAS         7
70      PUBLIC RELATIONS ATLANTA        8
```

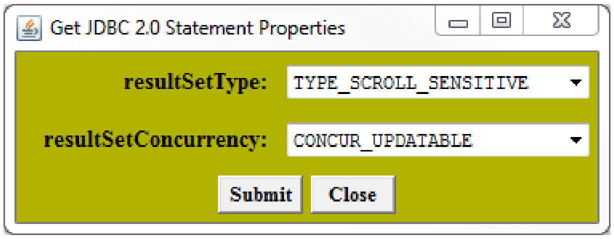
```
// GET ALL RESULTS
StringBuffer buf = new StringBuffer();
try {
    ResultSetMetaData rsmd = results.getMetaData()
    int numCols = rsmd.getColumnCount();
    int i, rowcount = 0;

   // get column header info
    for (i=1; i <= numCols; i++){
        if (i > 1) buf.append(",");
```

Updatable result sets

The following examples explain the concept of updatable result sets by deleting, inserting, and updating a row.

Deleting a row

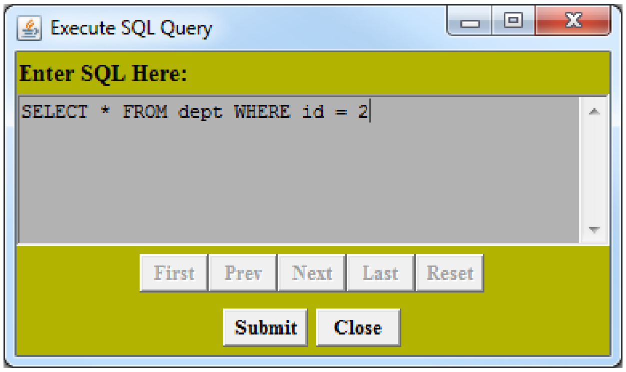
1. From the **Connection** window menu, select **Connection** / **Create JDBC 2.0 Statement**.
2. Complete the following fields:
   a) In the resultSetType field, select **TYPE_SCROLL_SENSITIVE**.
   b) In the resultSetConcurrency field, select **CONCUR_UPDATABLE**.


Get JDBC 2.0 Statement Properties

resultSetType: TYPE_SCROLL_SENSITIVE

resultSetConcurrency: CONCUR_UPDATABLE

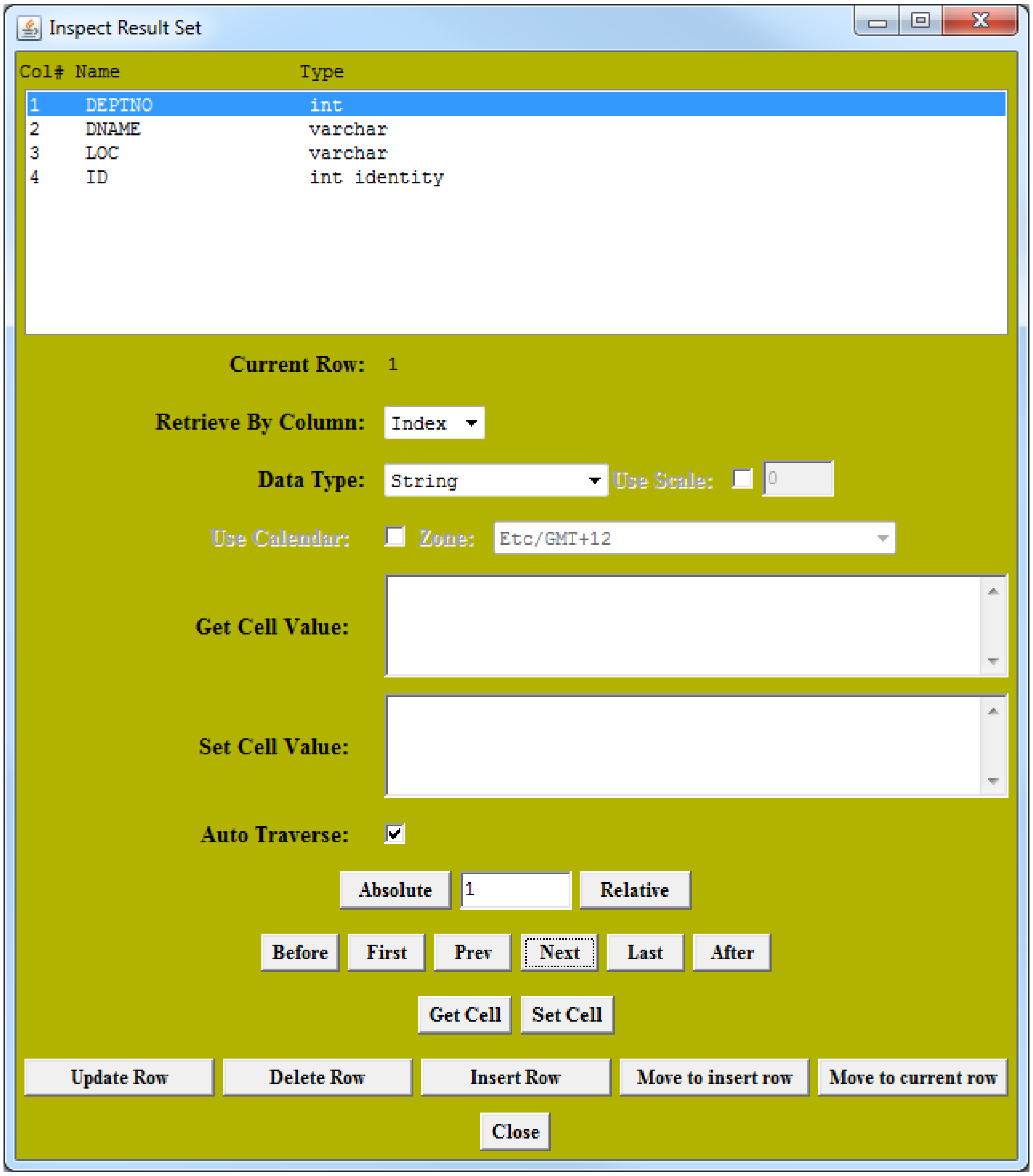
3. Click **Submit**; then, click **Close**.
4. Select **Statement** / **Execute Stmt Query**.
5. Specify the Select statement and click **Submit**. Then, click **Close**.


Execute SQL Query

Enter SQL Here:

```
SELECT * FROM dept WHERE id = 2
```

6. Select **Results** / **Inspect Results**. The **Inspect Result Set** window appears.



Inspect Result Set

Col#	Name	Type
1	DEPTNO	int
2	DNAME	varchar
3	LOC	varchar
4	ID	int identity

Current Row: 1

Retrieve By Column: Index

Data Type: String Use Scale: ☐ 0

Use Calendar: ☐ Zone: Etc/GMT+12

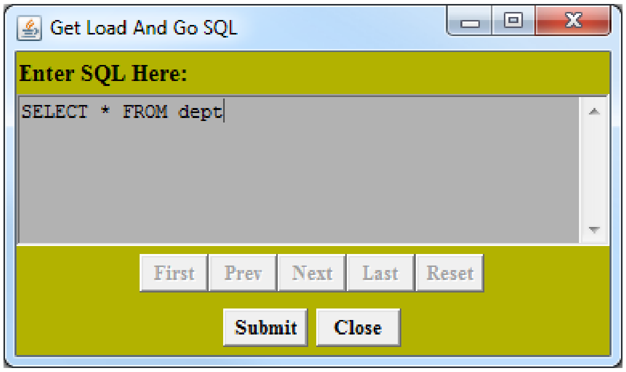
Get Cell Value:

Set Cell Value:

Auto Traverse: ☑

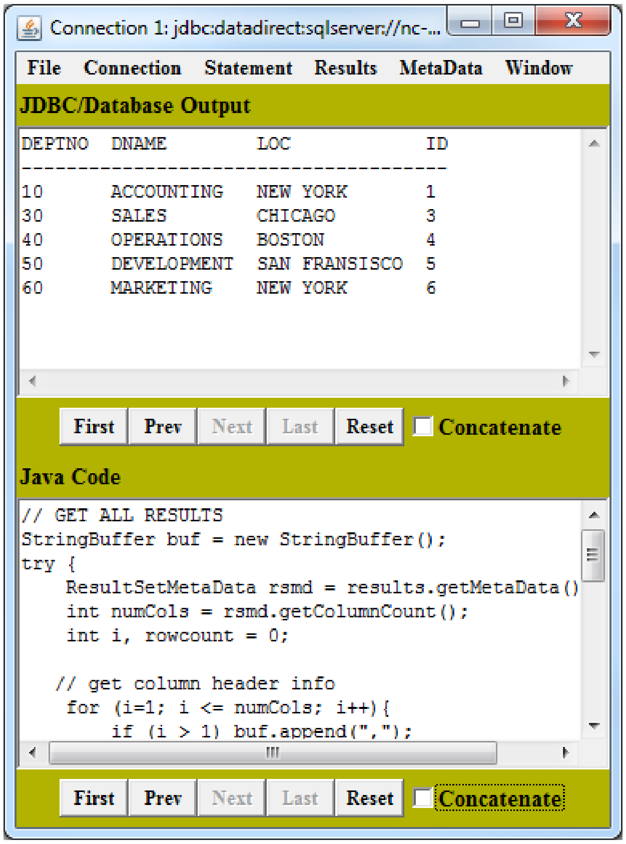
Absolute 1 Relative

7. Click **Next**. Current Row changes to 1.
8. Click **Delete Row**.
9. To verify the result, return to the Connection menu and select **Connection** / **Load and Go**. The **Get Load And Go SQL** window appears.
10. Specify the statement that you want to execute and click **Submit**. Then, click **Close**.



```
SELECT * FROM dept
```

11. The **Connection** window shows that the row has been deleted.

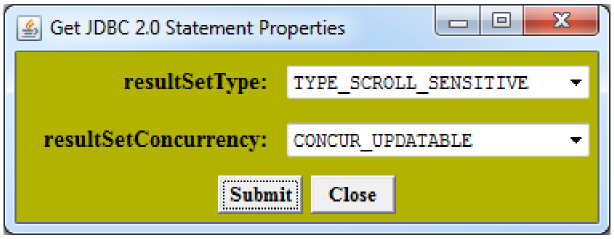


```
DEPTNO  DNAME         LOC            ID
--------------------------------------
10      ACCOUNTING    NEW YORK       1
30      SALES         CHICAGO        3
40      OPERATIONS    BOSTON         4
50      DEVELOPMENT   SAN FRANSISCO  5
60      MARKETING     NEW YORK       6
```

```
// GET ALL RESULTS
StringBuffer buf = new StringBuffer();
try {
    ResultSetMetaData rsmd = results.getMetaData()
    int numCols = rsmd.getColumnCount();
    int i, rowcount = 0;

   // get column header info
    for (i=1; i <= numCols; i++){
        if (i > 1) buf.append(",");
```

Inserting a row

1. From the **Connection** window menu, select **Connection** / **Create JDBC 2.0 Statement**.
2. Complete the following fields:
   a) In the resultSetType field, select **TYPE_SCROLL_SENSITIVE**.
   b) In the resultSetConcurrency field, select **CONCUR_UPDATABLE**.

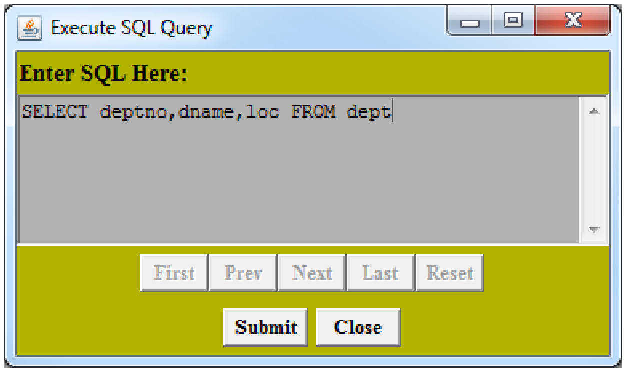


Get JDBC 2.0 Statement Properties

resultSetType: TYPE_SCROLL_SENSITIVE

resultSetConcurrency: CONCUR_UPDATABLE

3. Click **Submit**; then, click **Close**.
4. Select **Statement** / **Execute Stmt Query**.
5. Specify the Select statement that you want to execute and click **Submit**. Then, click **Close**.

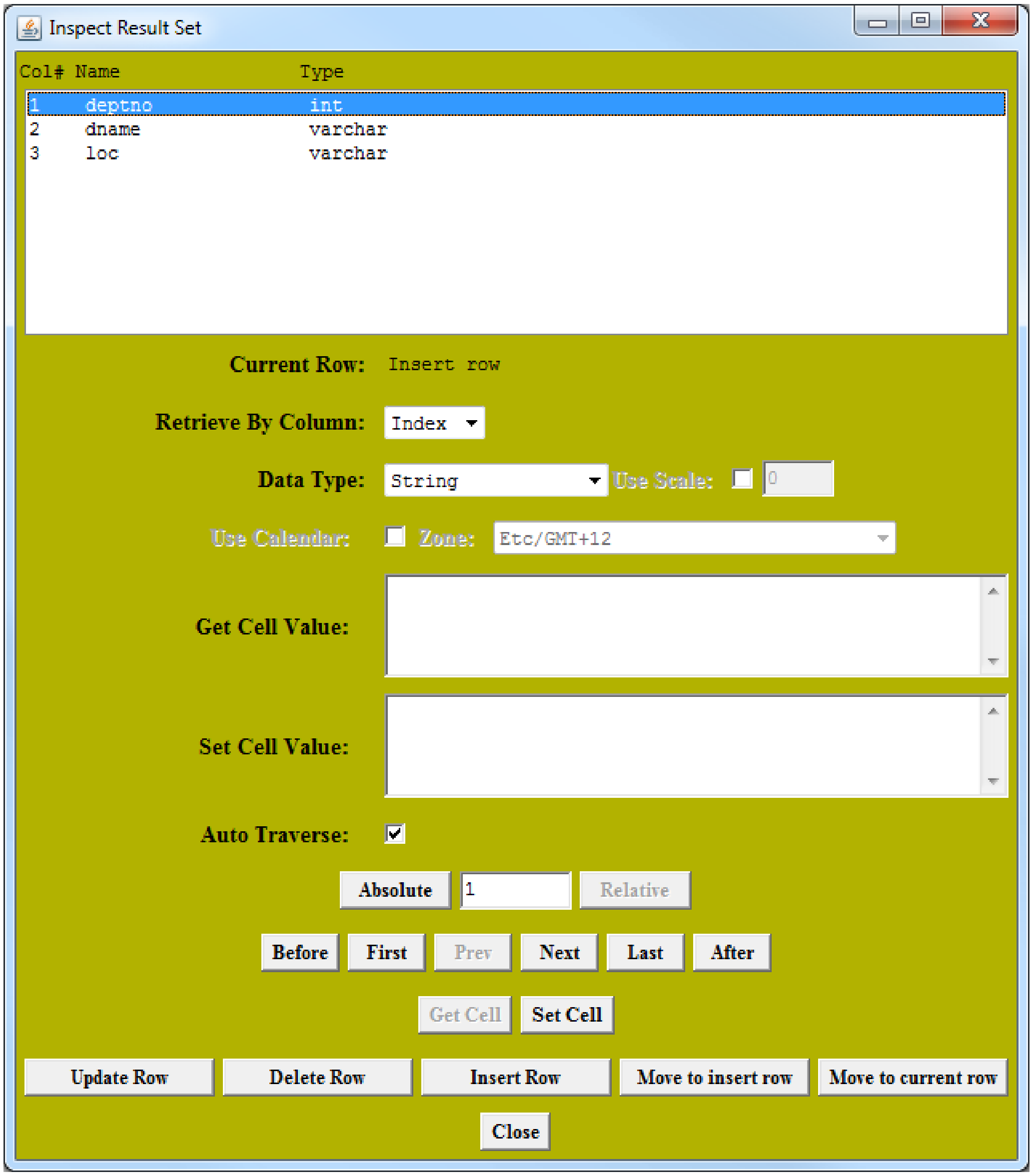


Execute SQL Query

Enter SQL Here:

```
SELECT deptno,dname,loc FROM dept
```

6. Select **Results** / **Inspect Results**. The **Inspect Result Set** window appears.



Col#	Name	Type
1	deptno	int
2	dname	varchar
3	loc	varchar

Current Row: Insert row

Retrieve By Column: Index

Data Type: String Use Scale: □ 0

Use Calendar: □ Zone: Etc/GMT+12

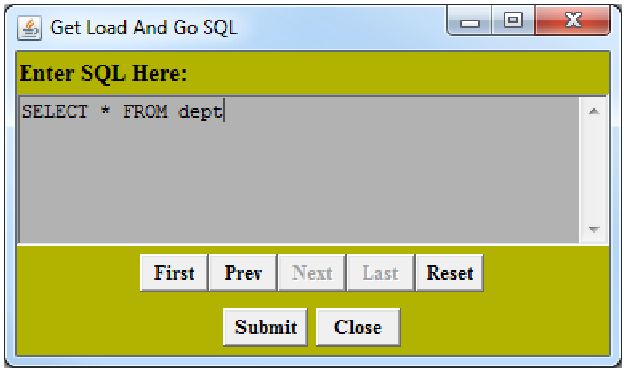
Get Cell Value:

Set Cell Value:

Auto Traverse: ☑

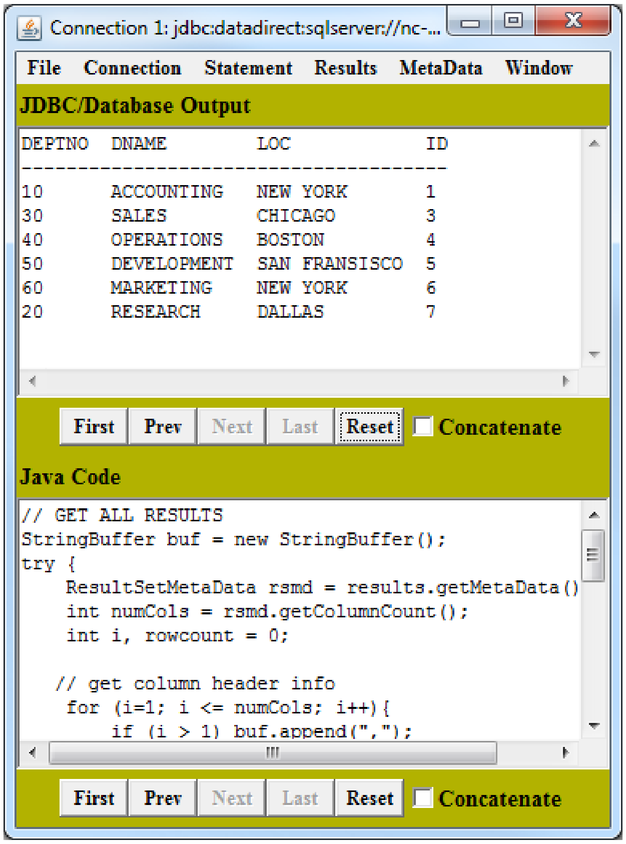
Absolute: 1

7. Click **Move to insert row**; Current Row is now Insert row.
8. Change Data Type to int. In Set Cell Value, enter `20`. Click **Set Cell**.
9. Select the second row in the top pane. Change the Data Type to String. In Set Cell Value, enter `RESEARCH`. Click **Set Cell**.
10. Select the third row in the top pane. In Set Cell Value, enter `DALLAS`. Click **Set Cell**.
11. Click **Insert Row**.
12. To verify the result, return to the Connection menu and select **Connection** / **Load and Go**. The **Get Load And Go SQL** window appears.
13. Specify the statement that you want to execute and click **Submit**. Then, click **Close**.



```
SELECT * FROM dept
```

14. The **Connection** window shows the newly inserted row.



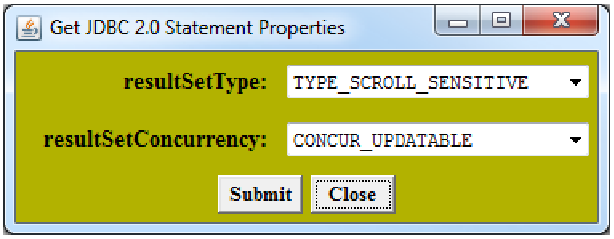
```
DEPTNO  DNAME        LOC            ID
--------------------------------------
10      ACCOUNTING   NEW YORK       1
30      SALES        CHICAGO        3
40      OPERATIONS   BOSTON         4
50      DEVELOPMENT  SAN FRANSISCO  5
60      MARKETING    NEW YORK       6
20      RESEARCH     DALLAS         7
```

```
// GET ALL RESULTS
StringBuffer buf = new StringBuffer();
try {
    ResultSetMetaData rsmd = results.getMetaData()
    int numCols = rsmd.getColumnCount();
    int i, rowcount = 0;

   // get column header info
    for (i=1; i <= numCols; i++){
        if (i > 1) buf.append(",");
```

Caution: The ID will be 3 for the row you just inserted because it is an auto increment column.

Updating a row

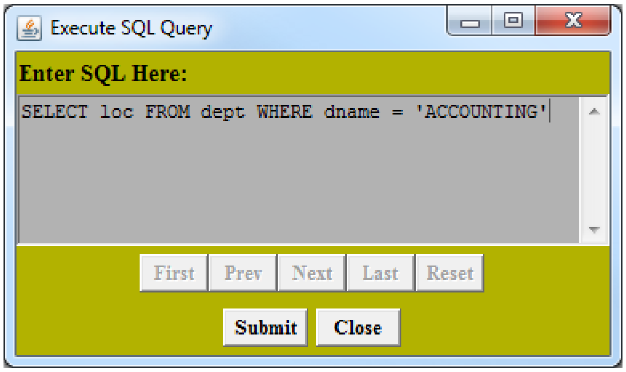
1. From the **Connection** window menu, select **Connection** / **Create JDBC 2.0 Statement**.
2. Complete the following fields:
   a) In the resultSetType field, select **TYPE_SCROLL_SENSITIVE**.
   b) In the resultSetConcurrency field, select **CONCUR_UPDATABLE**.



resultSetType: TYPE_SCROLL_SENSITIVE

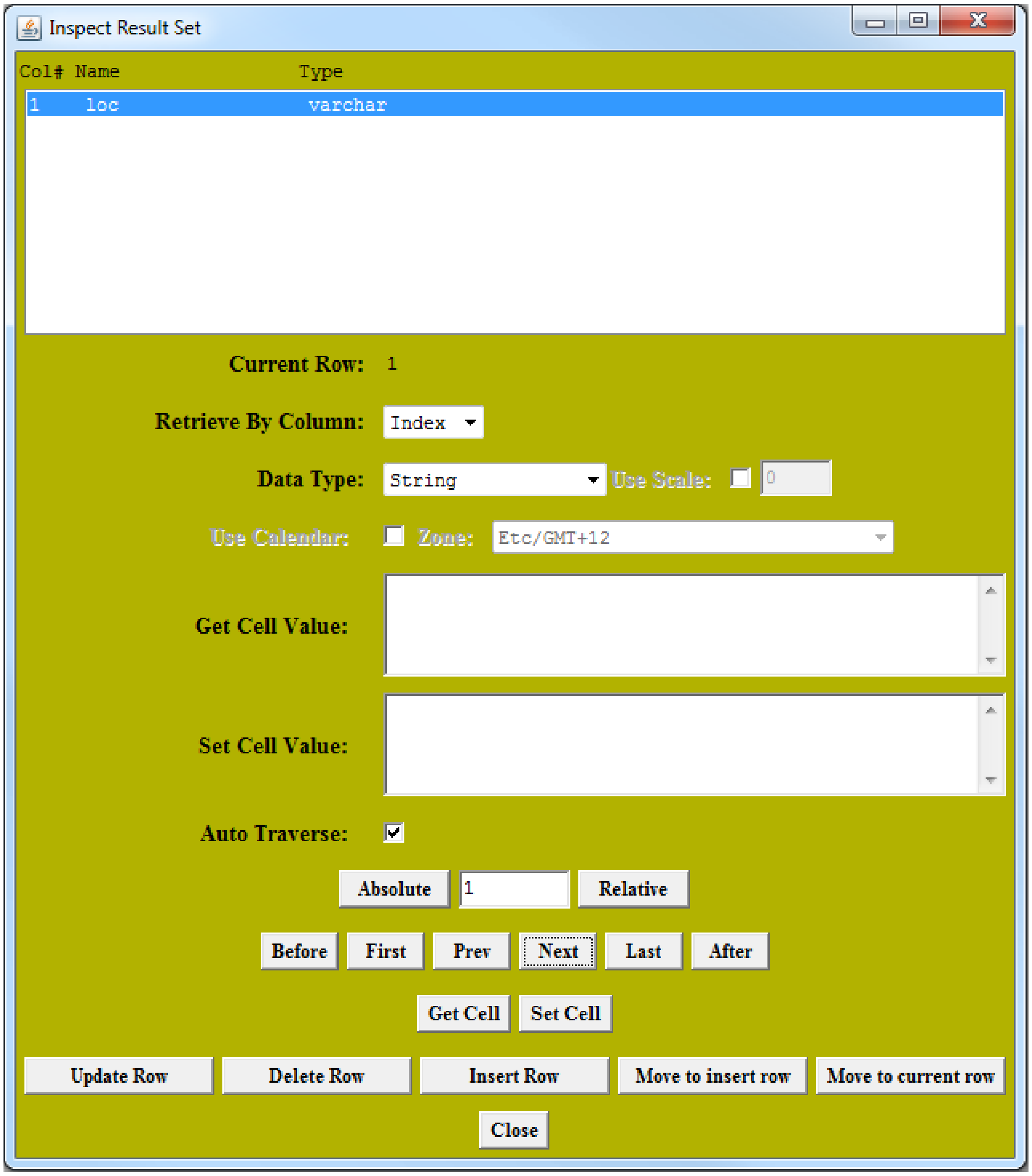
resultSetConcurrency: CONCUR_UPDATABLE

3. Click **Submit**; then, click **Close**.
4. Select **Statement** / **Execute Stmt Query**.
5. Specify the Select statement that you want to execute.



```
SELECT loc FROM dept WHERE dname = 'ACCOUNTING'
```

6. Click **Submit**; then, click **Close**.
7. Select **Results** / **Inspect Results**. The **Inspect Result Set** window appears.



Col#	Name	Type
1	loc	varchar

Current Row: 1

Retrieve By Column: Index

Data Type: String Use Scale: □ 0

Use Calendar: □ Zone: Etc/GMT+12

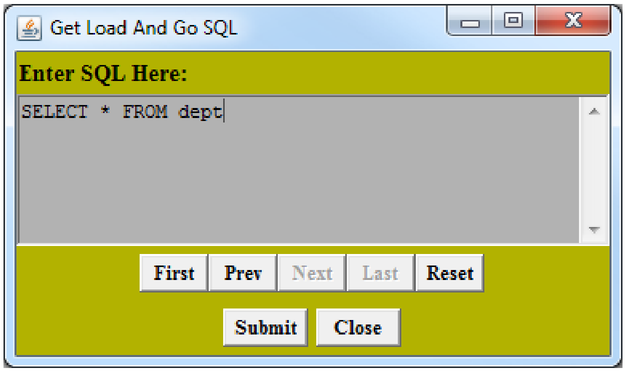
Get Cell Value:

Set Cell Value:

Auto Traverse: ☑

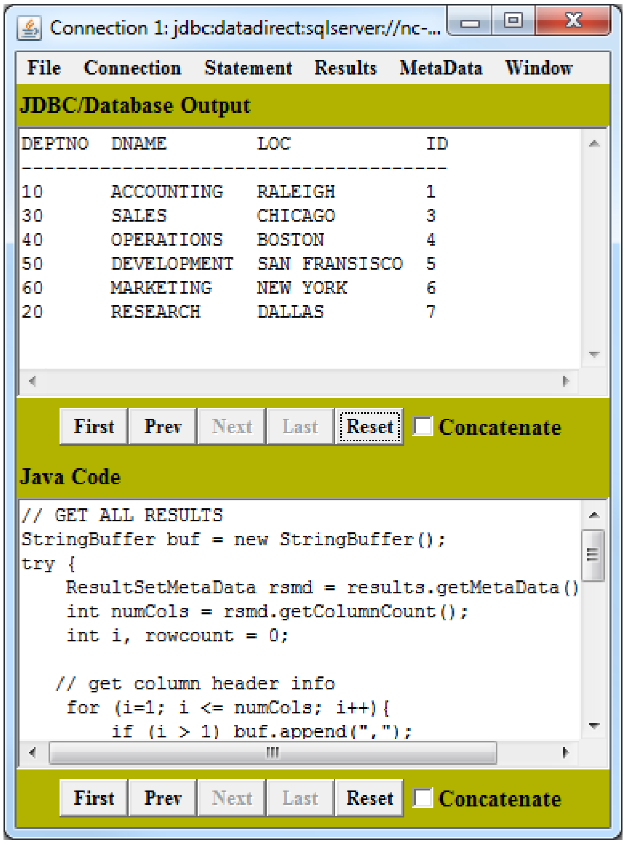
Absolute 1 Relative

8. Click **Next**. Current Row changes to 1.
9. In Set Cell Value, type `RALEIGH`. Then, click **Set Cell**.
10. Click **Update Row**.
11. To verify the result, return to the Connection menu and select **Connection** / **Load and Go**. The **Get Load And Go SQL** window appears.
12. Specify the statement that you want to execute.



```
SELECT * FROM dept
```

13. Click **Submit**; then, click **Close**.
14. The **Connection** window shows LOC for accounting changed from NEW YORK to RALEIGH.



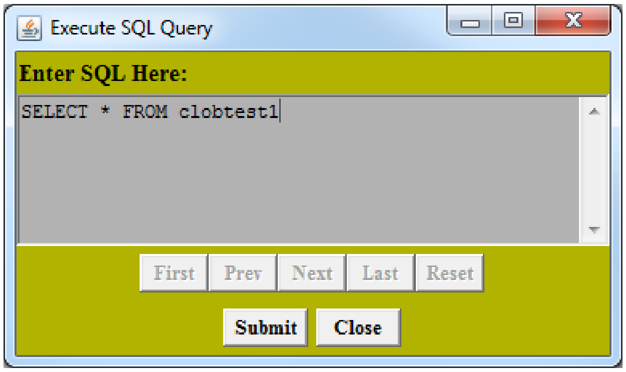
```
DEPTNO  DNAME        LOC            ID
--------------------------------------
10      ACCOUNTING   RALEIGH        1
30      SALES        CHICAGO        3
40      OPERATIONS   BOSTON         4
50      DEVELOPMENT  SAN FRANSISCO  5
60      MARKETING    NEW YORK       6
20      RESEARCH     DALLAS         7
```

```
// GET ALL RESULTS
StringBuffer buf = new StringBuffer();
try {
    ResultSetMetaData rsmd = results.getMetaData()
    int numCols = rsmd.getColumnCount();
    int i, rowcount = 0;

   // get column header info
    for (i=1; i <= numCols; i++){
        if (i > 1) buf.append(",");
```

Retrieving large object (LOB) data

The following example uses Clob data; however, this procedure also applies to Blob data. This example illustrates only one of multiple ways in which LOB data can be processed.

1. From the **Connection** window menu, select **Connection** / **Create Statement**.
2. Select **Statement** / **Execute Stmt Query**.
3. Specify the Select statement that you want to execute.

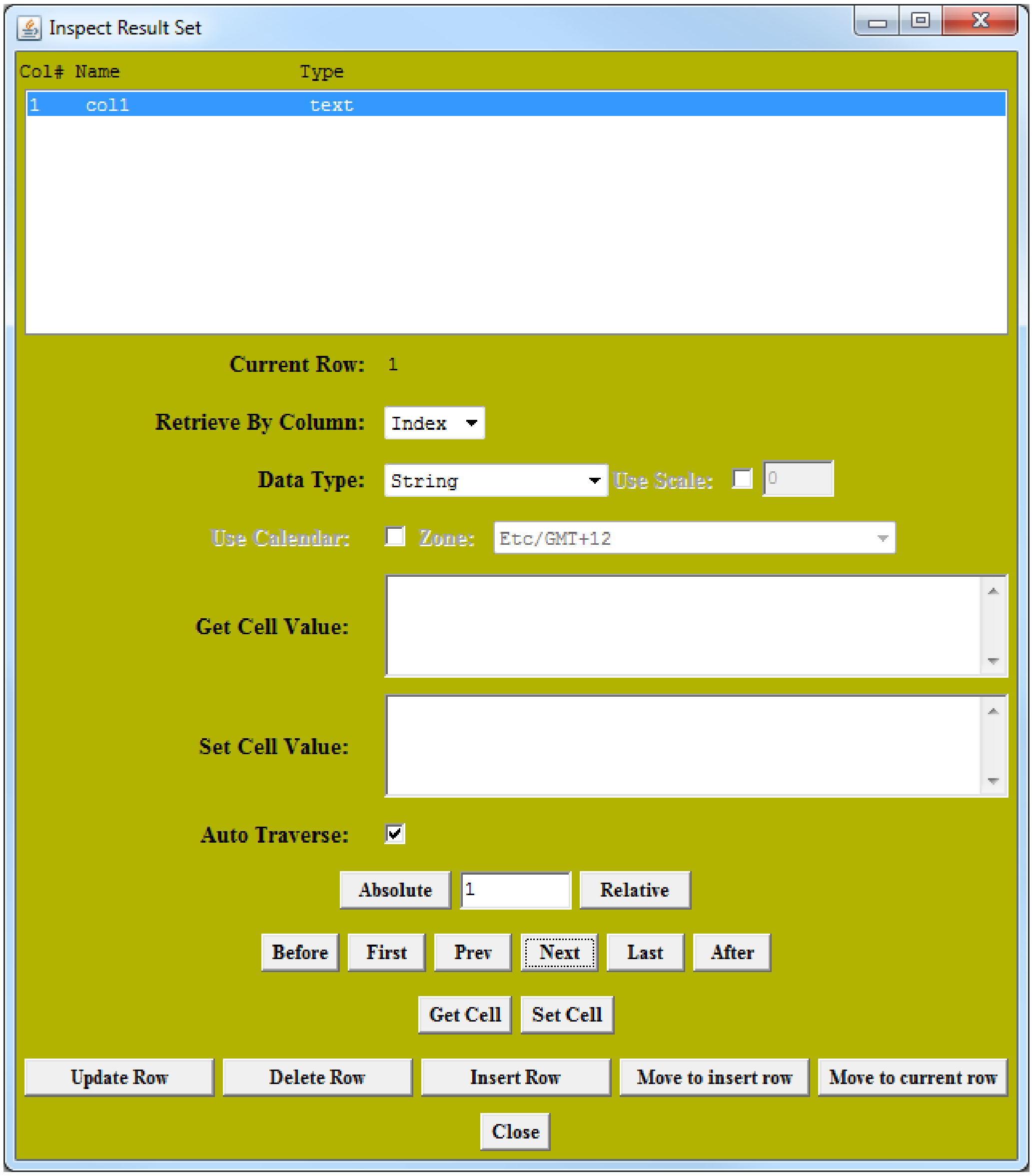


Execute SQL Query

Enter SQL Here:

```
SELECT * FROM clobtest1
```

4. Click **Submit**; then, click **Close**.
5. Select **Results** / **Inspect Results**. The **Inspect Result Set** window appears.
6. Click **Next**. Current Row changes to 1.



Inspect Result Set

Col#	Name	Type
1	col1	text

Current Row: 1

Retrieve By Column: Index

Data Type: String Use Scale: □ 0

Use Calendar: □ Zone: Etc/GMT+12

Get Cell Value:

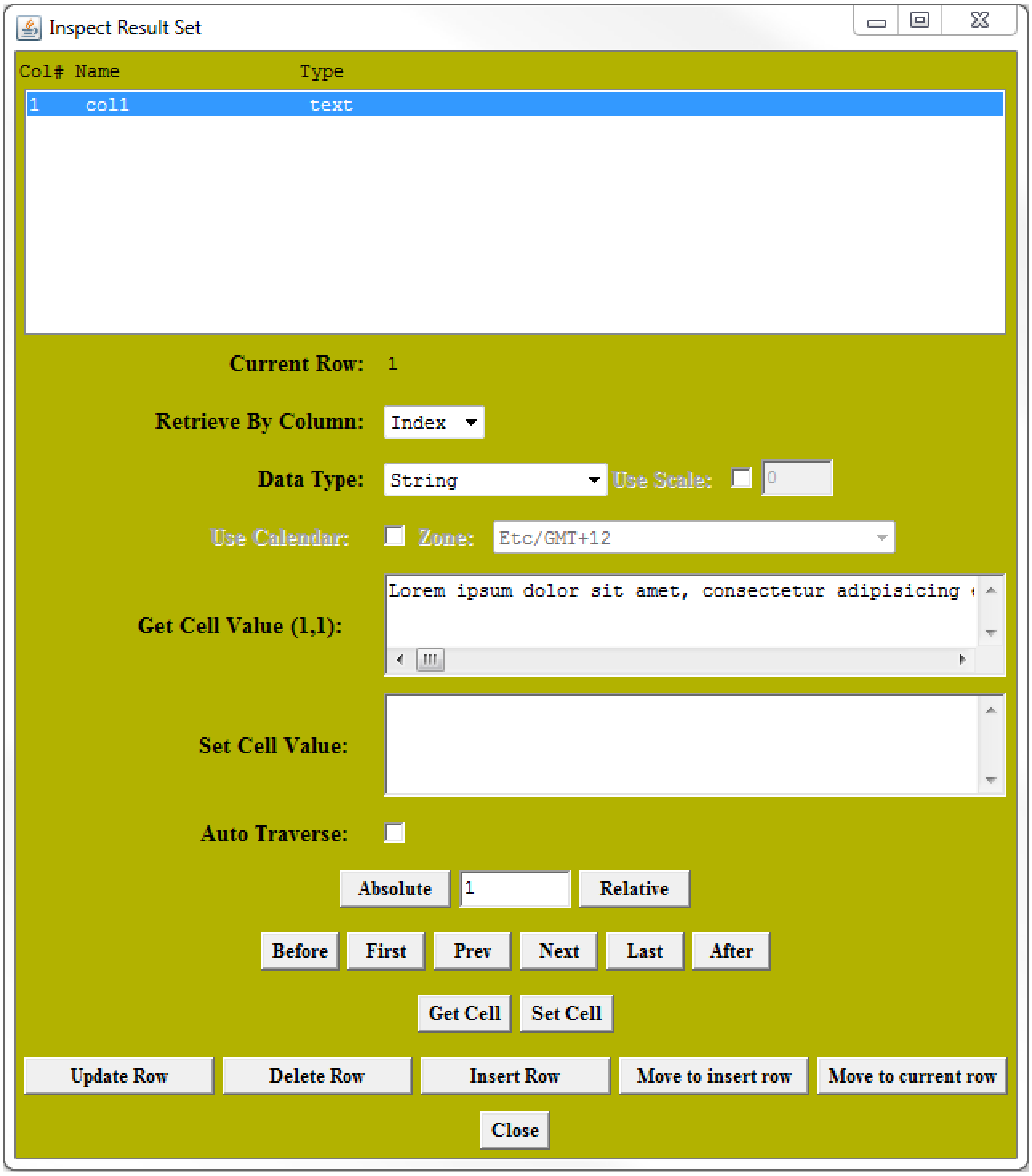
Set Cell Value:

Auto Traverse: ☑

Absolute 1 Relative

7. Deselect **Auto Traverse**. This disables automatic traversal to the next row.

8. Click **Get Cell**. Values are returned in the Get Cell Value field.



Inspect Result Set

```
Col# Name     Type
1    col1     text
```

Current Row: 1

Retrieve By Column: Index

Data Type: String Use Scale: □ 0

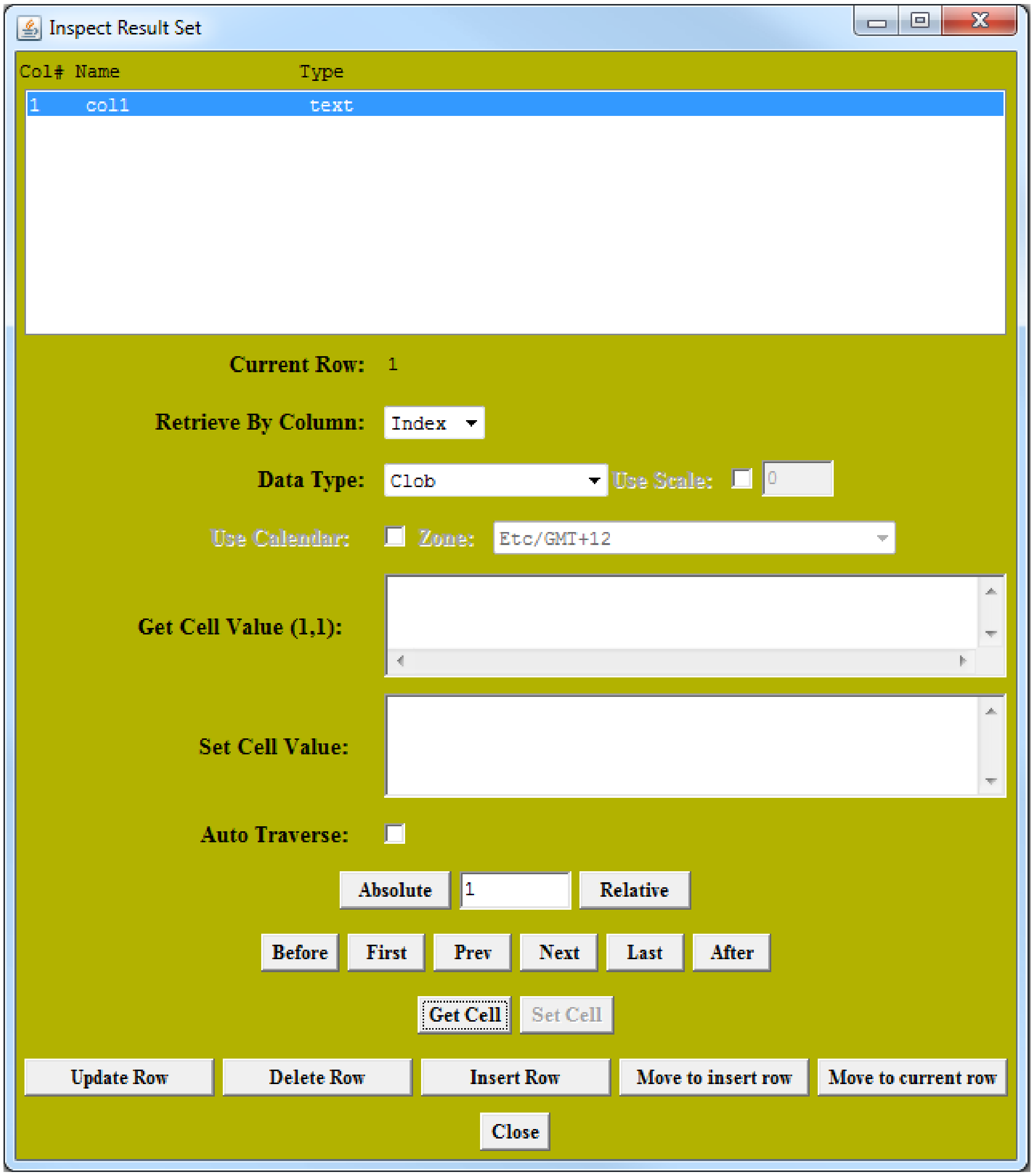
Use Calendar: □ Zone: Etc/GMT+12

Get Cell Value (1,1): Lorem ipsum dolor sit amet, consectetur adipisicing

Set Cell Value:

Auto Traverse: □

9. Change the data type to Clob.



Inspect Result Set

Col#	Name	Type
1	col1	text

Current Row: 1

Retrieve By Column: Index

Data Type: Clob Use Scale: □ 0

Use Calendar: □ Zone: Etc/GMT+12

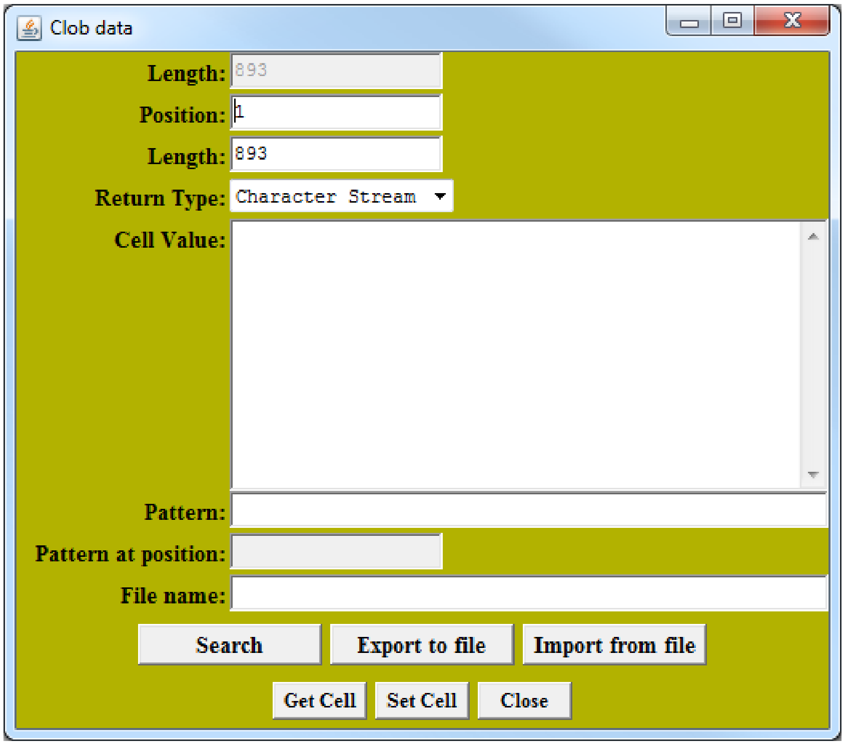
Get Cell Value (1,1):

Set Cell Value:

Auto Traverse: □

Absolute 1 Relative

10. Click **Get Cell**. The **Clob data** window appears.



Clob data

Length: 893

Position: 1

Length: 893

Return Type: Character Stream

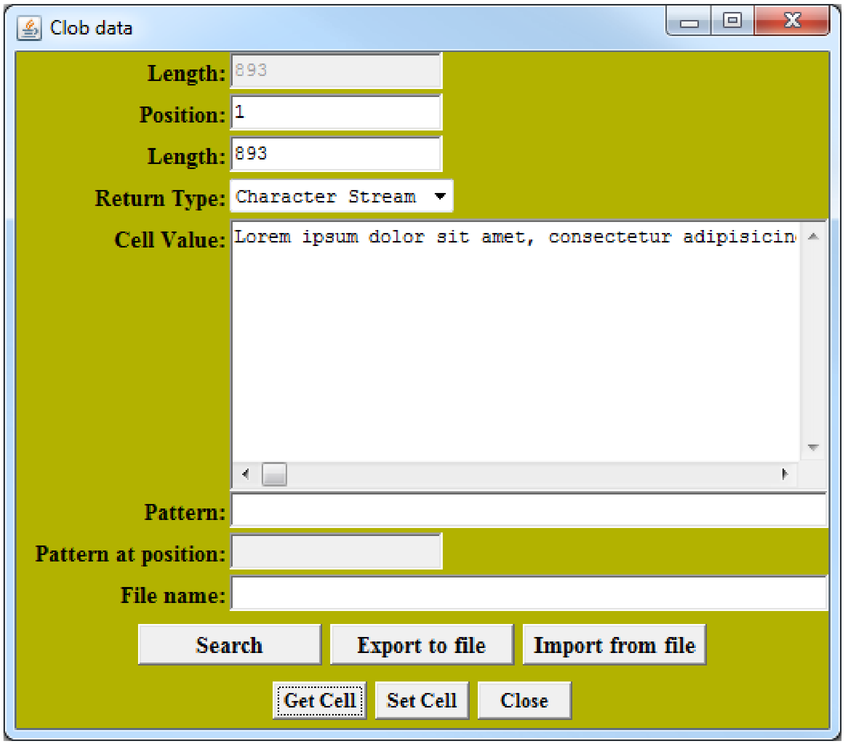
Cell Value:

Pattern:

Pattern at position:

File name:

11. Click **Get Cell**. Values are returned in the Cell Value field.



Clob data

Length: 893

Position: 1

Length: 893

Return Type: Character Stream

Cell Value: Lorem ipsum dolor sit amet, consectetur adipisicin

Pattern:

Pattern at position:

File name:

Tracking JDBC calls with DataDirect Spy

DataDirect Spy is functionality that is built into the drivers. It is used to log detailed information about calls your driver makes and provide information you can use for troubleshooting. DataDirect Spy provides the following advantages:

- Logging is JDBC 4.0-compliant.
- All parameters and function results for JDBC calls can be logged.
- Logging can be enabled without changing the application.

When you enable DataDirect Spy for a connection, you can customize logging by setting one or multiple options for DataDirect Spy. For example, you may want to direct logging to a local file on your machine.

Once logging is enabled for a connection, you can turn it on and off at runtime using the `setEnableLogging` method in the `com.ddtek.jdbc.extensions.ExtLogControl` interface. See "Troubleshooting your application" for information about using a DataDirect Spy log for troubleshooting.

See also

Troubleshooting your application on page 255

Enabling DataDirect Spy

You can enable and customize DataDirect Spy logging in either of the following ways.

- Specifying the SpyAttributes connection property for connections using the JDBC `DriverManager`.
- Specifying DataDirect Spy attributes using a JDBC data source.

Using the JDBC DriverManager

The SpyAttributes connection property allows you to specify a semi-colon separated list of DataDirect Spy attributes. The format for the value of the SpyAttributes property is:

```
(
spy_attribute
[;
spy_attribute
]...)
```

where *`spy_attribute`* is any valid DataDirect Spy attribute. See "DataDirect Spy attributes" for a list of supported attributes.

Windows example

```
Class.forName("com.ddtek.jdbc.sqlserver.SQLServerDriver");
Connection conn = DriverManager.getConnection
   ("jdbc:datadirect:sqlserver://MyServer:1433;DatabaseName=MyDB;
   User=User123;Password=secret;
   SpyAttributes=(log=(filePrefix)C:\\temp\\spy_;linelimit=80;logTName=yes;
   timestamp=yes)");
```

Note: If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(filePrefix)C:\\temp\\spy_`.

Using this example, DataDirect Spy loads the driver and logs all JDBC activity to the `spy_x.log` file located in the `C:\temp` directory (`log=(filePrefix)C:\\temp\\spy_`), where *x* is an integer that increments by 1 for each connection on which the prefix is specified. The `spy_x.log` file logs a maximum of 80 characters on each line (`linelimit=80`) and includes the name of the current thread (`logTName=yes`) and a timestamp on each line in the log (`timestamp=yes`).

UNIX example

```
Class.forName("com.ddtek.jdbc.sqlserver.SQLServerDriver");
Connection conn = DriverManager.getConnection
   ("jdbc:datadirect:sqlserver://MyServer:1433;DatabaseName=MyDB;
   User=User123;Password=secret;
   SpyAttributes=(log=(filePrefix)/tmp/spy_;logTName=yes;timestamp=yes)");
```

Using this example, DataDirect Spy loads the driver and logs all JDBC activity to the `spy_x.log` file located in the `/tmp directory` (`log=(filePrefix)/tmp/spy_`), where *x* is an integer that increments by 1 for each connection on which the prefix is specified. The `spy_x.log` file includes the name of the current thread (`logTName=yes`) and a timestamp on each line in the log (`timestamp=yes`).

See also

DataDirect Spy attributes on page 182

Using JDBC data sources

You can use DataDirect Spy to track JDBC calls made by a running application with either of these features:

- JNDI for Naming Databases
- Connection Pooling

The `com.ddtek.jdbcx.SQLServer.SQLServerDataSource` class supports setting a semi-colon separated list of DataDirect Spy attributes.

Windows example

```
SQLServerDataSource sds = new SQLServerDataSource();
sds.setDescription("My SQLServer Datasource");
sds.setServerName("MyServer");
sds.setPortNumber(1433);
sds.setUser("User123");
sds.setPassword("secret");
sds.setDatabaseName("MyDB");
sds.setSpyAttributes("log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes");
Connection conn=sds.getConnection;
...
```

Note: If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes`.

DataDirect Spy loads the driver and logs all JDBC activity to the `spy.log` file located in the `C:\temp` directory (`log=(file)C:\\temp\\spy.log`). In addition to regular JDBC activity, the `spy.log` file also logs activity on `InputStream` and `Reader` objects (`logIS=yes`). It also includes the name of the current thread (`logTName=yes`).

UNIX example

```
SQLServerDataSource mds = new SQLServerDataSource();
mds.setDescription("My SQLServer Datasource");
mds.setServerName("MyServer");
mds.setPortNumber(1433);
mds.setUser("User123");
mds.setPassword("secret");
mds.setDatabaseName("MyDB");
mds.setSpyAttributes("log=(file)/tmp/spy.log;logIS=yes;logTName=yes");
Connection conn=mds.getConnection;
...
```

DataDirect Spy loads the driver and logs all JDBC activity to the `spy.log` file located in the `/tmp` directory (`log=(file)/tmp/spy.log`). In addition to regular JDBC activity, the `spy.log` file also logs activity on `InputStream` and `Reader` objects (`logIS=yes`). It also includes the name of the current thread (`logTName=yes`).

See also

DataDirect Spy attributes on page 182

DataDirect Spy attributes

DataDirect Spy supports the attributes described in the following table.

Table 17: DataDirect Spy Attributes

Attribute	Description
`linelimit=`*`numberofchars`*	Sets the maximum number of characters that DataDirect Spy logs on a single line. The default is `0` (no maximum limit).
`load=`*`classname`*	Loads the driver specified by *`classname`*.
`log=(file)`*`filename`*	Directs logging to the file specified by *`filename`*. For Windows, if coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(file)C:\\temp\\spy.log;logIS=yes;logTName=yes`.

Attribute	Description

`log=(filePrefix)`*`file_prefix`*	Directs logging to a file prefixed by *`file_prefix`*. The log file is named *`file_prefixX`*`.log`
	where:
	X
	is an integer that increments by 1 for each connection on which the prefix is specified.
	For example, if the attribute `log=(filePrefix) C:\\temp\\spy_` is specified on multiple connections, the following logs are created:
	`C:\temp\spy_1.log`
	`C:\temp\spy_2.log`
	`C:\temp\spy_3.log`
	`...`
	If coding a path to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(filePrefix)C:\\temp\\spy_;logIS=yes;logTName=yes`.

`log=System.out`	Directs logging to the Java output standard, `System.out`.
`logIS={yes | no | nosingleread}`	Specifies whether DataDirect Spy logs activity on `InputStream` and `Reader` objects. When `logIS=nosingleread`, logging on `InputStream` and `Reader` objects is active; however, logging of the single-byte read `InputStream.read` or single-character `Reader.read` is suppressed to prevent generating large log files that contain single-byte or single character read messages. The default is no.
`logLobs={yes | no}`	Specifies whether DataDirect Spy logs activity on BLOB and CLOB objects.
`logTName={yes | no}`	Specifies whether DataDirect Spy logs the name of the current thread. The default is no.
`timestamp={yes | no}`	Specifies whether a timestamp is included on each line of the DataDirect Spy log. The default is no.

4

Connection property descriptions

You can use connection properties to customize the driver for your environment. This section lists the connection properties supported by the driver and describes each property. You can use these connection properties with either the JDBC `DriverManager` or a JDBC data source. For a `DriverManager` connection, a property is expressed as a key value pair and takes the form *`property`*`=`*`value`*. For a data source connection, a property is expressed as a JDBC method and takes the form `set`*`property`*`(`*`value`*`)`. Connection property names are case-insensitive. For example, `Password` is the same as `password`.

Note: In a JDBC `DataSource`, string values must be enclosed in double quotation marks, for example, `setUser("User123")`.

Note: The data type listed for each connection property is the Java data type used for the property value in a JDBC data source.

The following table provides a summary of the connection properties supported by the driver, their corresponding data source methods, and their default values.

Table 18: Connection properties

Property	Data source method	Default
AccountingInfo on page 192	`setAccountingInfo`	Empty string
AEKeyCacheTTL on page 193	`setAEKeyCacheTTL`	`7200`
AEKeystoreClientSecret on page 194	`setAEKeystoreClientSecret`	None

Property	Data source method	Default
AEKeystoreLocation on page 195	`setAEKeystoreLocation`	None
AEKeystorePrincipalId on page 196	`setAEKeystorePrincipalId`	None
AEKeystoreSecret on page 197	`setAEKeystoreSecret`	Empty string
AlternateServers on page 198	`setAlternateServers`	None
AlwaysReportTriggerResults on page 199	`setAlwaysReportTriggerResults`	`false`
ApplicationIntent on page 199	`setApplicationIntent`	`ReadWrite`
ApplicationName on page 200	`setApplicationName`	`Empty string`
AuthenticationMethod on page 201	`setAuthenticationMethod`	`auto`
BulkLoadBatchSize on page 202	`setBulkLoadBatchSize`	`1000` (rows)
BulkLoadOptions on page 203	`setBulkLoadOptions`	2
CatalogOptions on page 204	`setCatalogOptions`	`0`
ClientHostName on page 205	`setClientHostName`	Empty string
ClientUser on page 205	`setClientUser`	Empty string
ColumnEncryption on page 207	`setColumnEncryption`	`Disabled`
CodePageOverride on page 206	`setCodePageOverride`	None
ConnectionRetryCount on page 208	`setConnectionRetryCount`	5
ConnectionRetryDelay on page 209	`setConnectionRetryDelay`	1 (second)
ConvertNull on page 210	`setConvertNull`	1 (data type check is performed if column value is null)
CryptoProtocolVersion on page 210	`setCryptoProtocolVersion`	None

Property	Data source method	Default
DatabaseName on page 211	setDatabaseName	None
DateTimeInputParameterType on page 212	setDateTimeInputParameterType	auto
DateTimeOutputParameterType on page 213	setDateTimeOutputParameterType	auto
DescribeInputParameters on page 214	setDescribeInputParameters	noDescribe
DescribeOutputParameters on page 215	setDescribeOutputParameters	noDescribe
Domain on page 216	setDomain	None
EnableBulkLoad on page 216	setEnableBulkLoad	false
EnableCancelTimeout on page 217	setEnableCancelTimeout	false
EncryptionMethod on page 218	setEncryptionMethod	noEncryption
FailoverGranularity on page 219	setFailoverGranularity	nonAtomic
FailoverMode on page 220	setFailoverMode	connect
FailoverPreconnect on page 221	setFailoverPreconnect	false
FetchTSWTZAsTimestamp on page 221	setFetchTSWTZAsTimestamp	false
FetchTWFSasTime on page 222	setFetchTWFSasTime	false
GSSCredential on page 223	setGSSCredential	Null
HostNameInCertificate on page 224	setHostNameInCertificate	Empty string
ImportStatementPool on page 225	setImportStatementPool	Empty string

Property	Data source method	Default
InitializationString on page 225	setInitializationString	None
InsensitiveResultSetBufferSize on page 226	setInsensitiveResultSetBufferSize	2048 (KB)
JavaDoubleToString on page 227	setJavaDoubleToString	false
JDBCBehavior on page 228	setJDBCBehavior	1
LoadBalancing on page 228	setLoadBalancing	false
LoginConfigName on page 229	setLoginConfigName	JDBC_DRIVER_01
LoginTimeout on page 230	setLoginTimeout	0 (no timeout)
LongDataCacheSize on page 231	setLongDataCacheSize	2048 (KB)
MaxPooledStatements on page 232	setMaxPooledStatements	0 (driver's internal prepared statement pooling is not enabled)
MultiSubnetFailover on page 233	setMultiSubnetFailover	false
NetAddress on page 234	setNetAddress	000000000000
PacketSize on page 234	setPackeSize	-1 (maximum packet size accepted by the database server)
Password on page 236	setPassword	None
PortNumber on page 236	setPortNumber	1433
ProgramID on page 237	setProgramID	Empty string
QueryTimeout on page 238	setQueryTimeout	0 (query does not time out)
RegisterStatementPoolMonitorMBean on page 238	setRegisterStatementPoolMonitorMBean	false

Property	Data source method	Default
ResultSetMetaDataOptions on page 239	`setResultSetMetaDataOptions`	`0` (no additional processing to determine the correct table name for each column in the result set)
SelectMethod on page 240	`setSelectMethod`	`direct`
ServerName on page 241	`setServerName`	None
ServicePrincipalName on page 242	`setServicePrincipalName`	Driver builds value based on environment
SnapshotSerializable on page 243	`setSnapshotSerializable`	`false`
SpyAttributes on page 244	`setSpyAttributes`	None
StringInputParameterType on page 245	`setStringInputParameter`	`nvarchar`
StringOutputParameterType on page 246	`setStringOutputParameter`	`nvarchar`
SuppressConnectionWarnings on page 246	`setSuppressConnectionWarnings`	`false`
TransactionMode on page 247	`setTransactionMode`	`implicit`
TruncateFractionalSeconds on page 248	`setTruncateFractionalSeconds`	`true`
TrustStore on page 248	`setTrustStore`	None
TrustStorePassword on page 249	`setTrustStorePassword`	None
User on page 250	`setUser`	None
UseServerSideUpdatableCursors on page 251	`setUseServerSideUpdatableCursors`	`false`
ValidateServerCertificate on page 251	`setValidateServerCertificate`	`true`

Property	Data source method	Default
XATransactionGroup on page 252	`setXATransactionGroup`	None
XMLDescribeType on page 253	`setXMLDescribeType`	None

For details, see the following topics:

- AccountingInfo
- AEKeyCacheTTL
- AEKeystoreClientSecret
- AEKeystoreLocation
- AEKeystorePrincipalId
- AEKeystoreSecret
- AlternateServers
- AlwaysReportTriggerResults
- ApplicationIntent
- ApplicationName
- AuthenticationMethod
- BulkLoadBatchSize
- BulkLoadOptions
- CatalogOptions
- ClientHostName
- ClientUser
- CodePageOverride
- ColumnEncryption
- ConnectionRetryCount
- ConnectionRetryDelay
- ConvertNull
- CryptoProtocolVersion
- DatabaseName
- DateTimeInputParameterType
- DateTimeOutputParameterType
- DescribeInputParameters

- DescribeOutputParameters
- Domain
- EnableBulkLoad
- EnableCancelTimeout
- EncryptionMethod
- FailoverGranularity
- FailoverMode
- FailoverPreconnect
- FetchTSWTZAsTimestamp
- FetchTWFSasTime
- GSSCredential
- HostNameInCertificate
- ImportStatementPool
- InitializationString
- InsensitiveResultSetBufferSize
- JavaDoubleToString
- JDBCBehavior
- LoadBalancing
- LoginConfigName
- LoginTimeout
- LongDataCacheSize
- MaxPooledStatements
- MultiSubnetFailover
- NetAddress
- PacketSize
- Password
- PortNumber
- ProgramID
- QueryTimeout
- RegisterStatementPoolMonitorMBean
- ResultSetMetaDataOptions
- SelectMethod
- ServerName

- ServicePrincipalName
- SnapshotSerializable
- SpyAttributes
- StringInputParameterType
- StringOutputParameterType
- SuppressConnectionWarnings
- TransactionMode
- TruncateFractionalSeconds
- TrustStore
- TrustStorePassword
- User
- UseServerSideUpdatableCursors
- ValidateServerCertificate
- XATransactionGroup
- XMLDescribeType

AccountingInfo

Purpose

Defines accounting information. This value is stored locally and is used for database administration/monitoring purposes.

Valid values

`string`

where:

`string`

is the accounting information.

Data source method

`setAccountingInfo`

Default

Empty string

Data type

String

See also

- Using client information on page 104
- Client information properties on page 68

AEKeyCacheTTL

Purpose

Specifies the length of time, in seconds, column encryption keys live in the cache before the driver deletes them. This option is used when Always Encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`).

Valid Values

`0` | *x*

where:

x

is the number of seconds the driver stores a column encryption key in the cache.

Behavior

If set to `-1`, the driver caches column encryption keys for the life of the connection. The keys are deleted when the connection is closed or added to the connection pool.

If set to `0`, the driver does not cache column encryption keys.

If set to *x*, the driver caches column encryption keys for the specified number of seconds before deleting them. The timer starts for a key when it is first accessed and added to the cache. The timer does not reset if you access it after it has been added to the cache. The keys are deleted when the timer expires, or the connection is closed or added to the connection pool.

Notes

- Column encryption keys do not persist beyond the life of the connection. When a connection is closed, the driver purges the cache, leaving no column encryption key data in memory.
- Caching column encryption keys can provide performance gains by reducing the overhead associated with fetching and decrypting the keys for the same data multiple times during a connection. Specifying larger values for this property increases the length of time that a column encryption key persists in the cache; therefore, improving performance in some scenarios. Note that column encryption keys are designed to be deleted from the cache as a security measure and should not be configured to live for long periods of time.

Data source method

`setAEKeyCacheTTL`

Default

`7200`

Data Type

Long

See Also

- ColumnEncryption on page 207
- Always Encrypted on page 88
- Always Encrypted properties on page 69
- Performance considerations on page 75

AEKeystoreClientSecret

Purpose

Specifies the Client Secret used to authenticate against the Azure Key Vault. This property is used only when Always Encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`) and Azure Key Vault is the keystore provider. The Azure Key Vault stores the column master key used for Always Encrypted functionality. To access the column master key from the Azure Key Vault, the Client Secret and principal ID must be provided.

Valid Values

`client_secret`

where:

`client_secret`

is the Client Secret used to authenticate against the Azure Key Vault.

Notes

- To specify the principal ID, use the AEKeystorePrincipalId connection property.

Data source method

`setAEKeystoreClientSecret`

Default

None

Data Type

String

See Also

- ColumnEncryption on page 207
- AEKeystorePrincipalId on page 196
- Always Encrypted on page 88
- Always Encrypted properties on page 69

AEKeystoreLocation

Purpose

Specifies the absolute path to the Java KeyStore file. This property is used only when Always Encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`) and the Java KeyStore is the keystore provider. The Java KeyStore file contains the column master key used for Always Encrypted functionality.

Valid Values

`java_keystore_path`

where:

`java_keystore_path`

is the absolute path to the Java KeyStore file.

Notes

- To specify the password for the Java KeyStore file, use the AEKeystoreSecret connection property.
- This property is required when encryption keys are stored in a Java KeyStore and Always Encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`).
- The value for this property is ignored when Always Encrypted is disabled (`ColumnEncryption=Disabled`).
- The driver supports the JKS and PKCS12 file formats.

Data source method

`setAEKeystoreLocation`

Default

None

Data Type

String

See Also

- ColumnEncryption on page 207
- AEKeystoreSecret on page 197
- Always Encrypted on page 88
- Always Encrypted properties on page 69

AEKeystorePrincipalId

Purpose

Specifies the principal ID used to authenticate against the Azure Key Vault. This property is used only when Always Encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`) and Azure Key Vault is the keystore provider. The Azure Key Vault stores the column master key used for Always Encrypted functionality. To access the column master key from the Azure Key Vault, the principal ID and Client Secret must be provided.

Valid Values

`principal_id`

where:

`principal_id`

is the Application ID created during Azure App Registration and used to authenticate against the Azure Key Vault.

Notes

- To specify the Client Secret, use the AEKeystoreClientSecret connection property.
- The driver currently supports only Azure App Registration as the principal ID.
- This property is used only when the Azure Key Vault is specified as the keystore provider by column metadata or in statement parameters.

Data source method

`setAEKeystorePrincipalId`

Default

None

Data Type

String

See Also

- ColumnEncryption on page 207
- AEKeystoreClientSecret on page 194
- Always Encrypted on page 88
- Always Encrypted properties on page 69

AEKeystoreSecret

Purpose

Specifies the password used to access the Java KeyStore file. This property is used only when Always Encrypted is enabled (`ColumnEncryption=Enabled` | `ResultsetOnly`) and the Java KeyStore is the keystore provider. The Java KeyStore contains the column master key used for Always Encrypted functionality.

Valid Values

`keystore_password`

where:

`keystore_password`

is the password used to access the key(s) in the Java keystore.

Notes

- This property is used to access the key(s) in the Java KeyStore file specified by the AEKeystoreLocation connection property.
- If no value is specified for this property, an empty string is passed to the Java KeyStore file.
- The value for this property is ignored when Always Encrypted is disabled (`ColumnEncryption=Disabled`).

Data source method

`setAEKeystoreSecret`

Default

Empty string

Data Type

String

See Also

- ColumnEncryption on page 207
- AEKeystoreLocation on page 195
- Always Encrypted on page 88
- Always Encrypted properties on page 69

AlternateServers

Purpose

A list of alternate database servers that is used to failover new or lost connections, depending on the failover method selected. See FailoverMode on page 220 for information about choosing a failover method.

Valid values

```
(servername1[:port1][;property=value[;...]][,servername2[:port2]
[;property=value[;...]]]...)
```

Behavior

The server name (*servername1*, *servername2*, and so on) is required for each alternate server entry. Port number (*port1*, *port2*, and so on) and connection properties (*property*=*value*) are optional for each alternate server entry. If the port is unspecified, the port number of the primary server is used. If a port number for the primary server is unspecified, a default port number of 1433 is used.

The driver allows only one optional connection property, DatabaseName.

Notes

If using failover with Microsoft Cluster Server (MSCS), which determines the alternate server for failover instead of the driver, any alternate server specified must be the same as the primary server. For example:

```
jdbc:datadirect:sqlserver://server1:1433;DatabaseName=TEST;User=test;
Password=secret;AlternateServers=(server1:1433;DatabaseName=TEST)
```

Example

The following URL contains alternate server entries for server2 and server3. The alternate server entries contain the optional DatabaseName property.

```
jdbc:datadirect:sqlserver://server1:1433;DatabaseName=TEST;User=test;
Password=secret;AlternateServers=(server2:1433;DatabaseName=TEST2,
server2:1433;DatabaseName=TEST3)
```

Data source method

`setAlternateServers`

Default

None

Data type

String

See also

- FailoverMode on page 220
- Using failover on page 92

AlwaysReportTriggerResults

Purpose

Determines how the driver reports results that are generated by database triggers (procedures that are stored in the database and executed, or fired, when a table is modified).

Valid values

`true` | `false`

Behavior

If set to `true`, the driver returns all results, including results that are generated by triggers. Multiple trigger results are returned sequentially. Use the Statement.getMoreResults() method to return individual trigger results. Warnings and errors are reported in the results as they are encountered.

If set to `false`, the driver does not report trigger results if the statement is a single Insert, Update, Delete, Create, Alter, Drop, Grant, Revoke, or Deny statement. The only result that is returned is the update count that is generated by the statement that was executed (if errors do not occur). Although trigger results are ignored, any errors that are generated by the trigger are reported. Any warnings that are generated by the trigger are enqueued. If errors are reported, the update count is not reported.

Data source method

`setAlwaysReportTriggerResults`

Default

`false`

Data type

Boolean

ApplicationIntent

Purpose

Specifies whether the driver connects to read-write databases or requests read-only routing to connect to read-only database replicas. Read-only routing only applies to connections in Microsoft SQL Server 2012 where AlwaysOn Availability Groups have been deployed.

Valid values

`ReadWrite` | `ReadOnly`

Behavior

If set to `ReadWrite`, the driver connects to a read-write node in the AlwaysOn environment.

If set to `ReadOnly`, the driver requests read-only routing and connects to the read-only database replicas specified by the server.

Notes

By setting applicationIntent to `ReadOnly` and querying read-only database replicas when possible, you shift load away from the read-write nodes of your database cluster to read-only nodes.

Data source method

`setApplicationIntent`

Default

`ReadWrite`

Data type

String

See also

Always On Availability Groups on page 100

ApplicationName

Purpose

The name of the application to be stored in the database. This property sets the program_name column in the sysprocesses table in the database.

Valid values

`string`

where:

`string`

is the name of the application.

Notes

- ProgramName can be used as an alias for ApplicationName.
- Your database may impose character length restrictions on the value. If the value exceeds a restriction, the driver truncates it.

Data source method

`setApplicationName`

Default

Empty string

Data type

String

See also

- Using client information on page 104
- Client information properties on page 68

AuthenticationMethod

Purpose

Determines which authentication method the driver uses when establishing a connection.

Valid values

`ActiveDirectoryPassword` | `auto` | `kerberos` | `ntlmjava` | `ntlm2java` | `userIdPassword`

Behavior

If set to `ActiveDirectoryPassword`, the driver uses Azure Active Directory (Azure AD) authentication when establishing a connection to Azure. In addition to specifying a user ID and password, a value must be specified for the HostNameInCertificate property. All communications to the service are encrypted using SSL.

If set to `auto`, the driver uses SQL Server authentication or Kerberos authentication based on the following criteria.

- If a user ID and password is specified, the driver uses SQL Server authentication when establishing a connection. The User property provides the user ID. The Password property provides the password.
- If a user ID and password is not specified, the driver uses Kerberos authentication when establishing a connection.

If set to `kerberos`, the driver uses Kerberos authentication when establishing a connection. The driver ignores any values specified by the User and Password properties. The driver uses the authentication technology based on the value specified for the LoginConfigName property to establish a Kerberos connection.

If set to `ntlmjava`, the driver uses NTLMv1 or NTLMv2 depending on the size of the NTLM password. NTLMv1 is used if the password is 14 bytes or less; NTLMv2 is used if the password is more than 14 bytes. A user ID and password must also be specified. If the user ID and password are unspecified, the driver throws an exception. In addition, the driver requires the name of the domain server that administers the database server. You can specify it using the Domain property. If the Domain property is unspecified, the driver attempts to determine the domain server name from the User property. If no domain is specified, the driver throws an exception.

If set to `ntlm2java`, the driver uses NTLMv2 authentication. A user ID and password must also be specified. If the user ID and password are unspecified, the driver throws an exception. In addition, the driver requires the name of the domain server that administers the database server. You can specify it using the Domain property. If the Domain property is unspecified, the driver attempts to determine the domain server name from the User property. If no domain is specified, the driver throws an exception.

If set to `userIdPassword`, the driver uses SQL Server authentication when establishing a connection. The User property provides the user ID. The Password property provides the password. If a user ID is not specified, the driver throws an exception.

Notes

- If your are configuring your environment for Kerberos constrained delegation, AuthenticationMethod must be set to `kerberos`.

- The User property provides the user ID. The Password property provides the password.
- Azure AD authentication (`AuthenticationMethod=ActiveDirectoryPassword`) requires Java SE 7 or higher.
- When using Azure AD authentication (`AuthenticationMethod=ActiveDirectoryPassword`), the driver requires root CA certificates to establish an SSL connection to a database. The driver determines the location of the truststore containing the required certificates by using the default JRE `cacerts` file, unless a different file has been specified by the `javax.net.ssl.trustStore` Java system property. The truststore location cannot be specified using the driver's Truststore property.
- If you specify `AuthenticationMethod=ntlmjava` when the LMCompatabilityLevel has been restricted to NTLMv2, an error will be returned. When the LMCompatabilityLevel has been restricted to NTLMv2, AuthenticationMethod must be set to `ntlm2java`.

Data source method

`setAuthenticationMethod`

Default

`auto`

Data type

String

See also

- Authentication on page 79
- Authentication properties on page 59

BulkLoadBatchSize

Purpose

Provides a suggestion to the driver for the number of rows to load to the database at a time when bulk loading data. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Be aware that increasing the number of rows that are loaded also causes the driver to consume more memory on the client.

Valid values

x

where:

x

is a positive integer that represents a number of rows.

Notes

- This property suggests the number of rows regardless whether using a DDBulkLoad object or using native bulk protocols in the database for batch inserts.

- The DDBulkObject.setBatchSize() method overrides the value set by this property. See DataDirect Bulk Load on page 139 and JDBC extensions on page 329 for details.

Data source method

`setBulkLoadBatchSize`

Default

`1000` (rows)

Data type

Long

See also

- DataDirect Bulk Load on page 139
- Bulk load properties on page 63
- JDBC extensions on page 329

BulkLoadOptions

Purpose

Enables bulk load protocol options for batch inserts that the driver can take advantage of when EnableBulkLoad is set to a value of `true`.

Valid values

This value is the cumulative value of all enabled options. The following list describes the value and the corresponding option that is enabled:

Value	Option Enabled
`1`	The KeepIdentity option preserves identity values. If unspecified, identity values are ignored in the source and are assigned by the destination.
`2`	The TableLock option assigns a table lock for the duration of the bulk copy operation. Other applications cannot update the table until the operation completes. If unspecified, the default bulk locking mechanism specified by the database server is used.
`16`	The CheckConstraints option checks integrity constraints while data is being copied. If unspecified, constraints are not checked.
`32`	The FireTriggers option causes the database server to fire insert triggers for the rows being inserted into the database. If unspecified, triggers are not fired.
`64`	The KeepNulls option preserves null values in the destination table regardless of the settings for default values. If unspecified, null values are replaced by column default values where applicable.

Behavior

If set to `0`, all the options are disabled.

Example

A value of `67` means the KeepIdentity, TableLock, and KeepNulls options are enabled (1 + 2 + 64).

Data source method

`setBulkLoadOptions`

Default

`2`

Data type

Long

See also

- DataDirect Bulk Load on page 139
- Bulk load properties on page 63

CatalogOptions

Purpose

Determines which type of metadata information is included in result sets when an application calls DatabaseMetaData methods.

Valid values

`0` | `2`

Behavior

If set to `0`, result sets do not contain synonyms.

If set to `2`, result sets contain synonyms that are returned from the following DatabaseMetaData methods: getFunctions(), getTables(), getColumns(), getProcedures(), getProcedureColumns(), and getFunctionColumns()

Data source method

`setCatalogOptions`

Default

`0`

Data type

Int

ClientHostName

Purpose

The host name, or workstation ID, of the client machine to be stored in the database. This property sets the hostname column of the sysprocesses table in the database.

Valid values

`string`

where:

`string`

is the host name of the client machine.

Notes

- WSID can be used as an alias for ClientHostName.
- Your database may impose character length restrictions on the value. If the value exceeds a restriction, the driver truncates it.

Data source method

`setClientHostName`

Default

Empty string

Data type

String

See also

- Using client information on page 104
- Client information properties on page 68

ClientUser

Purpose

Specifies the user ID. This value is stored locally and is used for database administration/monitoring purposes.

Valid values

`string`

where:

`string`

is a valid user ID.

Data source method

`setClientUser`

Default

Empty string

Data type

String

See also

- Using client information on page 104
- Client information properties on page 68

CodePageOverride

Purpose

The code page to be used by the driver to convert Character data. The specified code page overrides the default database code page or column collation. All Character data returned from or written to the database is converted using the specified code page.

Valid values

`string`

where:

`string`

is the name of a valid code page that is supported by your JVM.

Behavior

By default, the driver automatically determines which code page to use to convert Character data. Use this property only if you need to change the driver's default behavior.

If a value is set for this property and the StringInputParameterType property, the driver ignores the StringInputParameterType property and generates a warning. The driver always sends parameters using the code page that is specified by CodePageOverride, if specified.

Example

`CP950`

Data source method

`setCodePageOverride`

Default

None

Data type

String

ColumnEncryption

Purpose

Specifies whether the driver is enabled for Always Encrypted functionality when accessing data from encrypted columns.

Valid Values

`Disabled` | `ResultsetOnly` | `Enabled`

Behavior

If set to `Disabled`, the driver does not use Always Encrypted functionality. The driver does not attempt to decrypt data from encrypted columns, but will return data as binary formatted cipher text. However, statements containing parameters that reference encrypted columns are not supported and will return an error.

If set to `ResultsetOnly`, the driver transparently decrypts result sets and returns them to the application. Queries containing parameters that affect encrypted columns will return an error.

If set to `Enabled`, the driver fully supports Always Encrypted functionality. The driver transparently decrypts result sets and returns them to the application. In addition, the driver transparently encrypts parameter values that are associated with encrypted columns.

Notes

- When Always Encrypted functionality is enabled, values must be provided for the following properties according to your keystore provider:
  - For Azure Key Vault, you must specify values for the AEKeystorePrincipalId and AEKeystoreClientSecret properties.
  - For Java KeyStore, you must specify values for the AEKeystoreLocation and AEKeystoreSecret properties.
- When Always Encrypted functionality is enabled, the driver transparently supports both randomized encryption and deterministic encryption.
- Parameter markers must be used when specifying values that are associated with encrypted columns. If literal values are specified in a statement targeting encrypted columns, the driver will return an error.

Data source method

`setColumnEncryption`

Default

`Disabled`

Data Type

String

See Also

- AEKeystorePrincipalId on page 196
- AEKeystoreClientSecret on page 194
- AEKeystoreLocation on page 195
- AEKeystoreSecret on page 197
- Always Encrypted on page 88
- Always Encrypted properties on page 69
- Performance considerations on page 75

ConnectionRetryCount

Purpose

The number of times the driver retries connection attempts to the primary database server, and if specified, alternate servers until a successful connection is established.

Valid values

`0` | *x*

where:

x

is a positive integer that represents the number of retries.

Behavior

If set to `0`, the driver does not try to reconnect after the initial unsuccessful attempt.

If set to *x*, the driver retries connection attempts the specified number of times. If a connection is not established during the retry attempts, the driver returns an exception that is generated by the last database server to which it tried to connect.

Notes

- If an application sets a login timeout value (for example, using DataSource.loginTimeout or DriverManager.loginTimeout), and the login timeout expires, the driver ceases connection attempts.
- The ConnectionRetryDelay property specifies the wait interval, in seconds, to occur between retry attempts.
- If MultiSubnetFailover is enabled and the connection attempt fails, the driver will attempt to connect two more times, regardless of the ConnectionRetryCount setting.

Example

If this property is set to 2 and alternate servers are specified using the AlternateServers property, the driver retries the list of servers (primary and alternate) twice after the initial retry attempt.

Data source method

`setConnectionRetryCount`

Default

`5`

Data type

Int

See also

Using failover on page 92

ConnectionRetryDelay

Purpose

The number of seconds the driver waits between connection retry attempts when ConnectionRetryCount is set to a positive integer.

Valid values

`0` | *x*

where:

x

is a number of seconds.

Behavior

If set to `0`, the driver does not delay between retries.

If set to *x*, the driver waits between connection retry attempts the specified number of seconds.

Example

If ConnectionRetryCount is set to 2, this property is set to 3, and alternate servers are specified using the AlternateServers property, the driver retries the list of servers (primary and alternate) twice after the initial retry attempt. The driver waits 3 seconds between retry attempts.

Notes

When MultiSubnetFailover is enabled, the ConnectionRetryDelay connection property is ignored.

Data source method

`setConnectionRetryDelay`

Default

`1` (second)

Data type

Int

See also

Using failover on page 92

ConvertNull

Purpose

Controls how data conversions are handled for null values.

Valid values

`0 | 1`

Behavior

If set to `0`, the driver does not perform the data type check if the value of the column is null. This allows null values to be returned even though a conversion between the requested type and the column type is undefined.

If set to `1`, the driver checks the data type this is requested against the data type of the table column that stores the data. If a conversion between the requested type and column type is not defined, the driver generates an "unsupported data conversion" exception regardless of the data type of the column value.

Data source method

`setConvertNull`

Default

`1`

Data type

Int

See also

Data type handling properties on page 64

CryptoProtocolVersion

Purpose

Specifies a cryptographic protocol or comma-separated list of cryptographic protocols that can be used when SSL is enabled (`EncryptionMethod=SSL`).

Valid Values

cryptographic_protocol [[, *cryptographic_protocol*]...]

where:

cryptographic_protocol

is one of the following cryptographic protocols:

TLSv1.2 | TLSv1.1 | TLSv1 | SSLv3 | SSLv2

Caution: To avoid vulnerabilities associated with SSLv3 and SSLv2, good security practices recommend using TLSv1 or higher.

Example

If your server supports TLSv1.1 and TLSv1.2, you can specify acceptable cryptographic protocols with the following key-value pair:

```
CryptoProtocolVersion=TLSv1.1,TLSv1.2
```

Notes

- When multiple protocols are specified, the driver uses the highest version supported by the server. If none of the specified protocols are supported by the server, the connection fails and the driver returns an error.
- When no value has been specified for CryptoProtocolVersion, the cryptographic protocol used depends on the highest protocol version supported by the server and the highest protocol version supported by the JDK. The driver uses the lower version of these two protocols to establish the SSL connection. Refer to the database management system documentation for information on which cryptographic protocols are supported.

Data source method

`setCryptoProtocolVersion`

Default

None

Data type

String

See also

- EncryptionMethod on page 218
- Data encryption on page 87

DatabaseName

Purpose

Specifies the name of the database to which you want to connect.

Valid Values

`string`

where:

`string`

is the name of a SQL Server or Azure database.

Notes

Database can be used as an alias for DatabaseName.

Data source method

`setDatabaseName`

Default

None

Data type

String

DateTimeInputParameterType

Purpose

Specifies how the driver describes the data type for Date/Time/Timestamp input parameters.

This property only applies to connections to Azure and SQL Server 2008 and higher. For prior versions of Microsoft SQL Server, the driver always describes Date/Time/Timestamp input parameters as datetime.

Valid values

`auto` | `dateTime` | `dateTimeOffset`

Behavior

If set to `auto`, the driver uses the following rules to describe the data type of Date/Time/Timestamp input parameters:

- If an input parameter is set using setDate(), the driver describes it as date.
- If an input parameter is set using setTime(), the driver describes it as time.
- If an input parameter is set using setTimestamp(), the driver describes it as datetimeoffset.

If set to `dateTime`, the driver describes Date/Time/Timestamp input parameters as datetime.

If set to `dateTimeOffset`, the driver describes Date/Time/Timestamp input parameters as datetimeoffset.

Data source method

`setDateTimeInputParameterType`

Default

`auto`

Data type

String

See also

Data type handling properties on page 64

DateTimeOutputParameterType

Purpose

Specifies how the driver describes the data type for Date/Time/Timestamp output parameters.

This property only applies to connections to Microsoft SQL Server 2008 and higher and Microsoft Windows Azure SQL Database. For connections to prior versions of Microsoft SQL Server, the driver always describes Date/Time/Timestamp output parameters as datetime.

Valid values

`auto` | `dateTime` | `dateTimeOffset`

Behavior

If set to `auto`, the driver uses the following rules to describe the data type of Date/Time/Timestamp output parameters:

- If an output parameter is set using setDate(), the driver describes it as date.
- If an output parameter is set using setTime(), the driver describes it as time.
- If an output parameter is set using setTimestamp(), the driver describes it as datetimeoffset.

If set to `dateTime`, the driver describes Date/Time/Timestamp output parameters as datetime.

If set to `dateTimeOffset`, the driver describes Date/Time/Timestamp output parameters as datetimeoffset.

Data source method

`setDateTimeOutputParameterType`

Default

`auto`

Data type

String

See also

Data type handling properties on page 64

DescribeInputParameters

Purpose

Determines whether the driver attempts to determine, at execute time, which data type to use to send input parameters to the database server. Sending parameters as the data type the database expects improves performance and prevents locking issues caused by data type mismatches.

Valid values

`noDescribe` | `describeIfString` | `describeIfDateTime` | `describeAll`

Behavior

If set to `noDescribe`, the driver does not attempt to describe input parameters and sends String and Date/Time/Timestamp input parameters to the server as specified by the StringInputParameterType and DateTimeInputParameterType properties.

If set to `describeIfString`, the driver submits a request to the database to describe String input parameters. The driver uses the data types that are returned by the driver to determine whether to describe the String input parameters as nvarchar or varchar. If this operation fails, the driver sends String input parameters to the server as specified by the StringInputParameterType property.

If set to `describeIfDateTime`, the driver submits a request to the database to describe Date/Time/Timestamp input parameters. The driver uses the data types that are returned by the driver to determine how to describe the Date/Time/Timestamp input parameters. If this operation fails, the driver sends Date/Time/Timestamp input parameters to the server as specified by the DateTimeInputParameterType property.

If set to `describeAll`, the driver submits a request to the database to describe both String and Date/Time/Timestamp input parameters and uses the data types that are returned by the driver to determine which data type to use to describe the input parameters. If this operation fails, the driver sends String input parameters to the server as specified by the StringInputParameterType property and sends Date/Time/Timestamp input parameters to the server as specified by the DateTimeInputParameterType property.

Data source method

`setDescribeInputParameters`

Default

`noDescribe`

Data type

String

See also

Data type handling properties on page 64

DescribeOutputParameters

Purpose

Determines whether the driver attempts to determine, at execute time, which data type to use to send output parameters to the database server. Sending parameters as the data type the database expects improves performance and prevents locking issues caused by data type mismatches.

Valid values

`noDescribe` | `describeIfString` | `describeIfDateTime` | `describeAll`

Behavior

If set to `noDescribe`, the driver does not attempt to describe output parameters and sends String and Date/Time/Timestamp output parameters to the server as specified by the StringOutputParameterType and DateTimeOutputParameterType properties.

If set to `describeIfString`, the driver submits a request to the database to describe String output parameters. The driver uses the data types that are returned by the driver to determine whether to describe the String output parameters as NVARCHAR or VARCHAR. If this operation fails, the driver sends String output parameters to the server as specified by the StringOutputParameterType property.

If set to `describeIfDateTime`, the driver submits a request to the database to describe Date/Time/Timestamp output parameters. The driver uses the data types that are returned by the driver to determine how to describe the Date/Time/Timestamp output parameters. If this operation fails, the driver sends Date/Time/Timestamp output parameters to the server as specified by the DateTimeOutputParameterType property.

If set to `describeAll`, the driver submits a request to the database to describe both String and Date/Time/Timestamp output parameters and uses the data types that are returned by the driver to determine which data type to use to describe the output parameters. If this operation fails, the driver sends String output parameters to the server as specified by the StringOutputParameterType property and sends Date/Time/Timestamp output parameters to the server as specified by the DateTimeOutputParameterType property.

Data source method

`setDescribeOutputParameters`

Default

`noDescribe`

Data type

String

See also

Data type handling properties on page 64

Domain

Purpose

Specifies the name of the domain server that administers the database. Set this property only if you are using NTLM authentication. If the Domain property is unspecified, the driver tries to determine the domain server name from the User property.

Valid values

`string`

where:

`string`

is the name of the domain server.

Data source method

`setDomain`

Default

None

Data type

String

See also

- Configuring NTLM authentication on page 86
- AuthenticationMethod on page 201

EnableBulkLoad

Purpose

Specifies whether the driver uses the native bulk load protocols in the database. Bulk load bypasses the data parsing that is usually done by the database, providing an additional performance gain over batch operations. This property allows existing applications with batch inserts to take advantage of bulk load without requiring changes to the application code.

Valid Values

`true` | `false`

Behavior

If set to `true`, the driver uses the database bulk load protocol when an application executes an INSERT with multiple rows of parameter data. If the protocol cannot be used, the driver returns a warning.

If set to `false`, the driver uses standard parameter arrays.

Data source method

`setEnableBulkLoad`

Default

`false`

Data type

Boolean

See also

- DataDirect Bulk Load on page 139
- Bulk load properties on page 63
- Performance considerations on page 75

EnableCancelTimeout

Purpose

Determines whether a cancel request that is sent by the driver as the result of a query timing out is subject to the same query timeout value as the statement it cancels.

Valid values

`true | false`

Behavior

If set to `true`, the cancel request times out using the same timeout value, in seconds, that is set for the statement it cancels. For example, if your application calls `Statement.setQueryTimeout(5)` on a statement and that statement is cancelled because its timeout value was exceeded, the driver sends a cancel request that also will time out if its execution exceeds 5 seconds. If the cancel request times out, because the server is down, for example, the driver throws an exception indicating that the cancel request was timed out and the connection is no longer valid.

If set to `false`, the cancel request does not time out.

Data source method

`setEnableCancelTimeout`

Default

`false`

Data type

Boolean

See also

- Timeouts on page 117
- Timeout properties on page 66

EncryptionMethod

Purpose

Determines whether data is encrypted and decrypted when transmitted over the network between the driver and database server.

Valid values

`noEncryption` | `SSL` | `requestSSL` | `loginSSL`

Behavior

If set to `noEncryption`, data is not encrypted or decrypted.

If set to `SSL`, data is encrypted using SSL. If the database server does not support SSL, the connection fails and the driver throws an exception.

If set to `requestSSL`, the login request and data is encrypted using SSL. If the database server does not support SSL, the driver establishes an unencrypted connection.

If set to `loginSSL`, the login request is encrypted using SSL. Data is encrypted using SSL If the database server is configured to require SSL. If the database server does not require SSL, data is not encrypted and only the login request is encrypted.

Notes

- Connection hangs can occur when the driver is configured for SSL and the database server does not support SSL. You may want to set a login timeout using the LoginTimeout property to avoid problems when connecting to a server that does not support SSL.
- If SSL is enabled, the driver communicates with database protocol packets that are set by the server's default packet size. Any value set by the PacketSize property is ignored.
- If SSL is enabled, the following properties also apply:

  CryptoProtocolVersion

  HostNameInCertificate

  TrustStore

  TrustStorePassword

  ValidateServerCertificate

Data source method

`setEncryptionMethod`

Default

`noEncryption`

Data type

String

See also

- Data encryption on page 87
- Data encryption properties on page 60

FailoverGranularity

Purpose

Determines how the driver behaves if exceptions occur while trying to reestablish a lost connection. This property is ignored if FailoverMode=`connect`.

Valid values

`nonAtomic` | `atomic` | `atomicWithRepositioning`

Behavior

If set to `nonAtomic`, the driver continues with the failover process and posts any exceptions on the statement on which they occur.

If set to `atomic`, the driver fails the entire failover process if an exception is generated as the result of restoring the state of the connection. The driver stops trying to connect to an alternative server and returns an exception indicating that the connection was lost. If an exception is generated as a result of restoring the state of work in progress by re-executing the Select statement, the driver continues with the failover process, but generates an exception warning that the Select statement must be reissued.

If set to `atomicWithRepositioning`, the driver fails the entire failover process if any exception is generated as the result of restoring the state of the connection or the state of work in progress. The driver stops trying to connect to an alternative server and returns an exception indicating that the connection was lost.

Data source method

`setFailoverGranularity`

Default

`nonAtomic`

Data type

String

See also

- FailoverMode on page 220
- Using failover on page 92

FailoverMode

Purpose

Specifies the type of failover method the driver uses.

Valid values

`connect` | `extended` | `select`

Behavior

If set to `connect`, the driver provides failover protection for new connections only.

If set to `extended`, the driver provides failover protection for new and lost connections, but not any work in progress.

If set to `select`, the driver provides failover protection for new and lost connections. In addition, it preserves the state of work performed by the last Select statement executed on the Statement object.

Notes

- The AlternateServers property specifies one or multiple alternate servers for failover and is required for all failover methods. To turn off failover, do not specify a value for the AlternateServers property.
- The FailoverGranularity property determines which action the driver takes if exceptions occur during the failover process.
- The FailoverPreconnect property specifies whether the driver tries to connect to multiple database servers (primary and alternate) at the same time.
- When MultiSubnetFailover is enabled, the driver does not support FailoverMode=`select`. If MultiSubnetFailover=`true` and FailoverMode=`select`, the driver downgrades the FailoverMode to `extended` and provides the following warning.

  ```
  failoverMode=select is not supported for AlwaysOn Availability Group,
  downgraded to failoverMode=extended
  ```

Data source method

`setFailoverMode`

Default

`connect`

Data type

String

See also

- Using failover on page 92
- Failover properties on page 61

FailoverPreconnect

Purpose

Specifies whether the driver tries to connect to the primary and an alternate server at the same time. This property is ignored if FailoverMode=`connect`.

Valid values

`true` | `false`

Behavior

If set to `true`, the driver tries to connect to the primary and an alternate server at the same time. This can be useful if your application is time-sensitive and cannot absorb the wait for the failover connection to succeed.

If set to `false`, the driver tries to connect to an alternate server only when failover is caused by an unsuccessful connection attempt or a lost connection. This value provides the best performance, but your application typically experiences a short wait while the failover connection is attempted.

Notes

The AlternateServers property specifies one or multiple alternate servers for failover.

Data source method

`setFailoverPreconnect`

Default

`false`

Data type

Boolean

See also

- FailoverMode on page 220
- Using failover on page 92

FetchTSWTZAsTimestamp

Purpose

Determines whether column values with the datetimeoffset data type are returned as a JDBC VARCHAR or TIMESTAMP data type.

This property only applies to connections to Azure and SQL Server 2008 and higher.

Valid values

`true | false`

Behavior

If set to `true`, column values with the datetimeoffset data type are returned as a JDBC TIMESTAMP data type.

If set to `false`, column values with the datetimeoffset data type are returned as a JDBC VARCHAR data type.

Data source method

`setFetchTSWTZAsTimestamp`

Default

`false`

Data type

Boolean

See also

Data type handling properties on page 64

FetchTWFSasTime

Purpose

Determines whether the driver returns column values for the native TIME data type as the JDBC TIME or TIMESTAMP data type.

Valid Values

`true | false`

Behavior

If set to `true`, the driver returns column values for the native TIME data type as the JDBC TIME data type. The fractional seconds portion of the value is truncated when the value is returned in the java.sql.Time object.

If set to `false`, the driver returns column values for the native TIME data type as the JDBC TIMESTAMP data type. The Java Epoch (Jan 1,1970) is returned in the date portion.

Data source method

`setFetchTWFSasTime`

Default

`false`

Data Type

boolean

See also

Data type handling properties on page 64

GSSCredential

Purpose

Specifies the GSS credential object used to instantiate Kerberos constrained delegation. Constrained delegation is a Kerberos mechanism that allows a client application to delegate authentication to a second service.

Important: Because the value of this property is a Java object, it cannot be specified in a connection URL. It can only be passed as a `Properties` or `DataSource` object.

Valid Values

`string`

where:

`string`

is the name of the GSS credential object.

Notes

- AuthenticationMethod must be set to `kerberos` to use constrained delegation.
- Java SE 8 or higher must be used to generate the GSS credential object for Kerberos constrained delegation.

Data Source Method

`setGSSCredential`

Default

Null

Data type

String

See also

- Authentication on page 79
- Configuring the driver for Kerberos authentication on page 81
- Constrained delegation on page 85

HostNameInCertificate

Purpose

Specifies a host name for certificate validation when SSL encryption is enabled (`EncryptionMethod=SSL`) and validation is enabled (`ValidateServerCertificate=true`). This property is optional and provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.

Valid values

`host_name`

where:

`host_name`

is a valid host name.

Behavior

If *`host_name`* is specified, the driver compares the specified host name to the DNSName value of the SubjectAlternativeName in the certificate. If a DNSName value does not exist in the SubjectAlternativeName or if the certificate does not have a SubjectAlternativeName, the driver compares the host name with the Common Name (CN) part of the certificate's Subject name. If the values do not match, the connection fails and the driver throws an exception.

Notes

- If the HostNameInCertificate is not specified, the driver automatically uses the value of the ServerName from the URL as the value for validating the certificate.
- If SSL encryption or certificate validation is not enabled, this property is ignored.
- If SSL encryption and validation is enabled and this property is unspecified, the driver uses the server name that is specified in the connection URL or data source of the connection to validate the certificate.

Data source method

`setHostNameInCertificate`

Default

Empty string

Data type

String

See also

- EncryptionMethod on page 218
- ValidateServerCertificate on page 251

ImportStatementPool

Purpose

Specifies the path and file name of the file to be used to load the contents of the statement pool. When this property is specified, statements are imported into the statement pool from the specified file.

If the driver cannot locate the specified file when establishing the connection, the connection fails and the driver throws an exception.

Valid values

`string`

where:

`string`

is the path and file name of the file to be used to load the contents of the statement pool.

Data source method

`setImportStatementPool`

Default

Empty string

Data type

String

See also

- Statement Pool Monitor on page 131
- Performance considerations on page 75

InitializationString

Purpose

Specifies one or multiple SQL commands to be executed by the driver after it has established the connection to the database and has performed all initialization for the connection. If the execution of a SQL command fails, the connection attempt also fails and the driver throws an exception indicating which SQL command or commands failed.

Valid values

`string`

where:

`string`

is one or multiple SQL commands.

Multiple commands must be separated by semicolons. In addition, if this property is specified in a connection URL, the entire value must be enclosed in parentheses when multiple commands are specified.

Example

The following connection URL sets the handling of null values to the Microsoft SQL Server default and allows delimited identifiers:

```
jdbc:datadirect:sqlserver://server1:1433;InitializationString=
(set ANSI_NULLS off;set QUOTED_IDENTIFIER on);DatabaseName=test
```

Data source method

`setInitializationString`

Default

None

Data type

String

InsensitiveResultSetBufferSize

Purpose

Determines the amount of memory used by the driver to cache insensitive result set data.

Valid values

`-1` | `0` | *x*

where:

x

is a positive integer that represents the size of the memory buffer.

Behavior

If set to `-1`, the driver caches insensitive result set data in memory. If the size of the result set exceeds available memory, an OutOfMemoryException is generated. With no need to write result set data to disk, the driver processes the data efficiently.

If set to `0`, the driver caches insensitive result set data in memory, up to a maximum of 2 GB. If the size of the result set data exceeds available memory, the driver pages the result set data to disk. Because result set data may be written to disk, the driver may have to reformat the data to write it correctly to disk.

If set to *x*, the driver caches insensitive result set data in memory and uses this value to set the size (in KB) of the memory buffer for caching insensitive result set data. If the size of the result set data exceeds available memory, the driver pages the result set data to disk. Because the result set data may be written to disk, the driver may have to reformat the data to write it correctly to disk. Specifying a buffer size that is a power of 2 results in efficient memory use.

Data source method

`setInsensitiveResultSetBufferSize`

Default

`2048` (KB)

Data type

Int

See also

Performance considerations on page 75

JavaDoubleToString

Purpose

Determines which algorithm the driver uses when converting a double or float value to a string value. By default, the driver uses its own internal conversion algorithm, which improves performance.

Valid values

`true` | `false`

Behavior

If set to `true`, the driver uses the JVM algorithm when converting a double or float value to a string value. If your application cannot accept rounding differences and you are willing to sacrifice performance, set this value to `true` to use the JVM conversion algorithm.

If set to `false`, the driver uses its own internal algorithm when converting a double or float value to a string value. This value improves performance, but slight rounding differences within the allowable error of the double and float data types can occur when compared to the same conversion using the JVM algorithm.

Data source method

`setJavaDoubleToString`

Default

`false`

Data type

Boolean

See also

Data type handling properties on page 64

JDBCBehavior

Purpose

Determines how the driver describes database data types that map to the following JDBC 4.0 data types: NCHAR, NVARCHAR, NLONGVARCHAR, NCLOB, and SQLXML.

Valid values

`0 | 1`

Behavior

If set to `0`, the driver describes the data types as JDBC 4.0 data types.

If set to `1`, the driver describes the data types using JDBC 3.0-equivalent data types. This allows your application to continue using JDBC 3.0 types in a Java SE 6 or higher environment. Additionally, the PROCEDURE_NAME column contains procedure name qualifiers. For example, for the fully qualified procedure named 1.sp_productadd, the driver would return sp_productadd;1.

Data source method

`setJDBCBehavior`

Default

`1`

Data type

Int

See also

Data type handling properties on page 64

LoadBalancing

Purpose

Determines whether the driver uses client load balancing in its attempts to connect to the database servers (primary and alternate). You can specify one or multiple alternate servers by setting the AlternateServers property.

Valid values

`true | false`

Behavior

If set to `true`, the driver uses client load balancing and attempts to connect to the database servers (primary and alternate) in random order. The driver randomly selects from the list of primary and alternate servers which server to connect to first. If that connection fails, the driver again randomly selects from this list of servers until all servers in the list have been tried or a connection is successfully established.

If set to `false`, the driver does not use client load balancing and connects to each server based on their sequential order (primary server first, then, alternate servers in the order they are specified).

Data source method

`setLoadBalancing`

Default

`false`

Data type

Boolean

See also

- AlternateServers on page 198
- Using failover on page 92

LoginConfigName

Purpose

Specifies the name of the entry in the JAAS login configuration file that contains the authentication technology used by the driver to establish a Kerberos connection. The LoginModule-specific items found in the entry are passed on to the LoginModule.

Valid values

`entry_name`

where:

`entry_name`

is the name of the entry that contains the authentication technology used with the driver.

Example

In the following example, `JDBC_DRIVER_01` is the entry name while the authentication technology and related settings are found in the brackets.

```
JDBC_DRIVER_01 {
  com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;
};
```

Data Source Method

`setLoginConfigName`

Default

`JDBC_DRIVER_01`

Data type

String

See also

- Authentication on page 79
- Configuring the driver for Kerberos authentication on page 81
- The JAAS login configuration file on page 83

LoginTimeout

Purpose

The amount of time, in seconds, that the driver waits for a connection to be established before timing out the connection request.

Valid values

`0` | *x*

where:

x

is a number of seconds.

Behavior

If set to `0`, the driver does not time out a connection request.

If set to *x*, the driver waits for the specified number of seconds before returning control to the application and throwing a timeout exception.

Notes

When MultiSubnetFailover is enabled, the value of the LoginTimeout property is 15 seconds by default. When LoginTimeout is set to 0 (zero), the driver will still timeout requests after 15 seconds. However, when the value of LoginTimeout is an integer greater than 0 (zero), the driver will timeout requests for the specified duration.

Data source method

`setLoginTimeout`

Default

`0`

Data type

Int

See also

- Timeouts on page 117
- Timeout properties on page 66

LongDataCacheSize

Purpose

Determines whether the driver caches long data (images, pictures, long text, binary data, or XML data) in result sets. To improve performance, you can disable long data caching if your application retrieves columns in the order in which they are defined in the result set.

Valid values

`-1` | `0` | *x*

where:

x

is a positive integer in KB that represents the size of the memory buffer.

Behavior

If set to `-1`, the driver does not cache long data in result sets. It is cached on the server. Use this value only if your application retrieves columns in the order in which they are defined in the result set.

If set to `0`, the driver caches long data in result sets in memory. If the size of the result set data exceeds available memory, the driver pages the result set data to disk.

If set to *x*, the driver caches long data in result sets in memory and uses this value to set the size in KB of the memory buffer for caching result set data. If the size of the result set data exceeds available memory, the driver pages the result set data to disk.

Data source method

`setLongDataCacheSize`

Default

`2048` (KB)

Data type

Int

See also

Performance considerations on page 75

MaxPooledStatements

Purpose

Specifies the maximum number of prepared statements to be pooled for each connection and enables the driver's internal prepared statement pooling when set to an integer greater than zero (0). The driver's internal prepared statement pooling provides performance benefits when the driver is not running from within an application server or another application that provides its own statement pooling.

Valid values

`0` | *`x`*

where:

`x`

is a positive integer that represents a number of prepared statements to be cached.

Behavior

If set to `0`, the driver's internal prepared statement pooling is not enabled.

If set to *`x`*, the driver's internal prepared statement pooling is enabled and the driver uses the specified value to cache up to that many prepared statements created by an application. If the value set for this property is greater than the number of prepared statements that are used by the application, all prepared statements that are created by the application are cached. Because CallableStatement is a sub-class of PreparedStatement, CallableStatements also are cached.

Notes

- MaxStatements can be used as an alias for MaxPooledStatements.
- When you enable statement pooling, your applications can access the Statement Pool Monitor directly with DataDirect-specific methods. However, you can also enable the Statement Pool Monitor as a JMX MBean. To enable the Statement Pool Monitor as an MBean, statement pooling must be enabled with MaxPooledStatements and the Statement Pool Monitor MBean must be registered using the RegisterStatementPoolMonitorMBean connection property.

Example

If the value of this property is set to `20`, the driver caches the last 20 prepared statements that are created by the application.

Data source method

`setMaxPooledStatements`

Default

`0`

Data type

Int

See also

- Statement Pool Monitor on page 131
- Performance considerations on page 75
- RegisterStatementPoolMonitorMBean on page 238

MultiSubnetFailover

Purpose

Determines whether the driver attempts parallel connections to the failover IP addresses of an Availability Group during initial connection or a multi-subnet failover. When MultiSubnetFailover is enabled, the driver simultaneously attempts to connect to all IP addresses associated with the Availability Group listener when establishing an initial connection or reconnecting after a connection is broken or the listener IP address becomes unavailable. The first IP address to successfully respond to the request is used for the connection. Using parallel-connection attempts offers improved response time over traditional failover, which attempts to connect to alternate servers one at a time.

Valid values

`true` | `false`

Behavior

If set to `true`, the driver attempts parallel connections to all failover IP addresses in an Availability Group when establishing an initial connection or reconnecting after a connection is broken or the listener IP address becomes unavailable. The first IP address to successfully respond to the request is used for the connection. This setting is only supported when your environment is configured for Always On Availability Groups.

If set to `false`, the driver connects to an alternate server or servers as specified by the AlternateServer property when the primary server is unavailable. Use this setting if your environment is not configured for Always On Availability Groups.

Notes

- When MultiSubnetFalover is enabled, the virtual network name (VNN) of the availability group listener must be specified with the ServerName connection property.
- When MultiSubnetFailover is enabled, the driver does not support FailoverMode=`select`. If MultiSubnetFailover=`true` and FailoverMode=`select`, the driver downgrades the FailoverMode to `extended` and provides the following warning.

  ```
  failoverMode=select is not supported for AlwaysOn Availability Group,
  downgraded to failoverMode=extended
  ```

- When MultiSubnetFailover is enabled, the ConnectionRetryDelay connection property is ignored.
- If MultiSubnetFailover is enabled and the connection attempt fails, the driver will attempt to connect two more times, regardless of the ConnectionRetryCount setting.
- When MultiSubnetFailover is enabled, the value of the LoginTimeout property is 15 seconds by default. When LoginTimeout is set to 0 (zero), the driver will still timeout requests after 15 seconds. However, when the value of LoginTimeout is an integer greater than 0 (zero), the driver will timeout requests for the specified duration.

Data source method

`setMultiSubnetFailover`

Default

`false`

Data type

Boolean

See also

- Always On Availability Groups on page 100
- Using failover on page 92

NetAddress

Purpose

The Media Access Control (MAC) address of the network interface card of the application connecting to Microsoft SQL Server. This value is stored in the net_address column of the sys.sysprocesses table.

Valid values

`string`

where:

`string`

is a maximum of 12 alphanumeric characters.

Data source method

`setNetAddress`

Default

`000000000000`

Data type

String

PacketSize

Purpose

Determines the number of bytes for each database protocol packet that is transferred from the database server to the client machine (Microsoft SQL Server refers to this packet as a network packet).

Adjusting the packet size can improve performance. The optimal value depends on the typical size of data that is inserted, updated, or returned by the application and the environment in which it is running. Typically, larger packet sizes work better for large amounts of data. For example, if an application regularly returns character values that are 10,000 characters in length, using a value of 32 (16 KB) typically results in improved performance.

Valid values

`-1` | `0` | *x*

where:

x

is an integer from 1 to 128 that represents a number of bytes.

Behavior

If set to `-1`, the driver uses the maximum packet size that the database server accepts.

If set to `0`, the driver uses the default packet size configured on the database server.

If set to *x*, the driver uses a packet size that is calculated using the specified value multiplied by 512.

Notes

- If SSL encryption is enabled using the EncryptionMethod property, any value set for the PacketSize property is ignored.
- If your application sends queries that only retrieve small result sets, you may want to use a packet size that is smaller than the maximum packet size that is configured on the database server. If a result set that contains only one or two rows of data does not completely fill a larger packet, performance will not improve by setting the value to the maximum packet size.

Example

If PacketSize=8, the packet size is set to 8 * 512 bytes (4096 bytes).

Data source method

`setPacketSize`

Default

`-1`

Data type

Int

See also

Performance considerations on page 75

Password

Purpose

Specifies a password that is used to connect to the database or instance.

Valid values

`string`

where

`string`

is a valid password. The password is case-insensitive.

Data source method

`setPassword`

Default

None

Data type

String

See also

- Authentication on page 79
- User on page 250

PortNumber

Purpose

Specifies the TCP port of the primary database server that is listening for connections to the database.

Valid values

`port`

where:

`port`

is the port number.

Data source method

`setPortNumber`

Default

`1433`

Data type

Int

ProgramID

Purpose

The driver name and version information on the client to be stored in the database. This property sets the hostprocess column in the sysprocesses table.

Valid values

`string`

where:

`string`

is a value that identifies the product and version of the driver on the client.

Example

```
DDJ04200
```

Notes

HostProcess can be used as an alias for ProgramID.

Data source method

`setProgramID`

Default

Empty string

Data type

String

See also

- Using client information on page 104
- Client information properties on page 68

QueryTimeout

Purpose

Sets the default query timeout (in seconds) for all statements that are created by a connection.

Valid values

`-1` | `0` | *x*

where:

x

is a number of seconds.

Behavior

If set to `-1`, the query timeout functionality is disabled. The driver silently ignores calls to the Statement.setQueryTimeout() method.

If set to `0`, the default query timeout is infinite (the query does not time out).

If set to *x*, the driver uses the value as the default timeout for any statement that is created by the connection. To override the default timeout value set by this connection option, call the Statement.setQueryTimeout() method to set a timeout value for a particular statement.

Data source method

`setQueryTimeout`

Default

`0`

Data type

Int

See also

- Timeouts on page 117
- Timeout properties on page 66

RegisterStatementPoolMonitorMBean

Purpose

Registers the Statement Pool Monitor as a JMX MBean when statement pooling has been enabled with MaxPooledStatements. This allows you to manage statement pooling with standard JMX API calls and to use JMX-compliant tools, such as JConsole.

Valid values

`true | false`

Behavior

If set to `true`, the driver registers an MBean for the statement pool monitor for each statement pool. This gives applications access to the Statement Pool Monitor through JMX when statement pooling is enabled.

If set to `false`, the driver does not register an MBean for the statement pool monitor for any statement pool.

Notes

Registering the MBean exports a reference to the Statement Pool Monitor. The exported reference can prevent garbage collection on connections if the connections are not properly closed. When garbage collection does not take place on these connections, out of memory errors can occur.

Data source method

`setRegisterStatementPoolMonitorMBean`

Default

`false`

Data type

Boolean

See also

- Statement Pool Monitor on page 131
- MaxPooledStatements on page 232

ResultSetMetaDataOptions

Purpose

Determines whether the driver returns table name information in the ResultSet metadata for Select statements.

Valid values

`0 | 1`

Behavior

If set to `0` and the ResultSetMetaData.getTableName() method is called, the driver does not perform additional processing to determine the correct table name for each column in the result set. The getTableName() method may return an empty string for each column in the result set.

If set to `1` and the ResultSetMetaData.getTableName() method is called, the driver performs additional processing to determine the correct table name for each column in the result set. The driver returns schema name and catalog name information when the ResultSetMetaData.getSchemaName() and ResultSetMetaData.getCatalogName() methods are called if the driver can determine that information.

Data source method

`setResultSetMetaDataOptions`

Default

`0`

Data type

Int

See also

Performance considerations on page 75

SelectMethod

Purpose

A hint to the driver that determines whether the driver requests a database cursor for Select statements. Performance and behavior of the driver are affected by this property, which is defined as a hint because the driver may not always be able to satisfy the requested method.

Valid values

`direct | cursor`

Behavior

If set to `direct`, the database server sends the complete result set in a single response to the driver when responding to a query. A server-side database cursor is not created if the requested result set type is a forward-only result set. Typically, responses are not cached by the driver. Using this method, the driver must process the entire response to a query before another query is submitted. If another query is submitted (using a different statement on the same connection, for example), the driver caches the response to the first query before submitting the second query. Typically, the direct method performs better than the cursor method.

If set to `cursor`, a server-side cursor is requested. When returning forward-only result sets, the rows are returned from the server in blocks. The setFetchSize() method can be used to control the number of rows that are returned for each request when forward-only result sets are returned. Performance tests show that, when returning forward-only result sets, the value of Statement.setFetchSize() significantly impacts performance. There is no simple rule for determining the setFetchSize() value that you should use. We recommend that you experiment with different setFetchSize() values to determine which value gives the best performance for your application. The cursor method is useful for queries that produce a large amount of data, particularly if multiple open result sets are used.

Notes

SelectMethod=`cursor` is not supported for Microsoft Azure Synapse Analytics or Microsoft Analytics Platform System. For these environments, the database server sends the complete result set in a single response to the driver when responding to a query, and a server-side cursor is not created.

Data source method

`setSelectMethod`

Default

`direct`

Data type

String

See also

Performance considerations on page 75

ServerName

Purpose

Specifies the name or IP address of the server to which you want to connect.

Valid values

`IP_address` | *`named_server`* | *`named_instance`* | *`virtual_network_name`*

where:

`IP_address`

is the IP address of the server to which you want to connect. For example, you can enter `199.226.224.34`. The IP address can be specified in either IPv4 or IPv6 format, or a combination of the two. See Using IP addresses on page 106 for details about these formats.

`named_server`

is the named server address of the server to which you want to connect. For example, you can enter `MyServer`.

`named_instance`

is a named instance of Microsoft SQL Server. The named instance should be specified as *`server_name`*`\\`*`instance_name`*, where *`server_name`* is the IP address and *`instance_name`* is the name of the instance to which you want to connect on the specified server. For example, `server1\\instance1`.

`virtual_network_name`

is the virtual network name (VNN) of the availability group listener when using an Always On Availability Group.

Data source method

`setServerName`

Default

None

Data type

String

See also

Connecting to named instances on page 77

ServicePrincipalName

Purpose

Specifies the service principal name to be used for Kerberos authentication.

Valid Values

`ServicePrincipalName`

where:

`ServicePrincipalName`

is the four-part service principal name registered with the key distribution center (KDC).

Specify the service principal name using the following format.

`Service_Name/Fully_Qualified_Domain_Name``:`*`Port_Number@REALM`*`.COM`

where:

`Service_Name`

is the name of the service hosting the instance. The *`Service_Name`* for Microsoft SQL Server is `MSSQLSvc`.

`Fully_Qualified_Domain_Name`

is the fully qualified domain name (FQDN) of the host machine. This value must match the FQDN registered with the KDC. The FQDN consists of a host name and a domain name. For the example `myserver.example.com`, `myserver` is the host name and `example.com` is the domain name.

`Port_Number`

is the port number as specified by the PortNumber property.

`REALM.COM`

is the domain name of the host machine. This value is optional. If no value is specified, the default domain is used. The domain must specified in upper-case characters. For example, `EXAMPLE.COM`. For Windows Active Directory, the Kerberos realm name is the Windows domain name.

Example

The following is an example of a valid service principal name:

```
MSSQLSvc/myserver.example.com:1433@EXAMPLE.COM
```

Notes

- The driver builds a service principal name in the following manner.
    - `MSSQLSvc` is used as the service name.
    - The value of the ServerName property is used as the FQDN.
    - The PortNumber property specifies the port number that is used.
    - The default realm in the `krb5.conf` file is used as the realm name.
- If the default does not match the service principal name registered with the KDC, then you can specify the value of the service principal name registered with the KDC.
- In a Kerberos configuration, an IP address cannot be used as a FQDN.

Data Source Method

`setServicePrincipalName`

Default

Driver builds value based on environment

Data type

String

See also

- Authentication on page 79
- AuthenticationMethod on page 201
- Configuring the driver for Kerberos authentication on page 81

SnapshotSerializable

Purpose

Allows your application to use Snapshot Isolation for connections.

This property is useful for applications that have the Serializable isolation level set. Using the SnapshotSerializable property allows you to use Snapshot Isolation with no or minimum code changes. If you are developing a new application, you may find that using the constant TRANSACTION_SNAPSHOT is a better choice.

Valid values

`true` | `false`

Behavior

If set to `true` and your application has the transaction isolation level set to Serializable, the application uses Snapshot Isolation for connections.

If set to `false` and your application has the transaction isolation level set to Serializable, the application uses the Serializable isolation level.

Notes

To use Snapshot Isolation, your database also must be configured for Snapshot Isolation.

Data source method

`setSnapshotSerializable`

Default

`false`

Data type

Boolean

See also

- Isolation levels on page 109
- Using the Snapshot isolation level on page 109
- Performance considerations on page 75

SpyAttributes

Purpose

Enables DataDirect Spy to log detailed information about calls issued by the driver on behalf of the application. DataDirect Spy is not enabled by default.

Valid values

`(` *`spy_attribute`* `[;` *`spy_attribute`* `]...)`

where:

`spy_attribute`

is any valid DataDirect Spy attribute. See to DataDirect Spy attributes on page 182 for a list of supported attributes.

Notes

- If coding a path on Windows to the log file in a Java string, the backslash character (\) must be preceded by the Java escape character, a backslash. For example: `log=(file)C:\\temp\\spy.log`.

Example

The following value instructs the driver to log all JDBC activity to a file using a maximum of 80 characters for each line.

```
 (log=(file)/tmp/spy.log;linelimit=80)
```

Data source method

`setSpyAttributes`

Default

None

Data type

String

See also

- Tracking JDBC calls with DataDirect Spy on page 180
- DataDirect Spy attributes on page 182

StringInputParameterType

Purpose

Determines whether the driver sends String input parameters to the database in Unicode or in the default character encoding of the database.

Valid values

`nvarchar` | `varchar`

Behavior

If set to `nvarchar`, the driver sends String input parameters to the database in Unicode.

If set to `varchar`, the driver sends String input parameters to the database in the default character encoding of the database. This value can improve performance because the server does not need to convert Unicode characters to the default encoding.

Notes

If a value is specified for the CodePageOverride property and this property is set to `nvarchar`, this property is ignored and a warning is generated.

Data source method

`setStringInputParameterType`

Default

`nvarchar`

Data type

String

See also

Performance considerations on page 75

StringOutputParameterType

Purpose

Determines whether the driver sends String output parameters to the database server in Unicode or in the default character encoding of the database.

Valid values

`nvarchar` | `varchar`

Behavior

If set to `nvarchar`, the driver sends String output parameters to the database as nvarchar(4000). Use this value when all output parameters that are returned in the connection are nchar or nvarchar. If the output parameters are char or varchar, the driver returns the output parameter value, but the returned value is limited to 4000 characters.

If set to `varchar`, the driver sends String output parameters to the database as varchar(8000). Use this value if all output parameters that are returned in the connection are char or varchar. If an output parameter is nchar or nvarchar, the value may not be returned correctly (for example, if the returned value uses a code page other than the default encoding).

Data source method

`setStringOutputParameterType`

Default

`nvarchar`

Data type

String

See also

Performance considerations on page 75

SuppressConnectionWarnings

Purpose

Determines whether the driver suppresses "changed database" and "changed language" warnings when connecting to the database server.

Valid values

`true` | `false`

Behavior

If set to `true`, warnings are suppressed.

If set to `false`, warnings are not suppressed.

Data source method

`setSuppressConnectionWarnings`

Default

`false`

Data type

Boolean

TransactionMode

Purpose

Controls how the driver delimits the start of a local transaction.

Valid values

`implicit` | `explicit`

Behavior

If set to `implicit`, the driver uses implicit transaction mode. This means that the database, not the driver, automatically starts a transaction when a transactionable statement is executed. Typically, implicit transaction mode is more efficient than explicit transaction mode because the driver does not have to send commands to start a transaction and a transaction is not started until it is needed. When TRUNCATE TABLE statements are used with implicit transaction mode, the database may roll back the transaction if an error occurs. If this occurs, use the explicit value for this property.

If set to `explicit`, the driver uses explicit transaction mode. This means that the driver, not the database starts a new transaction if the previous transaction was committed or rolled back.

Data source method

`setTransactionMode`

Default

`implicit`

Data type

String

TruncateFractionalSeconds

Purpose

Determines whether the driver truncates timestamp values to three fractional seconds. For example, a value of the datetime2 data type can have a maximum of seven fractional seconds.

Valid values

`true | false`

Behavior

If set to `true`, the driver truncates all timestamp values to three fractional seconds.

If set to `false`, the driver does not truncate fractional seconds.

Data source method

`setTruncateFractionalSeconds`

Default

`true`

Data type

Boolean

TrustStore

Purpose

Specifies the directory of the truststore file to be used when SSL is enabled (`EncryptionMethod=SSL`) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

This value overrides the directory of the truststore file that is specified by the javax.net.ssl.trustStore Java system property. If this property is not specified, the truststore directory is specified by the javax.net.ssl.trustStore Java system property.

This property is ignored if ValidateServerCertificate=`false`.

Valid values

`string`

where:

`string`

is the directory of the truststore file.

Data source method

`setTrustStore`

Default

None

Data type

String

See also

- EncryptionMethod on page 218
- ValidateServerCertificate on page 251

TrustStorePassword

Purpose

Specifies the password that is used to access the truststore file when SSL is enabled (`EncryptionMethod=SSL`) and server authentication is used. The truststore file contains a list of the Certificate Authorities (CAs) that the client trusts.

This value overrides the password of the truststore file that is specified by the javax.net.ssl.trustStorePassword Java system property. If this property is not specified, the truststore password is specified by the javax.net.ssl.trustStorePassword Java system property.

This property is ignored if ValidateServerCertificate=`false`.

Valid values

`string`

where:

`string`

is a valid password for the truststore file.

Data source method

`setTrustStorePassword`

Default

None

Data type

String

See also

- EncryptionMethod on page 218

- ValidateServerCertificate on page 251

User

Purpose

Specifies the user ID for user ID/password authentication or the domain user name for NTLM authentication.

Valid values

`[`*`domain_name`*`\]`*`user_name`*

where:

`domain_name`

is the name of a valid domain server. The name is case-insensitive and optional. If specified, you must separate the domain name from the user name by a backward slash (\).

`user_name`

is a valid user name. It is case-insensitive.

Notes

Only set the domain server name if AuthenticationMethod=`ntlmjava` or `ntlm2java`.

Example

`Smith` or `DOMAIN1\Smith`

Data source method

`setUser`

Default

None

Data type

String

See also

- Authentication on page 79
- Password on page 236

UseServerSideUpdatableCursors

Purpose

Determines whether the driver uses server-side cursors when an updatable result set is requested.

Valid values

`true` | `false`

Behavior

If set to `true`, server-side updatable cursors are created when an updatable result set is requested.

If set to `false`, the client-side updatable cursors are created when an updatable result set is requested.

Notes

Server-side updatable cursors are not supported for Microsoft Azure Synapse Analytics or Microsoft Analytics Platform System.

Data source method

`setUseServerSideUpdatableCursors`

Default

`false`

Data type

Boolean

See also

- Server-side updatable cursors on page 110
- Performance considerations on page 75

ValidateServerCertificate

Purpose

Determines whether the driver validates the certificate that is sent by the database server when SSL encryption is enabled (`EncryptionMethod=SSL`). When using SSL server authentication, any certificate that is sent by the server must be issued by a trusted Certificate Authority (CA).

Allowing the driver to trust any certificate that is returned from the server even if the issuer is not a trusted CA is useful in test environments because it eliminates the need to specify truststore information on each client in the test environment.

Valid values

`true | false`

Behavior

If set to `true`, the driver validates the certificate that is sent by the database server. Any certificate from the server must be issued by a trusted CA in the truststore file. If the HostNameInCertificate property is specified, the driver also validates the certificate using a host name. The HostNameInCertificate property is optional and provides additional security against man-in-the-middle (MITM) attacks by ensuring that the server the driver is connecting to is the server that was requested.

If set to `false`, the driver does not validate the certificate that is sent by the database server. The driver ignores any truststore information that is specified by the TrustStore and TrustStorePassword properties or Java system properties.

Notes

Truststore information is specified using the TrustStore and TrustStorePassword properties or by using Java system properties.

Data source method

`setValidateServerCertificate`

Default

`true`

Data type

Boolean

See also

- EncryptionMethod on page 218
- HostNameInCertificate on page 224

XATransactionGroup

Purpose

The transaction group ID that identifies any distributed transactions that are initiated by the connection. This ID can be used for distributed transaction cleanup purposes.

You can use the XAResource.recover method to roll back any transactions left in an unprepared state. When you call XAResource.recover, any unprepared transactions that match the ID on the connection used to call XAResource.recover are rolled back.

Valid values

`string`

where:

string

is a valid transaction group ID.

Example

If you specify `XATransactionGroup=ACCT200` and call XAResource.recover on the same connection, any transactions that are left in an unprepared state identified by the transaction group ID of ACCT200 are rolled back.

Data source method

`setXATransactionGroup`

Default

None

Data type

String

See also

Distributed transaction cleanup on page 112

XMLDescribeType

Purpose

Determines whether the driver maps XML data to the LONGVARCHAR or LONGVARBINARY data type.

Valid values

`longvarchar` | `longvarbinary`

Behavior

If set to `longvarchar`, the driver maps XML data to the LONGVARCHAR data type.

If set to `longvarbinary`, the driver maps XML data to the LONGVARBINARY data type.

Data source method

`setXMLDescribeType`

Default

None

Data type

String

See also

- Returning and inserting/updating XML data on page 100
- Data type handling properties on page 64

5

Troubleshooting

This section provides information that can help you troubleshoot problems when they occur.

For details, see the following topics:

- Troubleshooting your application
- Troubleshooting connection pooling
- Troubleshooting statement pooling
- Configuring logging

Troubleshooting your application

To help you troubleshoot any problems that occur with your application, you can use DataDirect Spy to log detailed information about calls issued by the drivers on behalf of your application. When you enable DataDirect Spy for a connection, you can customize DataDirect Spy logging by setting one or multiple options. See "Tracking JDBC calls with DataDirect Spy" for information about using DataDirect Spy and instructions on enabling and customizing logging.

See also

Tracking JDBC calls with DataDirect Spy on page 180

Turning on and off DataDirect Spy logging

Once DataDirect Spy logging is enabled for a connection, you can turn on and off the logging at runtime using the setEnableLogging() method in the com.ddtek.jdbc.extensions.ExtLogControl interface. When DataDirect Spy logging is enabled, all Connection objects returned to an application provide an implementation of the ExtLogControl interface.

The following code example shows how to turn off logging using setEnableLogging(false).

```
import com.ddtek.jdbc.extensions.*

// Get Database Connection
Connection con = DriverManager.getConnection
   ("jdbc:datadirect:sqlserver://MyServer:1433;User=TEST;Password=secret;
      SpyAttributes=(log=(file)/tmp/spy.log");

((ExtLogControl) con).setEnableLogging(false);
...
```

The setEnableLogging() method only turns on and off logging if DataDirect Spy logging has already been enabled for a connection; it does not set or change DataDirect Spy attributes. See "Enabling DataDirect Spy" for information about enabling and customizing DataDirect Spy logging.

See also

Enabling DataDirect Spy on page 180

DataDirect Spy log example

This section provides information to help you understand the content of your own DataDirect Spy logs.

For example, suppose your application executes the following code and performs some operations:

```
Class.forName("com.ddtek.jdbc.sqlserver.SQLServerDriver");
DriverManager.getConnection("jdbc:datadirect:sqlserver://nc-myserver\
\sqlserver2005;useServerSideUpdatableCursors=true;resultsetMetaDataOptions=1;
sendStringParametersAsUnicode=true;alwaysReportTriggerResults=false;
spyAttributes=(log=(file)c:\\temp\\spy.log)","test04", "test04");
```

The log file generated by DataDirect Spy would look similar to the following example. Notes provide explanations for the referenced text.

```
spy>> Connection[1].getMetaData()
spy>> OK (DatabaseMetaData[1])

spy>> DatabaseMetaData[1].getURL()
spy>> OK
(jdbc:datadirect:sqlserver://nc-myserver\sqlserver2005:1433;CONNECTIONRETRYCOUNT=5;
RECEIVESTRINGPARAMETERTYPE=nvarchar;ALTERNATESERVERS=;DATABASENAME=;PACKETSIZE=16;INITIALIZATIONSTRING=;
ENABLECANCELTIMEOUT=false;BATCHPERFORMANCEWORKAROUND=false;AUTHENTICATIONMETHOD=auto;
SENDSTRINGPARAMETERSASUNICODE=true;LOGINTIMEOUT=0;WSID=;SPYATTRIBUTES=(log=(file)c:\temp\spy.log);
RESULTSETMETADATAOPTIONS=1;ALWAYSREPORTTRIGGERRESULTS=false;TRANSACTIONMODE=implicit;
USESERVERSIDEUPDATABLECURSORS=true;SNAPSHOTSERIALIZABLE=false;JAVADOUBLETOSTRING=false;
SELECTMETHOD=direct;LOADLIBRARYPATH=;CONNECTIONRETRYDELAY=1;INSENSITIVERESULTSETBUFFERSIZE=2048;
MAXPOOLEDSTATEMENTS=0;DESCRIBEPARAMETERS=noDescribe;CODEPAGEOVERRIDE=;NETADDRESS=000000000000;
PROGRAMNAME=;LOADBALANCING=false;HOSTPROCESS=0)[9]
spy>> DatabaseMetaData[1].getDriverName()
spy>> OK (SQLServer)

spy>> DatabaseMetaData[1].getDriverVersion()
spy>> OK (3.60.0 (000000.000000.000000))
```

[9] The combination of the URL specified by the application and the default values of all connection properties not specified.

```
spy>> DatabaseMetaData[1].getDatabaseProductName()
spy>> OK (Microsoft SQL Server)

spy>> DatabaseMetaData[1].getDatabaseProductVersion()
spy>> OK (Microsoft SQL Server Yukon - 9.00.1399)

spy>> Connection Options :[10]
spy>>    CONNECTIONRETRYCOUNT=5
spy>>    RECEIVESTRINGPARAMETERTYPE=nvarchar
spy>>    ALTERNATESERVERS=
spy>>    DATABASENAME=
spy>>    PACKETSIZE=16
spy>>    INITIALIZATIONSTRING=
spy>>    ENABLECANCELTIMEOUT=false
spy>>    BATCHPERFORMANCEWORKAROUND=false
spy>>    AUTHENTICATIONMETHOD=auto
spy>>    SENDSTRINGPARAMETERSASUNICODE=true
spy>>    LOGINTIMEOUT=0
spy>>    WSID=
spy>>    SPYATTRIBUTES=(log=(file)c:\temp\spy.log)
spy>>    RESULTSETMETADATAOPTIONS=1
spy>>    ALWAYSREPORTTRIGGERRESULTS=false
spy>>    TRANSACTIONMODE=implicit
spy>>    USESERVERSIDEUPDATABLECURSORS=true
spy>>    SNAPSHOTSERIALIZABLE=false
spy>>    JAVADOUBLETOSTRING=false
spy>>    SELECTMETHOD=direct
spy>>    LOADLIBRARYPATH=
spy>>    CONNECTIONRETRYDELAY=1
spy>>    INSENSITIVERESULTSETBUFFERSIZE=2048
spy>>    MAXPOOLEDSTATEMENTS=0
spy>>    DESCRIBEPARAMETERS=noDescribe
spy>>    CODEPAGEOVERRIDE=
spy>>    NETADDRESS=000000000000
spy>>    PROGRAMNAME=
spy>>    LOADBALANCING=false
spy>>    HOSTPROCESS=0
spy>> Driver Name = SQLServer[11]
spy>> Driver Version = 3.60.0 (000000.000000.000000)[12]
spy>> Database Name = Microsoft SQL Server[13]
spy>> Database Version = Microsoft SQL Server Yukon - 9.00.1399[14]
spy>> Connection[1].getWarnings()
spy>> OK[15]spy>> Connection[1].createStatement
spy>> OK (Statement[1])

spy>> Statement[1].executeQuery(String sql)
spy>> sql = select empno,ename,job from emp where empno=7369
spy>> OK (ResultSet[1])[16]
spy>> ResultSet[1].getMetaData()
spy>> OK (ResultSetMetaData[1])[17]
spy>> ResultSetMetaData[1].getColumnCount()
spy>> OK (3)[18]
spy>> ResultSetMetaData[1].getColumnLabel(int column)
spy>> column = 1
spy>> OK (EMPNO)[19]spy>> ResultSetMetaData[1].getColumnLabel(int column)
spy>> column = 2
spy>> OK (ENAME)[20]
spy>> ResultSetMetaData[1].getColumnLabel(int column)
```

[10] The combination of the connection properties specified by the application and the default values of all connection properties not specified.

[11] The name of the driver.

[12] The version of the driver.

[13] The name of the database server to which the driver connects.

[14] The version of the database to which the driver connects.

[15] The application checks to see if there are any warnings. In this example, no warnings are present.

[16] The statement SELECT empno,ename,job FROM emp WHERE empno=7369 is created.

[17] Some metadata is requested.

[18] Some metadata is requested.

[19] Some metadata is requested.

[20] Some metadata is requested.

```
spy>> column = 3
spy>> OK (JOB)[21]spy>> ResultSet[1].next()
spy>> OK (true)[22]
spy>> ResultSet[1].getString(int columnIndex)
spy>> columnIndex = 1
spy>> OK (7369)[23]
spy>> ResultSet[1].getString(int columnIndex)
spy>> columnIndex = 2
spy>> OK (SMITH)[24]
spy>> ResultSet[1].getString(int columnIndex)
spy>> columnIndex = 3
spy>> OK (CLERK)[25]
spy>> ResultSet[1].next()
spy>> OK (false)[26]spy>> ResultSet[1].close()
spy>> OK[27]
spy>> Connection[1].close()
spy>> OK[28]
```

Troubleshooting connection pooling

Connection pooling allows connections to be reused rather than created each time a connection is requested. If your application is using connection pooling through the DataDirect Connection Pool Manager, you can generate a trace file that shows all the actions taken by the Pool Manager. See "Connection Pool Manager" for information about using the Pool Manager.

See also

Connection Pool Manager on page 117

Enabling tracing with the setTracing method

You can enable Pool Manager logging by calling `setTracing(true)` on the `PooledConnectionDataSource` connection. To disable tracing, call `setTracing(false)` on the connection.

By default, the DataDirect Connection Pool Manager logs its pool activities to the standard output `System.out`. You can change where the Pool Manager trace information is written by calling the `setLogWriter()` method on the `PooledConnectionDataSource` connection.

Pool Manager trace file example

The following example shows a DataDirect Connection Pool Manager trace file. Notes provide explanations for the referenced text to help you understand the content of your own Pool Manager trace files.

```
jdbc/SQLServerNCMarkBPool: *** ConnectionPool Created
   (jdbc/SQLServerNCMarkBPool,
   com.ddtek.jdbcx.sqlserver.SQLServerDataSource@1835282, 5, 5, 10, scott)[29]
```

[21] Some metadata is requested.
[22] The first row is retrieved and the application retrieves the result values.
[23] The first row is retrieved and the application retrieves the result values.
[24] The first row is retrieved and the application retrieves the result values.
[25] The first row is retrieved and the application retrieves the result values.
[26] The application attempts to retrieve the next row, but only one row was returned for this query.
[27] After the application has completed retrieving result values, the result set is closed.
[28] The application finishes and disconnects.
[29] The Pool Manager creates a connection pool. In this example, the characteristics of the connection pool are shown using the following format:

```
jdbc/SQLServerNCMarkBPool: Number pooled connections = 0.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

jdbc/SQLServerNCMarkBPool: Enforced minimum![30]
NrFreeConnections was: 0
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 5.

jdbc/SQLServerNCMarkBPool: Reused free connection.[31]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 4.

jdbc/SQLServerNCMarkBPool: Reused free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 3.

jdbc/SQLServerNCMarkBPool: Reused free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 2.

jdbc/SQLServerNCMarkBPool: Reused free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 1.

jdbc/SQLServerNCMarkBPool: Reused free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

jdbc/SQLServerNCMarkBPool: Created new connection.[32]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 6.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

jdbc/SQLServerNCMarkBPool: Created new connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 7.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

jdbc/SQLServerNCMarkBPool: Created new connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 8.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

jdbc/SQLServerNCMarkBPool: Created new connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 9.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

jdbc/SQLServerNCMarkBPool: Created new connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 10.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.
```

(*JNDI_name*,*DataSource_class*,*initial_pool_size*,*min_pool_size*,*max_pool_size*, *user*)

where:

JNDI_name is the JNDI name used to look up the connection pool (for example, jdbc/SQLServerNCMarkBPool).

DataSource_class is the DataSource class associated with the connection pool (for example com.ddtek.jdbcx.sqlserver.SQLServerDataSource).

initial_pool_size is the number of physical connections created when the connection pool is initialized (for example, 5).

min_pool_size is the minimum number of physical connections be kept open in the connection pool (for example, 5).

max_pool_size is the maximum number of physical connections allowed within a single pool at any one time. When this number is reached, additional connections that would normally be placed in a connection pool are closed (for example, 10).

user is the name of the user establishing the connection (for example, scott).

[30] The Pool Manager checks the pool size. Because the minimum pool size is five connections, the Pool Manager creates new connections to satisfy the minimum pool size.

[31] The driver requests a connection from the connection pool. The driver retrieves an available connection.

[32] The driver requests a connection from the connection pool. Because a connection is unavailable, the Pool Manager creates a new connection for the request.

```
jdbc/SQLServerNCMarkBPool: Created new connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.[33]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 1.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 2.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 3.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 4.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 5.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 6.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 7.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 8.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 9.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 10.

jdbc/SQLServerNCMarkBPool: Connection was closed and added to the cache.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 11.

jdbc/SQLServerNCMarkBPool: Enforced minimum![34]
NrFreeConnections was: 11
jdbc/SQLServerNCMarkBPool: Number pooled connections = 11.
jdbc/SQLServerNCMarkBPool: Number free connections = 11.

jdbc/SQLServerNCMarkBPool: Enforced maximum![35]
NrFreeConnections was: 11
jdbc/SQLServerNCMarkBPool: Number pooled connections = 10.
jdbc/SQLServerNCMarkBPool: Number free connections = 10.

jdbc/SQLServerNCMarkBPool: Enforced minimum!
NrFreeConnections was: 10
jdbc/SQLServerNCMarkBPool: Number pooled connections = 10.
jdbc/SQLServerNCMarkBPool: Number free connections = 10.
```

[33] A connection is closed by the application and returned to the connection pool.

[34] The Pool Manager checks the pool size. Because the number of connections in the connection pool is greater than the minimum pool size, five connections, no action is taken by the Pool Manager.

[35] The Pool Manager checks the pool size. Because the number of connections in the connection pool is greater than the maximum pool size, 10 connections, a connection is closed and discarded from the pool.

```
jdbc/SQLServerNCMarkBPool: Enforced maximum!
NrFreeConnections was: 10
jdbc/SQLServerNCMarkBPool: Number pooled connections = 10.
jdbc/SQLServerNCMarkBPool: Number free connections = 10.

jdbc/SQLServerNCMarkBPool: Enforced minimum!
NrFreeConnections was: 10
jdbc/SQLServerNCMarkBPool: Number pooled connections = 10.
jdbc/SQLServerNCMarkBPool: Number free connections = 10.

jdbc/SQLServerNCMarkBPool: Enforced maximum!
NrFreeConnections was: 10
jdbc/SQLServerNCMarkBPool: Number pooled connections = 10.
jdbc/SQLServerNCMarkBPool: Number free connections = 10.

jdbc/SQLServerNCMarkBPool: Dumped free connection.[36]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 9.
jdbc/SQLServerNCMarkBPool: Number free connections = 9.

jdbc/SQLServerNCMarkBPool: Dumped free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 8.
jdbc/SQLServerNCMarkBPool: Number free connections = 8.

jdbc/SQLServerNCMarkBPool: Dumped free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 7.
jdbc/SQLServerNCMarkBPool: Number free connections = 7.

jdbc/SQLServerNCMarkBPool: Dumped free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 6.
jdbc/SQLServerNCMarkBPool: Number free connections = 6.

jdbc/SQLServerNCMarkBPool: Dumped free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 5.

jdbc/SQLServerNCMarkBPool: Dumped free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 4.
jdbc/SQLServerNCMarkBPool: Number free connections = 4.

jdbc/SQLServerNCMarkBPool: Dumped free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 3.
jdbc/SQLServerNCMarkBPool: Number free connections = 3.

jdbc/SQLServerNCMarkBPool: Dumped free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 2.
jdbc/SQLServerNCMarkBPool: Number free connections = 2.

jdbc/SQLServerNCMarkBPool: Dumped free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 1.
jdbc/SQLServerNCMarkBPool: Number free connections = 1.

jdbc/SQLServerNCMarkBPool: Dumped free connection.
jdbc/SQLServerNCMarkBPool: Number pooled connections = 0.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.

jdbc/SQLServerNCMarkBPool: Enforced minimum![37]
NrFreeConnections was: 0
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 5.

jdbc/SQLServerNCMarkBPool: Enforced maximum!
NrFreeConnections was: 5
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
```

[36] The Pool Manager detects that a connection was idle in the connection pool longer than the maximum idle timeout. The idle connection is closed and discarded from the pool.

[37] The Pool Manager detects that the number of connections dropped below the limit set by the minimum pool size, five connections. The Pool Manager creates new connections to satisfy the minimum pool size.

```
jdbc/SQLServerNCMarkBPool: Number free connections = 5.

jdbc/SQLServerNCMarkBPool: Closing a pool of the group
      jdbc/SQLServerNCMarkBPool[38]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 5.
jdbc/SQLServerNCMarkBPool: Number free connections = 5.

jdbc/SQLServerNCMarkBPool: Pool closed[39]
jdbc/SQLServerNCMarkBPool: Number pooled connections = 0.
jdbc/SQLServerNCMarkBPool: Number free connections = 0.
```

Troubleshooting statement pooling

Similar to connection pooling, statement pooling provides performance gains for applications that execute the same SQL statements multiple times in the life of the application. The DataDirect Statement Pool Monitor provides the following functionality to help you troubleshoot problems that may occur with statement pooling:

- You can generate a statement pool export file that shows you all statements in the statement pool. Each statement pool entry in the file includes information about statement characteristics such as the SQL text used to generate the statement, statement type, result set type, and result set concurrency type.
- You can use the following methods of the `ExtStatementPoolMonitorMBean` interface to return useful information to determine if your workload is using the statement pool effectively:
  - The `getHitCount` method returns the hit count for the statement pool. The hit count should be high for good performance.
  - The `getMissCount` method returns the miss count for the statement pool. The miss count should be low for good performance.

See also

Statement Pool Monitor on page 131

Generating an export file with the exportStatement method

You can generate an export file by calling the `exportStatements` method of the `ExtStatementPoolMonitorMBean` interface. For example, the following code exports the contents of the statement pool associated with the connection to a file named `stmt_export`.

```
ExtStatementPoolMonitor monitor =
   ((ExtConnection) con).getStatementPoolMonitor();
exportStatements(stmt_export.txt)
```

[38] The Pool Manager closes one of the connection pools in the pool group. A pool group is a collection of pools created from the same PooledConnectionDataSource call. Different pools are created when different user IDs are used to retrieve connections from the pool. A pool group is created for each user ID that requests a connection. In our example, because only one user ID was used, only one pool group is closed.

[39] The Pool Manager closed all the pools in the pool group. The connection pool is closed.

Statement pool export file example

The following example shows a sample export file. The footnotes provide explanations for the referenced text to help you understand the content of your own statement pool export files.

```
[DDTEK_STMT_POOL][40]
VERSION=1[41]

[STMT_ENTRY][42]
SQL_TEXT=[
INSERT INTO emp(id, name) VALUES(?,?)
]
STATEMENT_TYPE=Prepared Statement
RESULTSET_TYPE=Forward Only
RESULTSET_CONCURRENCY=Read Only
AUTOGENERATEDKEYSREQUESTED=false
REQUESTEDKEYCOLUMNS=

[STMT_ENTRY][43]
SQL_TEXT=[
INSERT INTO emp(id, name) VALUES(99,?)
]
STATEMENT_TYPE=Prepared Statement
RESULTSET_TYPE=Forward Only
RESULTSET_CONCURRENCY=Read Only
AUTOGENERATEDKEYSREQUESTED=false
REQUESTEDKEYCOLUMNS=id,name
```

Configuring logging

You can configure logging using a standard Java properties file that is shipped with your JVM. See "Using the JVM for logging" for details.

See also

Using the JVM for logging on page 263

Using the JVM for logging

If you want to configure logging using the properties file that is shipped with your JVM, use a text editor to modify the properties file in your JVM. Typically, this file is named logging.properties and is located in the `JRE/lib` subdirectory of your JVM. The JRE looks for this file when it is loading.

You can also specify which properties file to use by setting the java.util.logging.config.file system property. At a command prompt, enter:

```
java -Djava.util.logging.config.file=properties_file
```

where:

[40] A string that identifies the file as a statement pool export file.

[41] The version of the export file.

[42] The first statement pool entry. Each statement pool entry lists the SQL text, statement type, result set type, result set concurrency type, and generated keys information.

[43] The next statement pool entry.

`properties_file`

is the name of the properties file you want to load.

6

SQL escape sequences for JDBC

Language features, such as outer joins and scalar function calls, are commonly implemented by database systems. The syntax for these features is often database-specific, even when a standard syntax has been defined. JDBC defines escape sequences that contain the standard syntax for the following language features:

- Date, time, and timestamp literals
- Scalar functions such as numeric, string, and data type conversion functions
- Outer joins
- Escape characters for wildcards used in LIKE clauses

Note: The Progress DataDirect MongoDB for JDBC driver also supports the custom function escape CAST_TO_NATIVE.

The escape sequence used by JDBC is:

```
{extension}
```

The escape sequence is recognized and parsed by the drivers, which replaces the escape sequences with data store-specific grammar.

For details, see the following topics:

- Date, time, and timestamp escape sequences
- Scalar functions
- Outer join escape sequences
- LIKE escape character sequence for wildcards

- Procedure call escape sequences

Date, time, and timestamp escape sequences

The escape sequence for date, time, and timestamp literals is:

```
{literal-type 'value'}
```

where:

literal-type

is one of the following:

literal-type	Description	Value Format
d	Date	*yyyy-mm-dd*
t	Time	*hh:mm:ss []*
ts	Timestamp	*yyyy-mm-dd hh:mm:ss[.f...]*

Example:

```
UPDATE Orders SET OpenDate={d '1995-01-15'} WHERE OrderID=1023
```

Scalar functions

You can use scalar functions in SQL statements with the following syntax:

```
{fn scalar-function}
```

where:

scalar-function

is a scalar function supported by the driver as indicated in the following table.

Example:

```
SELECT id, name FROM emp WHERE name LIKE {fn UCASE('Smith')}
```

Note: See Azure Synapse Analytics and Analytics Platform System on page 77 for information on the scalar functions supported in Azure Synapse Analytics and Analytics Platform System environments.

Table 19: Supported scalar functions

String functions	Numeric functions	Timedate functions	System functions
ASCII	ABS	DAYNAME	DATABASE

String functions	Numeric functions	Timedate functions	System functions
CHAR	ACOS	DAYOFMONTH	IFNULL
CONCAT	ASIN	DAYOFWEEK	USER
DIFFERENCE	ATAN	DAYOFYEAR	
INSERT	ATAN2	EXTRACT	
LCASE	CEILING	HOUR	
LEFT	COS	MINUTE	
LENGTH	COT	MONTH	
LOCATE	DEGREES	MONTHNAME	
LTRIM	EXP	NOW	
REPEAT	FLOOR	QUARTER	
REPLACE	LOG	SECOND	
RIGHT	LOG10	TIMESTAMPADD	
RTRIM	MOD	TIMESTAMPDIFF	
SOUNDEX	PI	WEEK	
SPACE	POWER	YEAR	
SUBSTRING	RADIANS		
UCASE	RAND		
	ROUND		
	SIGN		
	SIN		
	SQRT		
	TAN		
	TRUNCATE		

Outer join escape sequences

JDBC supports the SQL-92 left, right, and full outer join syntax. The escape sequence for outer joins is:

`{oj` *`outer-join`*`}`

where:

`outer-join`

is *`table-reference`* `{LEFT | RIGHT | FULL} OUTER JOIN {`*`table-reference`* `| outer-join} ON` *`search-condition`* *table-reference*

is a database table name.

search-condition

is the join condition you want to use for the tables.

Example:

```
SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status
   FROM {oj Customers LEFT OUTER JOIN
      Orders ON Customers.CustID=Orders.CustID}
   WHERE Orders.Status='OPEN'
```

The driver supports the following outer join escape sequences:

- Left outer joins
- Right outer joins
- Full outer joins
- Nested outer joins

LIKE escape character sequence for wildcards

You can specify the character to be used to escape wildcard characters (% and _, for example) in LIKE clauses.

Note: Escape characters are not supported for Azure Synapse Analytics and Analytics Platform System.

The escape sequence for escape characters is:

`{escape '`*`escape-character`*`'}`

where:

escape-character

is the character used to escape the wildcard character.

For example, the following SQL statement specifies that an asterisk (*) be used as the escape character in the LIKE clause for the wildcard character %:

```
SELECT col1 FROM table1 WHERE col1 LIKE '*%%' {escape '*'}
```

Procedure call escape sequences

A procedure is an executable object stored in the data store. Generally, it is one or more SQL statements that have been precompiled. The escape sequence for calling a procedure is:

`{[?=]call `*`procedure-name`*`[(`*`parameter`*`[,`*`parameter`*`]...)]}`

where:

procedure-name

specifies the name of a stored procedure.

parameter

specifies a stored procedure parameter.

7

JDBC support

Progress DataDirect *for* JDBC drivers are compatible with JDBC 2.0, 3.0, 4.0, 4.1, and 4.2. The following topics describe support for JDBC interfaces and methods across the JDBC driver product line. Support for JDBC interfaces and methods depends, in part, on which driver you are using.

For details, see the following topics:

- Array
- Blob
- CallableStatement
- Clob
- Connection
- ConnectionEventListener
- ConnectionPoolDataSource
- DatabaseMetaData
- DataSource
- Driver
- ParameterMetaData
- PooledConnection
- PreparedStatement
- Ref

- ResultSet
- ResultSetMetaData
- RowSet
- SavePoint
- Statement
- StatementEventListener
- Struct
- XAConnection
- XADataSource
- XAResource

Array

Array Methods	Version Introduced	Supported	Comments
void free()	4.0	Yes	
Object getArray()	2.0 Core	Yes	
Object getArray(map)	2.0 Core	Yes	The drivers ignore the map argument.
Object getArray(long, int)	2.0 Core	Yes	
Object getArray(long, int, map)	2.0 Core	Yes	The drivers ignore the map argument.
int getBaseType()	2.0 Core	Yes	
String getBaseTypeName()	2.0 Core	Yes	
ResultSet getResultSet()	2.0 Core	Yes	
ResultSet getResultSet(map)	2.0 Core	Yes	The drivers ignore the map argument.
ResultSet getResultSet(long, int)	2.0 Core	Yes	
ResultSet getResultSet(long, int, map)	2.0 Core	Yes	The drivers ignore the map argument.

Blob

Blob Methods	Version Introduced	Supported	Comments
void free()	4.0	Yes	
InputStream getBinaryStream()	2.0 Core	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
byte[] getBytes(long, int)	2.0 Core	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
long length()	2.0 Core	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
long position(Blob, long)	2.0 Core	Yes	The Informix driver requires that the pattern parameter (which specifies the Blob object designating the BLOB value for which to search) be less than or equal to a maximum value of 4096 bytes. All other drivers support using data types that map to the JDBC LONGVARBINARY data type.
long position(byte[], long)	2.0 Core	Yes	The Informix driver requires that the pattern parameter (which specifies the byte array for which to search) be less than or equal to a maximum value of 4096 bytes. All other drivers support using data types that map to the JDBC LONGVARBINARY data type.
OutputStream setBinaryStream(long)	3.0	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
int setBytes(long, byte[])	3.0	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
int setBytes(long, byte[], int, int)	3.0	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.
void truncate(long)	3.0	Yes	The drivers support using data types that map to the JDBC LONGVARBINARY data type.

CallableStatement

CallableStatement Methods	Version Introduced	Supported	Comments
Array getArray(int)	2.0 Core	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters. The Progress OpenEdge driver throws an "unsupported method" exception.
Array getArray(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw an "unsupported method" exception.
Reader getCharacterStream(int)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
Reader getCharacterStream(String)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
BigDecimal getBigDecimal(int)	2.0 Core	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
BigDecimal getBigDecimal(int, int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
BigDecimal getBigDecimal(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
Blob getBlob(int)	2.0 Core	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers support using data types that map to the JDBC LONGVARBINARY data type.
Blob getBlob(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
boolean getBoolean(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
boolean getBoolean(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
byte getByte(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
byte getByte(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
byte [] getBytes(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
byte [] getBytes(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
Clob getClob(int)	2.0 Core	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers support using data types that map to the JDBC LONGVARBINARY data type.
Clob getClob(String)	3.0	Yes	Supported for the SQL Server driver only using with data types that map to the JDBC LONGVARCHAR data type. All other drivers throw "unsupported method" exception.
Date getDate(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
Date getDate(int, Calendar)	2.0 Core	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
Date getDate(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Date getDate(String, Calendar)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
double getDouble(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
double getDouble(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
float getFloat(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
float getFloat(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
int getInt(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
int getInt(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
long getLong(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
long getLong(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Reader getNCharacterStream(int)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
Reader getNCharacterStream(String)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
NClob getNClob(int)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
NClob getNClob(String)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
String getNString(int)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw "unsupported method" exception.
String getNString(String)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
Object getObject(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
Object getObject(int, Map)	2.0 Core	Yes	The drivers ignore the Map argument.
Object getObject(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Object getObject(String, Map)	3.0	Yes	Supported for the SQL Server driver only. The SQL Server driver ignores the Map argument. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
Ref getRef(int)	2.0 Core	No	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers throw "unsupported method" exception.
Ref getRef(String)	3.0	No	The drivers throw "unsupported method" exception.
short getShort(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
short getShort(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
SQLXML getSQLXML(int)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
SQLXML getSQLXML(String)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
String getString(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
String getString(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
Time getTime(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
Time getTime(int, Calendar)	2.0 Core	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
Time getTime(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Time getTime(String, Calendar)	3.0	Yes	Supported for SQL Server driver only. All other drivers throw "unsupported method" exception.
Timestamp getTimestamp(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
Timestamp getTimestamp(int, Calendar)	2.0 Core	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
Timestamp getTimestamp(String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
Timestamp getTimestamp(String, Calendar)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
URL getURL(int)	3.0	No	The drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
URL getURL(String)	3.0	No	The drivers throw "unsupported method" exception.
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
void registerOutParameter(int, int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
void registerOutParameter(int, int, int)	1.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters.
void registerOutParameter(int, int, String)	2.0 Core	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters. The Oracle driver supports the String argument. For all other drivers, the String argument is ignored.
void registerOutParameter(String, int)	3.0	Yes	Supported for the SQL Server driver only. The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
void registerOutParameter(String, int, int)	3.0	Yes	Supported for the SQL Server driver only. The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers throw "unsupported method" exception.
void registerOutParameter(String, int, String)	3.0	Yes	Supported for the SQL Server driver only. The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "invalid parameter bindings" exception when your application calls output parameters. All other drivers throw "unsupported method" exception. String/typename ignored.
void setArray(int, Array)	2.0 Core	Yes	Supported for the Oracle driver only. All other drivers throw "unsupported method" exception.
void setAsciiStream(String, InputStream)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setAsciiStream(String, InputStream, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setAsciiStream(String, InputStream, long)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBigDecimal(String, BigDecimal)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBinaryStream(String, InputStream)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
void setBinaryStream(String, InputStream, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBinaryStream(String, InputStream, long)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBlob(String, Blob)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBlob(String, InputStream)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBlob(String, InputStream, long)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBoolean(String, boolean)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setByte(String, byte)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setBytes(String, byte [])	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setCharacterStream(String, Reader, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setCharacterStream(String, InputStream, long)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setClob(String, Clob)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
void setClob(String, Reader)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setClob(String, Reader, long)	4.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setDate(String, Date)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setDate(String, Date, Calendar)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setDouble(String, double)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setFloat(String, float)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setInt(String, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setLong(String, long)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setNCharacterStream(String, Reader, long)	4.0	Yes	
void setNClob(String, NClob)	4.0	Yes	
void setNClob(String, Reader)	4.0	Yes	
void setNClob(String, Reader, long)	4.0	Yes	
void setNString(String, String)	4.0	Yes	
void setNull(int, int, String)	2.0 Core	Yes	

CallableStatement Methods	Version Introduced	Supported	Comments
void setNull(String, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setNull(String, int, String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setObject(String, Object)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setObject(String, Object, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setObject(String, Object, int, int)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setShort(String, short)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setSQLXML(String, SQLXML)	4.0	Yes	The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
void setString(String, String)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setTime(String, Time)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setTime(String, Time, Calendar)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

CallableStatement Methods	Version Introduced	Supported	Comments
void setTimestamp(String, Timestamp)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
void setTimestamp(String, Timestamp, Calendar)	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.
<T> T unwrap(Class<T> iface)	4.0	Yes	
void setURL(String, URL)	3.0	No	The drivers throw "unsupported method" exception.
boolean wasNull()	1.0	Yes	

Clob

Clob Methods	Version Introduced	Supported	Comments
void free()	4.0	Yes	
InputStream getAsciiStream()	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
Reader getCharacterStream()	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
Reader getCharacterStream(long, long)	4.0	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
String getSubString(long, int)	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
long length()	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.

Clob Methods	Version Introduced	Supported	Comments
long position(Clob, long)	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type. The Informix driver requires that the searchStr parameter be less than or equal to a maximum value of 4096 bytes.
long position(String, long)	2.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type. The Informix driver requires that the searchStr parameter be less than or equal to a maximum value of 4096 bytes.
OutputStream setAsciiStream(long)	3.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
Writer setCharacterStream(long)	3.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
int setString(long, String)	3.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
int setString(long, String, int, int)	3.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.
void truncate(long)	3.0 Core	Yes	All drivers support using with data types that map to the JDBC LONGVARCHAR data type.

Connection

Connection Methods	Version Introduced	Supported	Comments
void clearWarnings()	1.0	Yes	
void close()	1.0	Yes	When a connection is closed while a transaction is still active, that transaction is rolled back.
void commit()	1.0	Yes	

Connection Methods	Version Introduced	Supported	Comments
Blob createBlob()	4.0	Yes	
Clob createClob()	4.0	Yes	
NClob createNClob()	4.0	Yes	
createArrayOf(String, Object[])	4.0	Yes	
createStruct(String, Object[])	4.0	Yes	Only the Oracle driver supports this method.
SQLXML createSQLXML()	4.0	Yes	
Statement createStatement()	1.0	Yes	
Statement createStatement(int, int)	2.0 Core	Yes	For the DB2 driver, ResultSet.TYPE_SCROLL_SENSITIVE is downgraded to TYPE_SCROLL_INSENSITIVE. For the Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, be aware that scroll-sensitive result sets are expensive from both a Web service call and a performance perspective. The drivers expend a network round trip for each row that is fetched.
Statement createStatement(int, int, int)	3.0	No	With the exception of the DB2 driver, the specified holdability must match the database default holdability. Otherwise, an "unsupported method" exception is thrown. For the DB2 driver, the method can be called regardless of whether the specified holdability matches the database default holdability.
Struct createStruct(String, Object[])	1.0	Yes	Supported for the Oracle driver only. All other drivers throw "unsupported method" exception.
boolean getAutoCommit()	1.0	Yes	

Connection Methods	Version Introduced	Supported	Comments
String getCatalog()	1.0	Yes	The Autonomous REST Connector and the drivers for the listed database systems return an empty string because they do not have the concept of a catalog: Amazon Redshift, Apache Cassandra, Apache Hive, Apache Spark SQL, Greenplum, Jira, Impala, MongoDB, Oracle, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, PostgreSQL, and Salesforce.
String getClientInfo()	4.0	Yes	The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce do not support storing or retrieving client information.
String getClientInfo(String)	4.0	Yes	The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce do not support storing or retrieving client information.
int getHoldability()	3.0	Yes	
DatabaseMetaData getMetaData()	1.0	Yes	
int getTransactionIsolation()	1.0	Yes	
Map getTypeMap()	2.0 Core	Yes	Always returns empty java.util.HashMap.
SQLWarning getWarnings()	1.0	Yes	
boolean isClosed()	1.0	Yes	
boolean isReadOnly()	1.0	Yes	
boolean isValid()	4.0	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
String nativeSQL(String)	1.0	Yes	Always returns the same String that was passed in from the application.
CallableStatement prepareCall(String)	1.0	Yes	

Connection Methods	Version Introduced	Supported	Comments
CallableStatement prepareCall(String, int, int)	2.0 Core	Yes	For the Autonomous REST Connector and the drivers for Apache Cassandra, DB2, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce ResultSet.TYPE_SCROLL_ SENSITIVE is downgraded to TYPE_SCROLL_INSENSITIVE.
CallableStatement prepareCall(String, int, int, int)	3.0	Yes	The DB2 driver allows this method whether or not the specified holdability is the same as the default holdability. The other drivers throw the exception "Changing the default holdability is not supported" when the specified holdability does not match the default holdability.
PreparedStatement prepareStatement (String)	1.0	Yes	
PreparedStatement prepareStatement (String, int)	3.0	Yes	
PreparedStatement prepareStatement (String, int, int)	2.0 Core	Yes	For the DB2 driver, ResultSet.TYPE_SCROLL_ SENSITIVE is downgraded to TYPE_SCROLL_INSENSITIVE. For the Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, be aware that scroll-sensitive result sets are expensive from both a Web service call and a performance perspective. The drivers expend a network round trip for each row that is fetched.
PreparedStatement prepareStatement (String, int, int, int)	3.0	No	All drivers throw "unsupported method" exception.
PreparedStatement prepareStatement (String, int[])	3.0	Yes	Supported for the Oracle and SQL Server drivers. All other drivers throw "unsupported method" exception.
PreparedStatement prepareStatement (String, String [])	3.0	Yes	Supported for the SQL Server driver only. All other drivers throw "unsupported method" exception.

Connection Methods	Version Introduced	Supported	Comments
void releaseSavepoint(Savepoint)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
void rollback()	1.0	Yes	
void rollback(Savepoint)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* for i. The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
void setAutoCommit(boolean)	1.0	Yes	The Autonomous REST Connector and the drivers for Apache Cassandra, Apache Hive, Apache Spark SQL, Impala, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw "transactions not supported" exception if set to `false`.
void setCatalog(String)	1.0	Yes	The Autonomous REST Connector and the drivers for the listed database systems ignore any value set by the String argument.The corresponding drivers return an empty string because they do not have the concept of a catalog: Amazon Redshift, Apache Cassandra, Apache Hive, Apache Spark SQL, Greenplum, Impala, Jira, MongoDB, Oracle, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, PostgreSQL, and Salesforce.
String setClientInfo(Properties)	4.0	Yes	The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce do not support storing or retrieving client information.

Connection Methods	Version Introduced	Supported	Comments
String setClientInfo(String, String)	4.0	Yes	The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce do not support storing or retrieving client information.
void setHoldability(int)	3.0	Yes	The DB2 driver supports the Holdability parameter value. For other drivers, the Holdability parameter value is ignored.
void setReadOnly(boolean)	1.0	Yes	
Savepoint setSavepoint()	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. In addition, the DB2 driver only supports multiple nested savepoints for DB2 V8.2 and higher for Linux/UNIX/Windows. The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
Savepoint setSavepoint(String)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. In addition, the DB2 driver only supports multiple nested savepoints for DB2 V8.2 and higher for Linux/UNIX/Windows. The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce throw an "unsupported method" exception.
void setTransactionIsolation(int)	1.0	Yes	The Autonomous REST Connector and the drivers for Apache Cassandra, Apache Hive, Apache Spark SQL, Impala, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce ignore any specified transaction isolation level.
void setTypeMap(Map)	2.0 Core	Yes	The drivers ignore this connection method.
<T> T unwrap(Class<T> iface)	4.0	Yes	

ConnectionEventListener

ConnectionEventListener Methods	Version Introduced	Supported	Comments
void connectionClosed(event)	3.0	Yes	
void connectionErrorOccurred(event)	3.0	Yes	

ConnectionPoolDataSource

ConnectionPoolDataSource Methods	Version Introduced	Supported	Comments
int getLoginTimeout()	2.0 Optional	Yes	
PrintWriter getLogWriter()	2.0 Optional	Yes	
PooledConnection getPooledConnection()	2.0 Optional	Yes	
PooledConnection getPooledConnection (String, String)	2.0 Optional	Yes	
void setLoginTimeout(int)	2.0 Optional	Yes	
void setLogWriter(PrintWriter)	2.0 Optional	Yes	

DatabaseMetaData

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean autoCommitFailureClosesAllResultSets()	4.0	Yes	
boolean allProceduresAreCallable()	1.0	Yes	
boolean allTablesAreSelectable()	1.0	Yes	
boolean dataDefinitionCausesTransactionCommit()	1.0	Yes	
boolean dataDefinitionIgnoredInTransactions()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean deletesAreDetected(int)	2.0 Core	Yes	
boolean doesMaxRowSizeIncludeBlobs()	1.0	Yes	Not supported by the SQL Server and Sybase drivers.
getAttributes(String, String, String, String)	3.0	Yes	The Oracle driver may return results. All other drivers return an empty result set.
ResultSet getBestRowIdentifier(String, String, String, int, boolean)	1.0	Yes	
ResultSet getCatalogs()	1.0	Yes	
String getCatalogSeparator()	1.0	Yes	
String getCatalogTerm()	1.0	Yes	
String getClientInfoProperties()	4.0	Yes	The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce do not support storing or retrieving client information.
ResultSet getColumnPrivileges(String, String, String, String)	1.0	Yes	Not supported by the drivers for Apache Hive, Apache Spark SQL, Impala, Oracle Eloqua, and Oracle Sales Cloud.
ResultSet getColumns(String, String, String, String)	1.0	Yes	
Connection getConnection()	2.0 Core	Yes	
ResultSet getCrossReference(String, String, String, String, String, String)	1.0	Yes	
ResultSet getFunctions()	4.0	Yes	The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce return an empty result set. Not supported by the drivers for Apache Hive, Apache Spark SQL, or Impala.

DatabaseMetaData Methods	Version Introduced	Supported	Comments
ResultSet getFunctionColumns()	4.0	Yes	The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce return an empty result set. Not supported by the drivers for Apache Hive, Apache Spark SQL, or Impala.
int getDatabaseMajorVersion()	3.0	Yes	
int getDatabaseMinorVersion()	3.0	Yes	
String getDatabaseProductName()	1.0	Yes	
String getDatabaseProductVersion()	1.0	Yes	
int getDefaultTransactionIsolation()	1.0	Yes	
int getDriverMajorVersion()	1.0	Yes	
int getDriverMinorVersion()	1.0	Yes	
String getDriverName()	1.0	Yes	
String getDriverVersion()	1.0	Yes	
ResultSet getExportedKeys(String, String, String)	1.0	Yes	
String getExtraNameCharacters()	1.0	Yes	
String getIdentifierQuoteString()	1.0	Yes	
ResultSet getImportedKeys(String, String, String)	1.0	Yes	
ResultSet getIndexInfo(String, String, String, boolean, boolean)	1.0	Yes	
int getJDBCMajorVersion()	3.0	Yes	
int getJDBCMinorVersion()	3.0	Yes	
int getMaxBinaryLiteralLength()	1.0	Yes	
int getMaxCatalogNameLength()	1.0	Yes	
int getMaxCharLiteralLength()	1.0	Yes	
int getMaxColumnNameLength()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
int getMaxColumnsInGroupBy()	1.0	Yes	
int getMaxColumnsInIndex()	1.0	Yes	
int getMaxColumnsInOrderBy()	1.0	Yes	
int getMaxColumnsInSelect()	1.0	Yes	
int getMaxColumnsInTable()	1.0	Yes	
int getMaxConnections()	1.0	Yes	
int getMaxCursorNameLength()	1.0	Yes	
int getMaxIndexLength()	1.0	Yes	
int getMaxProcedureNameLength()	1.0	Yes	
int getMaxRowSize()	1.0	Yes	
int getMaxSchemaNameLength()	1.0	Yes	
int getMaxStatementLength()	1.0	Yes	
int getMaxStatements()	1.0	Yes	
int getMaxTableNameLength()	1.0	Yes	
int getMaxTablesInSelect()	1.0	Yes	
int getMaxUserNameLength()	1.0	Yes	
String getNumericFunctions()	1.0	Yes	
ResultSet getPrimaryKeys(String, String, String)	1.0	Yes	
ResultSet getProcedureColumns(String, String, String, String)	1.0	Yes	For the Autonomous REST Connector and the drivers for Jira, Oracle Service Cloud, and Salesforce, SchemaName and ProcedureName must be explicit values; they cannot be patterns. The drivers for Apache Cassandra and MongoDB return an empty result set. Not supported for the drivers for Apache Hive, Apache Spark SQL, or Impala.

DatabaseMetaData Methods	Version Introduced	Supported	Comments
ResultSet getProcedures(String, String, String)	1.0	Yes	The drivers for Apache Cassandra, MongoDB, Oracle Eloqua, and Oracle Sales Cloud return an empty result set. Not supported for the drivers for Apache Hive, Apache Spark SQL, or Impala.
String getProcedureTerm()	1.0	Yes	
int getResultSetHoldability()	3.0	Yes	
ResultSet getSchemas()	1.0	Yes	
ResultSet getSchemas(catalog, pattern)	4.0	Yes	
String getSchemaTerm()	1.0	Yes	
String getSearchStringEscape()	1.0	Yes	
String getSQLKeywords()	1.0	Yes	
int getSQLStateType()	3.0	Yes	
String getStringFunctions()	1.0	Yes	
ResultSet getSuperTables(String, String, String)	3.0	Yes	Returns an empty result set.
ResultSet getSuperTypes(String, String, String)	3.0	Yes	Returns an empty result set.
String getSystemFunctions()	1.0	Yes	
ResultSet getTablePrivileges(String, String, String)	1.0	Yes	Not supported for the drivers for Apache Hive, Apache Spark SQL, Impala, Oracle Eloqua, and Oracle Sales Cloud.
ResultSet getTables(String, String, String, String [])	1.0	Yes	
ResultSet getTableTypes()	1.0	Yes	
String getTimeDateFunctions()	1.0	Yes	
ResultSet getTypeInfo()	1.0	Yes	
ResultSet getUDTs(String, String, String, int [])	2.0 Core	Yes	Supported for Oracle only.
String getURL()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
String getUserName()	1.0	Yes	
ResultSet getVersionColumns(String, String, String)	1.0	Yes	
boolean insertsAreDetected(int)	2.0 Core	Yes	
boolean isCatalogAtStart()	1.0	Yes	
boolean isReadOnly()	1.0	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
boolean locatorsUpdateCopy()	3.0	Yes	
boolean nullPlusNonNullIsNull()	1.0	Yes	
boolean nullsAreSortedAtEnd()	1.0	Yes	
boolean nullsAreSortedAtStart()	1.0	Yes	
boolean nullsAreSortedHigh()	1.0	Yes	
boolean nullsAreSortedLow()	1.0	Yes	
boolean othersDeletesAreVisible(int)	2.0 Core	Yes	
boolean othersInsertsAreVisible(int)	2.0 Core	Yes	
boolean othersUpdatesAreVisible(int)	2.0 Core	Yes	
boolean ownDeletesAreVisible(int)	2.0 Core	Yes	
boolean ownInsertsAreVisible(int)	2.0 Core	Yes	
boolean ownUpdatesAreVisible(int)	2.0 Core	Yes	
boolean storesLowerCaseIdentifiers()	1.0	Yes	
boolean storesLowerCaseQuotedIdentifiers()	1.0	Yes	
boolean storesMixedCaseIdentifiers()	1.0	Yes	
boolean storesMixedCaseQuotedIdentifiers()	1.0	Yes	
boolean storesUpperCaseIdentifiers()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean storesUpperCaseQuotedIdentifiers()	1.0	Yes	
boolean supportsAlterTableWithAddColumn()	1.0	Yes	
boolean supportsAlterTableWithDropColumn()	1.0	Yes	
boolean supportsANSI92EntryLevelSQL()	1.0	Yes	
boolean supportsANSI92FullSQL()	1.0	Yes	
boolean supportsANSI92IntermediateSQL()	1.0	Yes	
boolean supportsBatchUpdates()	2.0 Core	Yes	
boolean supportsCatalogsInDataManipulation()	1.0	Yes	
boolean supportsCatalogsInIndexDefinitions()	1.0	Yes	
boolean supportsCatalogsInPrivilegeDefinitions()	1.0	Yes	
boolean supportsCatalogsInProcedureCalls()	1.0	Yes	
boolean supportsCatalogsInTableDefinitions()	1.0	Yes	
boolean supportsColumnAliasing()	1.0	Yes	
boolean supportsConvert()	1.0	Yes	
boolean supportsConvert(int, int)	1.0	Yes	
boolean supportsCoreSQLGrammar()	1.0	Yes	
boolean supportsCorrelatedSubqueries()	1.0	Yes	
boolean supportsDataDefinitionAndDataManipulationTransactions()	1.0	Yes	
boolean supportsDataManipulationTransactionsOnly()	1.0	Yes	
boolean supportsDifferentTableCorrelationNames()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean supportsExpressionsInOrderBy()	1.0	Yes	
boolean supportsExtendedSQLGrammar()	1.0	Yes	
boolean supportsFullOuterJoins()	1.0	Yes	
boolean supportsGetGeneratedKeys()	3.0	Yes	
boolean supportsGroupBy()	1.0	Yes	
boolean supportsGroupByBeyondSelect()	1.0	Yes	
boolean supportsGroupByUnrelated()	1.0	Yes	
boolean supportsIntegrityEnhancementFacility()	1.0	Yes	
boolean supportsLikeEscapeClause()	1.0	Yes	
boolean supportsLimitedOuterJoins()	1.0	Yes	
boolean supportsMinimumSQLGrammar()	1.0	Yes	
boolean supportsMixedCaseIdentifiers()	1.0	Yes	
boolean supportsMixedCaseQuotedIdentifiers()	1.0	Yes	
boolean supportsMultipleOpenResults()	3.0	Yes	
boolean supportsMultipleResultSets()	1.0	Yes	
boolean supportsMultipleTransactions()	1.0	Yes	
boolean supportsNamedParameters()	3.0	Yes	
boolean supportsNonNullableColumns()	1.0	Yes	
boolean supportsOpenCursorsAcrossCommit()	1.0	Yes	
boolean supportsOpenCursorsAcrossRollback()	1.0	Yes	
boolean supportsOpenStatementsAcrossCommit()	1.0	Yes	
boolean supportsOpenStatementsAcrossRollback()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean supportsOrderByUnrelated()	1.0	Yes	
boolean supportsOuterJoins()	1.0	Yes	
boolean supportsPositionedDelete()	1.0	Yes	
boolean supportsPositionedUpdate()	1.0	Yes	
boolean supportsResultSetConcurrency(int, int)	2.0 Core	Yes	
boolean supportsResultSetHoldability(int)	3.0	Yes	
boolean supportsResultSetType(int)	2.0 Core	Yes	
boolean supportsSavePoints()	3.0	Yes	
boolean supportsSchemasInDataManipulation()	1.0	Yes	
boolean supportsSchemasInIndexDefinitions()	1.0	Yes	
boolean supportsSchemasInPrivilegeDefinitions()	1.0	Yes	
boolean supportsSchemasInProcedureCalls()	1.0	Yes	
boolean supportsSchemasInTableDefinitions()	1.0	Yes	
boolean supportsSelectForUpdate()	1.0	Yes	
boolean supportsStoredFunctionsUsingCallSyntax()	4.0	Yes	
boolean supportsStoredProcedures()	1.0	Yes	
boolean supportsSubqueriesInComparisons()	1.0	Yes	
boolean supportsSubqueriesInExists()	1.0	Yes	
boolean supportsSubqueriesInIns()	1.0	Yes	
boolean supportsSubqueriesInQuantifieds()	1.0	Yes	
boolean supportsTableCorrelationNames()	1.0	Yes	

DatabaseMetaData Methods	Version Introduced	Supported	Comments
boolean supportsTransactionIsolationLevel(int)	1.0	Yes	
boolean supportsTransactions()	1.0	Yes	
boolean supportsUnion()	1.0	Yes	
boolean supportsUnionAll()	1.0	Yes	
<T> T unwrap(Class<T> iface)	4.0	Yes	
boolean updatesAreDetected(int)	2.0 Core	Yes	
boolean usesLocalFilePerTable()	1.0	Yes	
boolean usesLocalFiles()	1.0	Yes	

DataSource

The DataSource interface implements the javax.naming.Referenceable and java.io.Serializable interfaces.

DataSource Methods	Version Introduced	Supported	Comments
Connection getConnection()	2.0 Optional	Yes	
Connection getConnection(String, String)	2.0 Optional	Yes	
int getLoginTimeout()	2.0 Optional	Yes	
PrintWriter getLogWriter()	2.0 Optional	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
void setLoginTimeout(int)	2.0 Optional	Yes	
void setLogWriter(PrintWriter)	2.0 Optional	Yes	Enables DataDirect Spy, which traces JDBC information into the specified PrintWriter.
<T> T unwrap(Class<T> iface)	4.0	Yes	

Driver

Driver Methods	Version Introduced	Supported	Comments
boolean acceptsURL(String)	1.0	Yes	
Connection connect(String, Properties)	1.0	Yes	
int getMajorVersion()	1.0	Yes	
int getMinorVersion()	1.0	Yes	
DriverPropertyInfo [] getPropertyInfo(String, Properties)	1.0	Yes	

ParameterMetaData

ParameterMetaData Methods	Version Introduced	Supported	Comments
String getParameterClassName(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
int getParameterCount()	3.0	Yes	
int getParameterMode(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
int getParameterType(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
String getParameterTypeName(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.x and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.

ParameterMetaData Methods	Version Introduced	Supported	Comments
int getPrecision(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
int getScale(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
int isNullable(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
boolean isSigned(int)	3.0	Yes	The DB2 driver supports parameter metadata for stored procedures for DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
boolean jdbcCompliant()	1.0	Yes	
<T> T unwrap(Class<T> iface)	4.0	Yes	

PooledConnection

PooledConnection Methods	Version Introduced	Supported	Comments
void addConnectionEventListener(listener)	2.0 Optional	Yes	
void addStatementEventListener(listener)	4.0	Yes	
void close()	2.0 Optional	Yes	

PooledConnection Methods	Version Introduced	Supported	Comments
Connection getConnection()	2.0 Optional	Yes	A pooled connection object can have only one Connection object open (the one most recently created). The purpose of allowing the server (PoolManager implementation) to invoke this a second time is to give an application server a way to take a connection away from an application and give it to another user (a rare occurrence). The drivers do not support the "reclaiming" of connections and will throw an exception.
void removeConnectionEventListener(listener)	2.0 Optional	Yes	
void removeStatementEventListener(listener)	4.0	Yes	

PreparedStatement

PreparedStatement Methods	Version Introduced	Supported	Comments
void addBatch()	2.0 Core	Yes	
void clearParameters()	1.0	Yes	
boolean execute()	1.0	Yes	
ResultSet executeQuery()	1.0	Yes	
int executeUpdate()	1.0	Yes	
ResultSetMetaData getMetaData()	2.0 Core	Yes	
ParameterMetaData getParameterMetaData()	3.0	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
void setArray(int, Array)	2.0 Core	Yes	Supported for the Oracle driver only. All other drivers throw an "unsupported method" exception.
void setAsciiStream(int, InputStream)	4.0	Yes	
void setAsciiStream(int, InputStream, int)	1.0	Yes	

PreparedStatement Methods	Version Introduced	Supported	Comments
void setAsciiStream(int, InputStream, long)	4.0	Yes	
void setBigDecimal(int, BigDecimal)	1.0	Yes	
void setBinaryStream(int, InputStream)	4.0	Yes	When used with Blobs, the DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
void setBinaryStream(int, InputStream, int)	1.0	Yes	When used with Blobs, the DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
void setBinaryStream(int, InputStream, long)	4.0	Yes	When used with Blobs, the DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.
void setBlob(int, Blob)	2.0 Core	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setBlob(int, InputStream)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setBlob(int, InputStream, long)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setBoolean(int, boolean)	1.0	Yes	
void setByte(int, byte)	1.0	Yes	
void setBytes(int, byte [])	1.0	Yes	When used with Blobs, the DB2 driver only supports with DB2 V8.x and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i.

PreparedStatement Methods	Version Introduced	Supported	Comments
void setCharacterStream(int, Reader)	4.0	Yes	
void setCharacterStream(int, Reader, int)	2.0 Core	Yes	
void setCharacterStream(int, Reader, long)	4.0	Yes	
void setClob(int, Clob)	2.0 Core	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setClob(int, Reader)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setClob(int, Reader, long)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void setDate(int, Date)	1.0	Yes	
void setDate(int, Date, Calendar)	2.0 Core	Yes	
void setDouble(int, double)	1.0	Yes	
void setFloat(int, float)	1.0	Yes	
void setInt(int, int)	1.0	Yes	
void setLong(int, long)	1.0	Yes	
void setNCharacterStream(int, Reader)	4.0	Yes	For the Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void setNCharacterStream(int, Reader, long)	4.0	Yes	For the Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void setNClob(int, NClob)	4.0	Yes	For the Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.

PreparedStatement Methods	Version Introduced	Supported	Comments
void setNClob(int, Reader)	4.0	Yes	For the Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void setNClob(int, Reader, long)	4.0	Yes	For the Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void setNull(int, int)	1.0	Yes	
void setNull(int, int, String)	2.0 Core	Yes	
void setNString(int, String)	4.0	Yes	
void setObject(int, Object)	1.0	Yes	
void setObject(int, Object, int)	1.0	Yes	
void setObject(int, Object, int, int)	1.0	Yes	

PreparedStatement Methods	Version Introduced	Supported	Comments
void setQueryTimeout(int)	1.0	Yes	The DB2 driver supports setting a timeout value, in seconds, for a statement with DB2 V8.*x* and higher for Linux/UNIX/Windows and DB2 V8.1 and higher for z/OS. If the execution of the statement exceeds the timeout value, the statement is timed out by the database server, and the driver throws an exception indicating that the statement was timed out. The DB2 driver throws an "unsupported method" exception with other DB2 versions. The Informix driver throws an "unsupported method" exception. The Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce ignore any value set using this method. Use the WSTimeout connection property to set a timeout value. The drivers for Apache Cassandra and MongoDB ignore any value set using this method. All other drivers support setting a timeout value, in seconds, for a statement. If the execution of the statement exceeds the timeout value, the statement is timed out by the database server, and the driver throws an exception indicating that the statement was timed out.
void setRef(int, Ref)	2.0 Core	No	All drivers throw "unsupported method" exception.
void setShort(int, short)	1.0	Yes	
void setSQLXML(int, SQLXML)	4.0	Yes	
void setString(int, String)	1.0	Yes	
void setTime(int, Time)	1.0	Yes	
void setTime(int, Time, Calendar)	2.0 Core	Yes	
void setTimestamp(int, Timestamp)	1.0	Yes	
void setTimestamp(int, Timestamp, Calendar)	2.0 Core	Yes	

PreparedStatement Methods	Version Introduced	Supported	Comments
void setUnicodeStream(int, InputStream, int)	1.0	No	This method was deprecated in JDBC 2.0. All drivers throw "unsupported method" exception.
<T> T unwrap(Class<T> iface)	4.0	Yes	
void setURL(int, URL)	3.0	No	All drivers throw "unsupported method" exception.

Ref

Ref MethodsRef interface	Version Introduced	Supported	Comments
(all)	2.0 Core	No	

ResultSet

ResultSet Methods	Version Introduced	Supported	Comments
boolean absolute(int)	2.0 Core	Yes	
void afterLast()	2.0 Core	Yes	
void beforeFirst()	2.0 Core	Yes	
void cancelRowUpdates()	2.0 Core	Yes	
void clearWarnings()	1.0	Yes	
void close()	1.0	Yes	
void deleteRow()	2.0 Core	Yes	
int findColumn(String)	1.0	Yes	
boolean first()	2.0 Core	Yes	
Array getArray(int)	2.0 Core	Yes	
Array getArray(String)	2.0 Core	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
InputStream getAsciiStream(int)	1.0	Yes	
InputStream getAsciiStream(String)	1.0	Yes	
BigDecimal getBigDecimal(int)	2.0 Core	Yes	
BigDecimal getBigDecimal(int, int)	1.0	Yes	
BigDecimal getBigDecimal(String)	2.0 Core	Yes	
BigDecimal getBigDecimal(String, int)	1.0	Yes	
InputStream getBinaryStream(int)	1.0	Yes	The DB2 driver supports for all DB2 versions when retrieving BINARY, VARBINARY, and LONGVARBINARY data. The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i when retrieving Blob data.
InputStream getBinaryStream(String)	1.0	Yes	The DB2 driver supports for all DB2 versions when retrieving BINARY, VARBINARY, and LONGVARBINARY data. The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i when retrieving Blob data.
Blob getBlob(int)	2.0 Core	Yes	The DB2 driver only supports with DB2 V8.x and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
Blob getBlob(String)	2.0 Core	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
boolean getBoolean(int)	1.0	Yes	
boolean getBoolean(String)	1.0	Yes	
byte getByte(int)	1.0	Yes	
byte getByte(String)	1.0	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
byte [] getBytes(int)	1.0	Yes	The DB2 driver supports for all DB2 versions when retrieving BINARY, VARBINARY, and LONGVARBINARY data. The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i when retrieving Blob data.
byte [] getBytes(String)	1.0	Yes	The DB2 driver supports for all DB2 versions when retrieving BINARY, VARBINARY, and LONGVARBINARY data. The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i when retrieving Blob data.
Reader getCharacterStream(int)	2.0 Core	Yes	
Reader getCharacterStream(String)	2.0 Core	Yes	
Clob getClob(int)	2.0 Core	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
Clob getClob(String)	2.0 Core	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
int getConcurrency()	2.0 Core	Yes	
String getCursorName()	1.0	No	All drivers throw "unsupported method" exception.
Date getDate(int)	1.0	Yes	
Date getDate(int, Calendar)	2.0 Core	Yes	
Date getDate(String)	1.0	Yes	
Date getDate(String, Calendar)	2.0 Core	Yes	
double getDouble(int)	1.0	Yes	
double getDouble(String)	1.0	Yes	
int getFetchDirection()	2.0 Core	Yes	
int getFetchSize()	2.0 Core	Yes	
float getFloat(int)	1.0	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
float getFloat(String)	1.0	Yes	
int getHoldability()	4.0	Yes	
int getInt(int)	1.0	Yes	
int getInt(String)	1.0	Yes	
long getLong(int)	1.0	Yes	
long getLong(String)	1.0	Yes	
ResultSetMetaData getMetaData()	1.0	Yes	
Reader getNCharacterStream(int)	4.0	Yes	
Reader getNCharacterStream(String)	4.0	Yes	
NClob getNClob(int)	4.0	Yes	
NClob getNClob(String)	4.0	Yes	
String getNString(int)	4.0	Yes	
String getNString(String)	4.0	Yes	
Object getObject(int)	1.0	Yes	The DB2 driver returns a Long object when called on Bigint columns.
Object getObject(int, Map)	2.0 Core	Yes	The Oracle and Sybase drivers support the Map argument. For all other drivers, the Map argument is ignored.
Object getObject(String)	1.0	Yes	
Object getObject(String, Map)	2.0 Core	Yes	The Oracle and Sybase drivers support the Map argument. For all other drivers, the Map argument is ignored.
Ref getRef(int)	2.0 Core	No	All drivers throw "unsupported method" exception.
Ref getRef(String)	2.0 Core	No	All drivers throw "unsupported method" exception.
int getRow()	2.0 Core	Yes	
short getShort(int)	1.0	Yes	
short getShort(String)	1.0	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
SQLXML getSQLXML(int)	4.0	Yes	
SQLXML getSQLXML(String)	4.0	Yes	
Statement getStatement()	2.0 Core	Yes	
String getString(int)	1.0	Yes	
String getString(String)	1.0	Yes	
Time getTime(int)	1.0	Yes	
Time getTime(int, Calendar)	2.0 Core	Yes	
Time getTime(String)	1.0	Yes	
Time getTime(String, Calendar)	2.0 Core	Yes	
Timestamp getTimestamp(int)	1.0	Yes	
Timestamp getTimestamp(int, Calendar)	2.0 Core	Yes	
Timestamp getTimestamp(String)	1.0	Yes	
Timestamp getTimestamp(String, Calendar)	2.0 Core	Yes	
int getType()	2.0 Core	Yes	
InputStream getUnicodeStream(int)	1.0	No	This method was deprecated in JDBC 2.0. All drivers throw "unsupported method" exception.
InputStream getUnicodeStream(String)	1.0	No	This method was deprecated in JDBC 2.0. All drivers throw "unsupported method" exception.
URL getURL(int)	3.0	No	All drivers throw "unsupported method" exception.
URL getURL(String)	3.0	No	All drivers throw "unsupported method" exception.
SQLWarning getWarnings()	1.0	Yes	
void insertRow()	2.0 Core	Yes	
boolean isAfterLast()	2.0 Core	Yes	
boolean isBeforeFirst()	2.0 Core	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
boolean isClosed()	4.0	Yes	
boolean isFirst()	2.0 Core	Yes	
boolean isLast()	2.0 Core	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
boolean last()	2.0 Core	Yes	
void moveToCurrentRow()	2.0 Core	Yes	
void moveToInsertRow()	2.0 Core	Yes	
boolean next()	1.0	Yes	
boolean previous()	2.0 Core	Yes	
void refreshRow()	2.0 Core	Yes	
boolean relative(int)	2.0 Core	Yes	
boolean rowDeleted()	2.0 Core	Yes	
boolean rowInserted()	2.0 Core	Yes	
boolean rowUpdated()	2.0 Core	Yes	
void setFetchDirection(int)	2.0 Core	Yes	
void setFetchSize(int)	2.0 Core	Yes	
<T> T unwrap(Class<T> iface)	4.0	Yes	
void updateArray(int, Array)	3.0	No	All drivers throw "unsupported method" exception.
void updateArray(String, Array)	3.0	No	All drivers throw "unsupported method" exception.
void updateAsciiStream(int, InputStream, int)	2.0 Core	Yes	
void updateAsciiStream(int, InputStream, long)	4.0	Yes	
void updateAsciiStream(String, InputStream)	4.0	Yes	
void updateAsciiStream(String, InputStream, int)	2.0 Core	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
void updateAsciiStream(String, InputStream, long)	4.0	Yes	
void updateBigDecimal(int, BigDecimal)	2.0 Core	Yes	
void updateBigDecimal(String, BigDecimal)	2.0 Core	Yes	
void updateBinaryStream(int, InputStream)	4.0	Yes	
void updateBinaryStream(int, InputStream, int)	2.0 Core	Yes	
void updateBinaryStream(int, InputStream, long)	4.0	Yes	
void updateBinaryStream(String, InputStream)	4.0	Yes	
void updateBinaryStream(String, InputStream, int)	2.0 Core	Yes	
void updateBinaryStream(String, InputStream, long)	4.0	Yes	
void updateBlob(int, Blob)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateBlob(int, InputStream)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateBlob(int, InputStream, long)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.

ResultSet Methods	Version Introduced	Supported	Comments
void updateBlob(String, Blob)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateBlob(String, InputStream)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateBlob(String, InputStream, long)	4.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS (all versions), and DB2 for i. All other drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateBoolean(int, boolean)	2.0 Core	Yes	
void updateBoolean(String, boolean)	2.0 Core	Yes	
void updateByte(int, byte)	2.0 Core	Yes	
void updateByte(String, byte)	2.0 Core	Yes	
void updateBytes(int, byte [])	2.0 Core	Yes	
void updateBytes(String, byte [])	2.0 Core	Yes	
void updateCharacterStream(int, Reader)	4.0	Yes	
void updateCharacterStream(int, Reader, int)	2.0 Core	Yes	
void updateCharacterStream(int, Reader, long)	4.0	Yes	
void updateCharacterStream(String, Reader)	4.0	Yes	
void updateCharacterStream(String, Reader, int)	2.0 Core	Yes	
void updateCharacterStream(String, Reader, long)	4.0	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
void updateClob(int, Clob)	3.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateClob(int, Reader)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateClob(int, Reader, long)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateClob(String, Clob)	3.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateClob(String, Reader)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateClob(String, Reader, long)	4.0	Yes	Drivers support using with data types that map to the JDBC LONGVARBINARY data type.
void updateDate(int, Date)	2.0 Core	Yes	
void updateDate(String, Date)	2.0 Core	Yes	
void updateDouble(int, double)	2.0 Core	Yes	
void updateDouble(String, double)	2.0 Core	Yes	
void updateFloat(int, float)	2.0 Core	Yes	
void updateFloat(String, float)	2.0 Core	Yes	
void updateInt(int, int)	2.0 Core	Yes	
void updateInt(String, int)	2.0 Core	Yes	
void updateLong(int, long)	2.0 Core	Yes	
void updateLong(String, long)	2.0 Core	Yes	
void updateNCharacterStream(int, Reader)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.

ResultSet Methods	Version Introduced	Supported	Comments
void updateNCharacterStream(int, Reader, long)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void updateNCharacterStream(String, Reader)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void updateNCharacterStream(String, Reader, long)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void updateNClob(int, NClob)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void updateNClob(int, Reader)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void updateNClob(int, Reader, long)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void updateNClob(String, NClob)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void updateNClob(String, Reader)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.

ResultSet Methods	Version Introduced	Supported	Comments
void updateNClob(String, Reader, long)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void updateNString(int, String)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void updateNString(String, String)	4.0	Yes	For the Autonomous REST Connector and the drivers for Jira, MongoDB, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce, N methods are identical to their non-N counterparts.
void updateNull(int)	2.0 Core	Yes	
void updateNull(String)	2.0 Core	Yes	
void updateObject(int, Object)	2.0 Core	Yes	
void updateObject(int, Object, int)	2.0 Core	Yes	
void updateObject(String, Object)	2.0 Core	Yes	
void updateObject(String, Object, int)	2.0 Core	Yes	
void updateRef(int, Ref)	3.0	No	All drivers throw "unsupported method" exception.
void updateRef(String, Ref)	3.0	No	All drivers throw "unsupported method" exception.
void updateRow()	2.0 Core	Yes	
void updateShort(int, short)	2.0 Core	Yes	
void updateShort(String, short)	2.0 Core	Yes	
void updateSQLXML(int, SQLXML)	4.0	Yes	
void updateSQLXML(String, SQLXML)	4.0	Yes	
void updateString(int, String)	2.0 Core	Yes	
void updateString(String, String)	2.0 Core	Yes	
void updateTime(int, Time)	2.0 Core	Yes	

ResultSet Methods	Version Introduced	Supported	Comments
void updateTime(String, Time)	2.0 Core	Yes	
void updateTimestamp(int, Timestamp)	2.0 Core	Yes	
void updateTimestamp(String, Timestamp)	2.0 Core	Yes	
boolean wasNull()	1.0	Yes	

ResultSetMetaData

ResultSetMetaData Methods	Version Introduced	Supported	Comments
String getCatalogName(int)	1.0	Yes	
String getColumnClassName(int)	2.0 Core	Yes	
int getColumnCount()	1.0	Yes	
int getColumnDisplaySize(int)	1.0	Yes	
String getColumnLabel(int)	1.0	Yes	
String getColumnName(int)	1.0	Yes	
int getColumnType(int)	1.0	Yes	
String getColumnTypeName(int)	1.0	Yes	
int getPrecision(int)	1.0	Yes	
int getScale(int)	1.0	Yes	
String getSchemaName(int)	1.0	Yes	
String getTableName(int)	1.0	Yes	
boolean isAutoIncrement(int)	1.0	Yes	
boolean isCaseSensitive(int)	1.0	Yes	
boolean isCurrency(int)	1.0	Yes	
boolean isDefinitelyWritable(int)	1.0	Yes	
int isNullable(int)	1.0	Yes	

ResultSetMetaData Methods	Version Introduced	Supported	Comments
boolean isReadOnly(int)	1.0	Yes	
boolean isSearchable(int)	1.0	Yes	
boolean isSigned(int)	1.0	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
boolean isWritable(int)	1.0	Yes	
<T> T unwrap(Class<T> iface)	4.0	Yes	

RowSet

RowSet Methods	Version Introduced	Supported	Comments
(all)	2.0 Optional	No	

SavePoint

SavePoint Methods	Version Introduced	Supported	Comments
(all)	3.0	Yes	The DB2 driver only supports with DB2 V8.*x* and higher for Linux/UNIX/Windows, DB2 for z/OS ((all versions), and DB2 for i.

Statement

Statement Methods	Version Introduced	Supported	Comments
void addBatch(String)	2.0 Core	Yes	All drivers throw "invalid method call" exception for PreparedStatement and CallableStatement.
void cancel()	1.0	Yes	The DB2 driver cancels the execution of the statement with DB2 V8.*x* and higher for Linux/UNIX/Windows and DB2 V8.1 and

Statement Methods	Version Introduced	Supported	Comments
			higher for z/OS. If the statement is canceled by the database server, the driver throws an exception indicating that it was canceled. The DB2 driver throws an "unsupported method" exception with other DB2 versions. The Autonomous REST Connector and the drivers for Apache Cassandra, Impala, Informix, Jira, MongoDB, Progess OpenEdge, Oracle Service Cloud, Oracle Eloqua, Oracle Sales Cloud, and Salesforce throw an "unsupported method" exception. The Apache Hive, Apache Spark SQL[44], Greenplum, Oracle, PostgreSQL, SQL Server, Sybase, and Amazon Redshift drivers cancel the execution of the statement. If the statement is canceled by the database server, the driver throws an exception indicating that it was canceled.
void clearBatch()	2.0 Core	Yes	
void clearWarnings()	1.0	Yes	
void close()	1.0	Yes	
boolean execute(String)	1.0	Yes	All drivers throw "invalid method call" exception for PreparedStatement and CallableStatement.
boolean execute(String, int)	3.0	Yes	
boolean execute(String, int [])	3.0	Yes	Supported for the Oracle and SQL Server drivers. All other drivers throw "unsupported method" exception.
boolean execute(String, String [])	3.0	Yes	Supported for the Oracle and SQL Server drivers. All other drivers throw "unsupported method" exception.
int [] executeBatch()	2.0 Core	Yes	
ResultSet executeQuery(String)	1.0	Yes	All drivers throw "invalid method call" exception for PreparedStatement and CallableStatement.

[44] Supported only for Apache Spark SQL 2.0 and higher. For earlier versions of Apache Spark SQL, the driver throws an "unsupported method" exception.

Statement Methods	Version Introduced	Supported	Comments
int executeUpdate(String)	1.0	Yes	All drivers throw "invalid method call" exception for PreparedStatement and CallableStatement.
int executeUpdate(String, int)	3.0	Yes	
int executeUpdate(String, int [])	3.0	Yes	Supported for the Oracle and SQL Server drivers. All other drivers throw "unsupported method" exception.
int executeUpdate(String, String [])	3.0	Yes	Supported for the Oracle and SQL Server drivers. All other drivers throw "unsupported method" exception.
Connection getConnection()	2.0 Core	Yes	
int getFetchDirection()	2.0 Core	Yes	
int getFetchSize()	2.0 Core	Yes	
ResultSet getGeneratedKeys()	3.0	Yes	The DB2, SQL Server, and Sybase drivers return the last value inserted into an identity column. If an identity column does not exist in the table, the drivers return an empty result set. The Informix driver returns the last value inserted into a Serial or Serial8 column. If a Serial or Serial8 column does not exist in the table, the driver returns an empty result set. The Oracle driver returns the ROWID of the last row that was inserted. The Autonomous REST Connector and the drivers for Apache Cassandra, Jira, MongoDB, Oracle Eloqua, Oracle Service Cloud, and Salesforce return the ID of the last row that was inserted. Auto-generated keys are not supported in any of the other drivers.
int getMaxFieldSize()	1.0	Yes	
int getMaxRows()	1.0	Yes	
boolean getMoreResults()	1.0	Yes	

Statement Methods	Version Introduced	Supported	Comments
boolean getMoreResults(int)	3.0	Yes	
int getQueryTimeout()	1.0	Yes	The DB2 driver returns the timeout value, in seconds, set for the statement with DB2 V8.*x* and higher for Linux/UNIX/Windows and DB2 V8.1 and higher for z/OS. The DB2 driver returns `0` with other DB2 versions. The Apache Hive, Apache Spark SQL, Impala, Informix and Progress OpenEdge drivers return `0`. The drivers for Apache Cassandra, Greenplum, Oracle, PostgreSQL, SQL Server, Sybase, and Amazon Redshift return the timeout value, in seconds, set for the statement. The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce return an "unsupported method" exception.
ResultSet getResultSet()	1.0	Yes	
int getResultSetConcurrency()	2.0 Core	Yes	
int getResultSetHoldability()	3.0	Yes	
int getResultSetType()	2.0 Core	Yes	
int getUpdateCount()	1.0	Yes	
SQLWarning getWarnings()	1.0	Yes	
boolean isClosed()	4.0	Yes	
boolean isPoolable()	4.0	Yes	
boolean isWrapperFor(Class<?> iface)	4.0	Yes	
void setCursorName(String)	1.0	No	Throws "unsupported method" exception.
void setEscapeProcessing(boolean)	1.0	Yes	Ignored.
void setFetchDirection(int)	2.0 Core	Yes	
void setFetchSize(int)	2.0 Core	Yes	
void setMaxFieldSize(int)	1.0	Yes	

Statement Methods	Version Introduced	Supported	Comments
void setMaxRows(int)	1.0	Yes	
void setPoolable(boolean)	4.0	Yes	
void setQueryTimeout(int)	1.0	Yes	The DB2 driver supports setting a timeout value, in seconds, for a statement with DB2 V8.*x* and higher for Linux/UNIX/Windows and DB2 V8.1 and higher for z/OS. If the execution of the statement exceeds the timeout value, the statement is timed out by the database server, and the driver throws an exception indicating that the statement was timed out. The DB2 driver throws an "unsupported method" exception with other DB2 versions. The drivers for Apache Hive, Apache Spark SQL, Impala, and Informix throw an "unsupported method" exception. The drivers for Greenplum, Oracle, PostgreSQL, Progress OpenEdge, SQL Server, Sybase, and Amazon Redshift support setting a timeout value, in seconds, for a statement. If the execution of the statement exceeds the timeout value, the statement is timed out by the database server, and the driver throws an exception indicating that the statement was timed out. The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce ignore any value set using this method. Use the WSTimeout connection property to set a timeout. The drivers for Apache Cassandra and MongoDB driver ignore any value set using this method.
<T> T unwrap(Class<T> iface)	4.0	Yes	

StatementEventListener

StatementEventListener Methods	Version Introduced	Supported	Comments
void statementClosed(event)	4.0	Yes	
void statementErrorOccurred(event)	4.0	Yes	

Struct

Struct Methods	Version Introduced	Supported	Comments
(all)	2.0	Yes	Supported for the Oracle driver only. All other drivers throw "unsupported method" exception.

XAConnection

XAConnection Methods	Version Introduced	Supported	Comments
(all)	2.0 Optional	Yes	Supported for all drivers except Amazon Redshift, Apache Hive, Apache Spark SQL, Autonomous REST Connector, DB2 V8.1 for z/OS, Greenplum, Impala, Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, PostgreSQL, and Salesforce.

XADataSource

XADataSource Methods	Version Introduced	Supported	Comments
(all)	2.0 Optional	Yes	Supported for all drivers except Amazon Redshift, Apache Hive, Apache Spark SQL, Autonomous REST Connecter, DB2 V8.1 for z/OS, Greenplum, Impala, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, PostgreSQL, and Salesforce.

XAResource

XAResource Methods	Version Introduced	Supported	Comments
(all)	2.0 Optional	Yes	Supported for all drivers except Amazon Redshift, Apache Hive, Apache Spark SQL, DB2 V8.1 for z/OS, Greenplum, Impala, Oracle Eloqua, Oracle Sales Cloud, and PostgreSQL.

8

JDBC extensions

This section describes the JDBC extensions provided by the com.ddtek.jdbc.extensions package. Some extensions apply to select drivers. In some cases, the functionality described may not apply to the driver or data store you are using. The interfaces in the com.ddtek.jdbc.extensions are:

Interface/Class	Description
DatabaseMetadata	The methods in this interface are used with the Autonomous REST Connector and the drivers for Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, Salesforce, and Google BigQuery to extend the standard JDBC metadata results returned by the DatabaseMetaData.getColumns() method to include an additional column.
DDBulkLoad	Methods that allow your application to perform bulk load operations.
ExtConnection	Methods that allow you to perform the following actions: • Store and return client information. • Switch the user associated with a connection to another user to minimize the number of connections that are required in a connection pool. • Access the DataDirect Statement Pool Monitor from a connection.

Interface/Class	Description
ExtDatabaseMetaData	Methods that allow your application to return client information parameters.
ExtLogControl	Methods that allow you to determine if DataDirect Spy logging is enabled and turning on and off DataDirect Spy logging if enabled.

For details, see the following topics:

- Using JDBC wrapper methods to access JDBC extensions
- DatabaseMetaData interface
- DDBulkLoad interface
- ExtConnection interface
- ExtDatabaseMetaData interface
- ExtLogControl class

Using JDBC wrapper methods to access JDBC extensions

The wrapper methods allow an application to access vendor-specific classes. The following example shows how to access the DataDirect-specific ExtConnection class using the wrapper methods:

```
ExtStatementPoolMonitor monitor = null;
Class<ExtConnection> cls = ExtConnection.class;
if (con.isWrapperFor(cls)) {
   ExtConnection extCon = con.unwrap(cls);
   extCon.setClientUser("Joe Smith");
   monitor = extCon.getStatementPoolMonitor();
}
...
if(monitor != null) {
   long hits = monitor.getHitCount();
   long misses = monitor.getMissCount();
}
...
```

DatabaseMetaData interface

The following drivers support the `DatabaseMetaData.getColumns()` method:

- Autonomous REST Connector
- Jira
- Oracle Eloqua
- Oracle Sales Cloud
- Oracle Service Cloud
- Salesforce

The `DatabaseMetaData.getColumns()` method extends the standard JDBC metadata results returned to include the additional columns described in the following tables.

Table 20: DatabaseMetaData.getColumns() method

Column	Data Type	Description
IS_EXTERNAL_ID	VARCHAR (3), NOT NULL	Provides an indication of whether the column can be used as an External ID. External ID columns can be used as the lookup column for insert and upsert operations and foreign-key relationship values. Valid values are: • `YES`: The column can be used as an external ID. • `NO`: The column cannot be used as an external ID. The standard catalog table SYSTEM_COLUMNS is also extended to include the IS_EXTERNAL_ID column.
LABEL	VARCHAR (128)	The text label for this column. If not present, this field is null.

The Autonomous REST Connector and the drivers for Jira, Oracle Eloqua, Oracle Sales Cloud, Oracle Service Cloud, and Salesforce extend the standard JDBC metadata results returned by the `DatabaseMetaData.getTables()` method to include the following additional column.

Table 21: DatabaseMetaData.getTables() Method

Column	Data Type	Description
LABEL	VARCHAR (128)	The text label for this table. If not present, this field is null.

DDBulkLoad interface

DDBulkLoad Methods	Description
void clearWarnings()	Clears all warnings that were generated by this DDBulkLoad object.

DDBulkLoad Methods	Description
void close()	Releases a DDBulkLoad object's resources immediately instead of waiting for the connection to close.
long export(File)	Exports all rows from the table into the specified CSV file specified by a file reference. The table is specified using the setTableName() method. If the CSV file does not already exist, the driver creates it when the export() method is executed. In addition, the driver creates a bulk load configuration file matching the CSV file. This method also returns the number of rows that were successfully exported from the table.
long export(ResultSet, File)	Exports all rows from the specified ResultSet into the CSV file specified by a file reference. If the CSV file does not already exist, the driver creates it when the export() method is executed. In addition, the driver creates a bulk load configuration file matching the CSV file. This method also returns the number of rows that were successfully exported from the ResultSet object.
long export(String)	Exports all rows from the table into the CSV file specified by name. The table is specified using the setTableName() method. If the CSV file does not already exist, the driver creates it when the export() method is executed. In addition, the driver creates a bulk load configuration file matching the CSV file. This method also returns the number of rows that were successfully exported from the table.
long getBatchSize()	Returns the number of rows that the driver sends at a time when bulk loading data.
long getBinaryThreshold()	Returns the maximum size (in bytes) of binary data that can be exported to the CSV file. Once this size is reached, binary data is written to one or multiple external overflow files.
long getCharacterThreshold()	Returns the maximum size (in bytes) of character data that can be exported to the CSV file. Once this size is reached, character data is written to one or multiple external overflow files.
String getCodePage()	Returns the code page that the driver uses for the CSV file.
String getConfigFile()	Returns the name of the bulk load configuration file.
String getDiscardFile()	Returns the name of the discard file. The discard file contains rows that were unable to be loaded as the result of a bulk load operation.
long getErrorTolerance()	Returns the number of errors that can occur before this DDBulkLoad object ends the bulk load operation.
String getLogFile()	Returns the name of the log file. The log file records information about each bulk load operation.
long getNumRows()	Returns the maximum number of rows from the CSV file or ResultSet object the driver will load when the load() method is executed.

DDBulkLoad Methods	Description
Properties getProperties()	Returns the properties specified for a DDBulkLoad object. Properties are specified using the setProperties() method.
long getReadBufferSize()	Returns the size (in KB) of the buffer that is used to read the CSV file.
long getStartPosition()	Returns the position (number of the row) in a CSV file or ResultSet object from which the driver starts loading. The position is specified using the setStartPosition() method.
void getTableName()	Returns the name of the table to which the data is loaded into or exported from.
long getTimeout()	Returns the number of seconds the bulk load operation requires to complete before it times out. The timeout is specified using the setTimeout() method.
SQLWarning getWarnings()	Returns any warnings generated by this DDBulkLoad object.
long getWarningTolerance()	Returns the maximum number of warnings that can occur. Once the maximum number is reached, the bulk load operation ends.
long load(File)	Loads data from the CSV file specified by a file reference into a table. The table is specified using the setTableName() method. This method also returns the number of rows that have been successfully loaded. If logging is enabled using the setLogFile() method, information about the bulk load operation is recorded in the log file. If a discard file is created using the setDiscardFile() method, rows that were unable to be loaded are recorded in the discard file. Before the bulk load operation is performed, your application can verify that the data in the CSV file is compatible with the structure of the target table using the validateTableFromFile() method.
long load(String)	Loads data from the CSV file specified by file name into a table. The table is specified using the setTableName() method. This method also returns the number of rows that have been successfully loaded. If logging is enabled using the setLogFile() method, information about the bulk load operation is recorded in the log file. If a discard file is created using the setDiscardFile() method, rows that were unable to be loaded are recorded in the discard file. Before the bulk load operation is performed, your application can verify that the data in the CSV file is compatible with the structure of the target table using the validateTableFromFile() method.

DDBulkLoad Methods	Description
long load(ResultSet)	Loads data from a ResultSet object into the table specified using the setTableName() method. This method also returns the number of rows that have been successfully loaded. If logging is enabled using the setLogFile() method, information about the bulk load operation is recorded in the log file. The structure of the table that produced the ResultSet object must match the structure of the target table. If not, the driver throws an exception.
void setBatchSize(long)	Specifies the number of rows that the driver sends at a time when bulk loading data. Performance can be improved by increasing the number of rows the driver loads at a time because fewer network round trips are required. Be aware that increasing the number of rows that are loaded also causes the driver to consume more memory on the client. If unspecified, the driver uses a value of `2048`.

void setBinaryThreshold(long)	Specifies the maximum size (in bytes) of binary data to be exported to the CSV file. Any column with data over this threshold is exported into individual external overflow files and a marker of the format {DD LOBFILE "*filename*"} is placed in the CSV file to signify that the data for this column is located in an external file. The format for overflow file names is:
	csv_filename_xxxxxx.lob
	where:
	csv_filename
	is the name of the CSV file.
	xxxxxx
	is a 6-digit number that increments the overflow file.
	For example, if multiple overflow files are created for a CSV file named CSV1, the file names would look like this:
	`CSV1.000001.lob`
	`CSV1.000002.lob`
	`CSV1.000003.lob`
	`...`
	If set to `-1`, the driver does not overflow binary data to external files. If unspecified, the driver uses a value of `4096`.

DDBulkLoad Methods	Description
void setCharacterThreshold(long)	Specifies the maximum size (in bytes) of character data to be exported to the CSV file. Any column with data over this threshold is exported into individual external overflow files and a marker of the format {DD LOBFILE "*filename*"} is placed in the CSV file to signify that the data for this column is located in an external file. The format for overflow file names is: *csv_filename_xxxxxx*.lob where: *csv_filename* is the name of the CSV file. *xxxxxx* is a 6-digit number that increments the overflow file. For example, if multiple overflow files are created for a CSV file named CSV1, the file names would look like this: CSV1.000001.lob CSV1.000002.lob CSV1.000003.lob ... If set to -1, the driver does not overflow character data to external files.If unspecified, the driver uses a value of 4096.
void setCodePage(String)	Specifies the code page the driver uses for the CSV file.
void setConfigFile(String)	Specifies the fully qualified directory and file name of the bulk load configuration file. If the Column Info section in the bulk load configuration file is specified, the driver uses it to map the columns in the CSV file to the columns in the target table when performing a bulk load operation. If unspecified, the name of the bulk load configuration file is assumed to be *csv_filename*.xml, where *csv_filename* is the file name of the CSV file. If set to an empty string, the driver does not try to use the bulk load configuration file and reads all data from the CSV file as character data.
void setDiscardFile(String)	Specifies the fully qualified directory and file name of the discard file. The discard file contains rows that were unable to be loaded from a CSV file as the result of a bulk load operation. After fixing the reported issues in the discard file, the bulk load can be reissued, using the discard file as the CSV file. If unspecified, a discard file is not created.

DDBulkLoad Methods	Description
void setErrorTolerance(long)	Specifies the maximum number of errors that can occur. Once the maximum number is reached, the bulk load operation ends. Errors are written to the log file. If set to `0`, no errors are tolerated; the bulk load operation fails if any error is encountered. Any rows that were processed before the error occurred are loaded. If unspecified or set to `-1`, an infinite number of errors are tolerated.
void setLogFile(String)	Specifies the fully qualified directory and file name of the log file. The log file records information about each bulk load operation.If unspecified, a log file is not created.
void setNumRows()	Specifies the maximum number of rows from the CSV file or ResultSet object the driver will load.

void setProperties(Properties)	Specifies one or more of the following properties for a DDBulkLoad object:
	`tableName`
	`codePage`
	`timeout`
	`logFile`
	`discardFile`
	`configFile`
	`startPosition`
	`operation`
	`numRows`
	`binaryThreshold`
	`characterThreshold`
	`errorTolerance`
	`warningTolerance`
	`readBufferSize`
	`batchSize`
	Except for the operation property, these properties also can be set using the corresponding set*xxx*() methods, which provide a description of the values that can be set.
	The operation property defines which type of bulk operation will be performed when a load method is called. The operation property accepts the following values: `insert`, `update`, `delete`, or `upsert`. The default value is `insert`.

void setReadBufferSize(long)	Specifies the size (in KB) of the buffer that is used to read the CSV file. If unspecified, the driver uses a value of `2048`.
void setStartPosition()	Specifies the position (number of the row) in a CSV file or ResultSet object from which the bulk load operation starts. For example, if a value of `10` is specified, the first 9 rows of the CSV file are skipped and the first row loaded is row 10.

DDBulkLoad Methods	Description

void setTableName(*tablename* ([*destinationColumnList*]))	When loading data into a table, specifies the name of the table into which the data is loaded (*tablename*).
	Optionally, for the Salesforce driver, you can specify the column names that identify which columns to update in the table (*destinationColumnList*). Specifying column names is useful when loading data from a CSV file into a table. The column names used in the column list must be the names reported by the driver for the columns in the table. For example, if you are loading data into the Salesforce system column NAME, the column list must identify the column as SYS_NAME.
	If *destinationColumnList* is not specified, a one-to-one mapping is performed between the columns in the CSV file and the columns in the table.
	destinationColumnList has the following format:
	`(`*destColumnName* `[,`*destColumnName*`]...)`
	where:
	destColumnName
	is the name of the column in the table to be updated.
	The number of specified columns must match the number of columns in the CSV file. For example, the following call tells the driver to update the Name, Address, City, State, PostalCode, Phone, and Website columns:
	`bulkload.setTableName("account(Name, Address, City,State, PostalCode, Phone, Website)")`
	When exporting data from a table, specifies the name of the table from which the data is exported. If the specified table does not exist, the driver throws an exception.

void setTimeout(long)	Sets the maximum number of seconds that can elapse for this bulk load operation to complete. Once this number is reached, the bulk load operation times out.

DDBulkLoad Methods	Description
void setWarningTolerance(long)	Specifies the maximum number of warnings that can occur. Once the maximum is reached, the bulk load operation ends. Warnings are written to the log file. If set to `0`, no warnings are tolerated; the bulk load operation fails if any warning is encountered. If unspecified or set to `-1`, an infinite number of warnings are tolerated.
Properties validateTableFromFile()	Verifies the metadata in the bulk load configuration file against the structure of the table to which the data is loaded. This method is used to ensure that the data in a CSV file is compatible with the structure of the target table before the actual bulk load operation is performed. The driver performs checks to detect mismatches of the following types: Data types Column sizes Code pages Column info This method returns a Properties object with an entry for each of these checks: • If no mismatches are found, the Properties object does not contain any messages. • If minor mismatches are found, the Properties object lists the problems. • If problems are detected that would prevent a successful bulk load operation, for example, if the target table does not exist, the driver throws an exception.

ExtConnection interface

The methods of this interface are supported for all drivers.

ExtConnection Methods	Description
void abortConnection()	Closes the current connection and marks the connection as closed. This method does not attempt to obtain any locks when closing the connection. If subsequent operations are performed on the connection, the driver throws an exception.
Connection createArray(String, Object[])	Supported by the Oracle driver only for use with Oracle VARRAY and TABLE data types. Creates an array object.

ExtConnection Methods	Description
String getClientAccountingInfo()	Returns the accounting client information on the connection or an empty string if the accounting client information value or the connection has not been set. If getting accounting client information is supported by the database and this operation fails, the driver throws an exception.
String getClientApplicationName()	Returns the name of the client application on the connection or an empty string if the client name value for the connection has not been set. If getting client name information is supported by the database and this operation fails, the driver throws an exception.
String getClientHostname()	Returns the name of the host used by the client application on the connection or an empty string if the client hostname value in the database has not been set. If getting host name information is supported by the database and this operation fails, the driver throws an exception.
String getClientUser()	Returns the user ID of the client on the connection or an empty string if the client user ID value for the connection has not been set. The user ID may be different from the user ID establishing the connection. If getting user ID application information is supported by the database and this operation fails, the driver throws an exception.
String getCurrentUser()	Returns the current user of the connection. If reauthentication was performed on the connection, the current user may be different than the user that created the connection. For the DB2 and Oracle drivers, the current user is the same as the user reported by DatabaseMetaData.getUserName(). For the SQL Server driver, the current user is the login user name. DatabaseMetaData.getUserName() reports the user name the login user name is mapped to in the database.
int getNetworkTimeout()	Supported by the SQL Server driver to return the network timeout. The network timeout is the maximum time (in milliseconds) that a connection, or objects created by a connection, will wait for the database to reply to an application request. A value of 0 means that no network timeout exists. See void setNetworkTimeout(int) for details about setting a network timeout.
ExtStatementPoolMonitor getStatementPoolMonitor()	Returns an ExtStatementPoolMonitor object for the statement pool associated with the connection. If the connection does not have a statement pool, this method returns null.

ExtConnection Methods	Description
void resetUser(String)	Specifies a non-null string that resets the current user on the connection to the user that created the connection. It also restores the current schema, current path, or current database to the original value used when the connection was created. If reauthentication was performed on the connection, this method is useful to reset the connection to the original user. For the SQL Server driver, the current user is the login user name.The driver throws an exception in the following circumstances: • The driver cannot change the current user to the initial user. • A transaction is active on the connection.
void setClientAccountingInfo(String)	Specifies a non-null string that sets the accounting client information on the connection. Some databases include this information in their usage reports. The maximum length allowed for accounting information for a particular database can be determined by calling the ExtDatabaseMetaData.getClientAccountingInfoLength() method. If the length of the information specified is longer than the maximum length allowed, the information is truncated to the maximum length, and the driver generates a warning. If setting accounting client information is supported by the database and this operation fails, the driver throws an exception.
void setClientApplicationName(String)	Specifies a non-null string that sets the name of the client application on the connection. The maximum client name length allowed for a particular database can be determined by calling the ExtDatabaseMetaData.getClientApplicationNameLength() method. If the length of the client application name specified is longer than the maximum name length allowed, the name is truncated to the maximum length allowed, and the driver generates a warning. If setting client name information is supported by the database and this operation fails, the driver throws an exception.
void setClientHostname(String)	Specifies a non-null string that sets the name of the host used by the client application on the connection. The maximum hostname length allowed for a particular database can be determined by calling the ExtDatabaseMetaData.getClientHostnameLength() method. If the length of the hostname specified is longer than the maximum hostname length allowed, the hostname is truncated to the maximum hostname length, and the driver generates a warning. If setting hostname information is supported by the database and this operation fails, the driver throws an exception.

ExtConnection Methods	Description
void setClientUser(String)	Specifies a non-null string that sets the user ID of the client on the connection. This user ID may be different from the user ID establishing the connection. The maximum user ID length allowed for a particular database can be determined by calling the ExtDatabaseMetaData.getClientUserLength() method. If the length of the user ID specified is longer than the maximum length allowed, the user ID is truncated to the maximum user ID length, and the driver generates a warning. If setting user ID information is supported by the database and this operation fails, the driver throws an exception.
void setCurrentUser(String)	Specifies a non-null string that sets the current user on the connection. This method is used to perform reauthentication on a connection. For the SQL Server driver, the current user is the login user name. The driver throws an exception in the following circumstances: • The driver is connected to a database server that does not support reauthentication. • The database server rejects the request to change the user on the connection. • A transaction is active on the connection.
void setCurrentUser(String, Properties)	Specifies a non-null string that sets the current user on the connection. This method is used to perform reauthentication on a connection. In addition, this method sets options that control how the driver handles reauthentication. The options that are supported depend on the driver. See the DB2 driver, Oracle driver, and SQL Server driver chapters for information on which options are supported by each driver. For the SQL Server driver, the current user is the login user name. The driver throws an exception in the following circumstances: • The driver is connected to a database server that does not support reauthentication. • The database server rejects the request to change the user on the connection. • A transaction is active on the connection.

ExtConnection Methods	Description
void setCurrentUser(javax.security.auth.Subject)	Specifies a non-null string that sets the current user on the connection to the user specified by the javax.security.auth.Subject object. This method is used to perform reauthentication on a connection. For the SQL Server driver, the current user is the login user name. The driver throws an exception in the following circumstances: • The driver does not support reauthentication. • The driver is connected to a database server that does not support reauthentication. • The database server rejects the request to change the user on the connection. • A transaction is active on the connection.
void setCurrentUser(javax.security.auth.Subject, Properties)	Specifies a non-null string that sets the current user on the connection to the user specified by the javax.security.auth.Subject object. This method is used to perform reauthentication on a connection. In addition, this method sets options that control how the driver handles reauthentication. The options that are supported depend on the driver. See your user's guide for information on which options are supported by each driver. For the SQL Server driver, the current user is the login user name. The driver throws an exception in the following circumstances: • The driver does not support reauthentication. • The driver is connected to a database server that does not support reauthentication. • The database server rejects the request to change the user on the connection. • A transaction is active on the connection.
void setNetworkTimeout(int)	Supported by the SQL Server driver to set the network timeout. The network timeout is the maximum time (in milliseconds) that a connection, or objects created by a connection, will wait for the database to reply to an application request. If this limit is exceeded, the connection or objects are closed and the driver returns an exception indicating that a timeout occurred. A value of 0 means that no network timeout exists. Note that if a query timeout occurs before a network timeout, the execution of the statement is cancelled. Both the connection and the statement can be used. If a network timeout occurs before a query timeout or if the query timeout fails because of network problems, the connection is closed and neither the connection or the statement can be used.
boolean supportsReauthentication()	Indicates whether the connection supports reauthentication. If true is returned, you can perform reauthentication on the connection. If false is returned, any attempt to perform reauthentication on the connection throws an exception.

ExtDatabaseMetaData interface

ExtDatabaseMetaData Methods	Description
int getClientApplicationNameLength()	Returns the maximum length of the client application name. A value of 0 indicates that the client application name is stored locally in the driver, not in the database. There is no maximum length if the application name is stored locally.
int getClientUserLength()	Returns the maximum length of the client user ID. A value of 0 indicates that the client user ID is stored locally in the driver, not in the database. There is no maximum length if the client user ID is stored locally.
int getClientHostnameLength()	Returns the maximum length of the hostname. A value of 0 indicates that the hostname is stored locally in the driver, not in the database. There is no maximum length if the hostname is stored locally.
int getClientAccountingInfoLength()	Returns the maximum length of the accounting information. A value of 0 indicates that the accounting information is stored locally in the driver, not in the database. There is no maximum length if the hostname is stored locally.

ExtLogControl class

ExtLogControl Methods	Description
void setEnableLogging(boolean enable|disable)	If DataDirect Spy was enabled when the connection was created, you can turn on or off DataDirect Spy logging at runtime using this method. If true, logging is turned on. If false, logging is turned off. If DataDirect Spy logging was not enabled when the connection was created, calling this method has no effect.
boolean getEnableLogging()	Indicates whether DataDirect Spy logging was enabled when the connection was created and whether logging is turned on. If the returned value is true, logging is turned on. If the returned value is false, logging is turned off.

9

Designing JDBC applications for performance optimization

Developing performance-oriented JDBC applications is not easy. JDBC drivers do not throw exceptions to tell you when your code is running too slow. This chapter presents some general guidelines for improving JDBC application performance that have been compiled by examining the JDBC implementations of numerous shipping JDBC applications. These guidelines include:

- Use DatabaseMetaData methods appropriately
- Return only required data
- Select functions that optimize performance
- Manage connections and updates

Following these general guidelines can help you solve some common JDBC system performance problems, such as those listed in the following table.

Problem	Solution	See guidelines in…
Network communication is slow.	Reduce network traffic.	Using database metadata methods on page 346
Evaluation of complex SQL queries on the database server is slow and can reduce concurrency.	Simplify queries.	Using database metadata methods on page 346 Selecting JDBC objects and methods on page 350

Problem	Solution	See guidelines in…
Excessive calls from the application to the driver slow performance.	Optimize application-to-driver interaction.	Returning data on page 348 Selecting JDBC objects and methods on page 350
Disk I/O is slow.	Limit disk I/O.	Managing connections and updates on page 353

In addition, most JDBC drivers provide options that improve performance, often with a trade-off in functionality. If your application is not affected by functionality that is modified by setting a particular option, significant performance improvements can be realized.

Note: The section describes functionality across a spectrum of data stores. In some cases, the functionality described may not apply to the driver or data store you are using. In addition, examples are drawn from a variety of drivers and data stores.

For details, see the following topics:

- Using database metadata methods
- Returning data
- Selecting JDBC objects and methods
- Managing connections and updates

Using database metadata methods

Because database metadata methods that generate ResultSet objects are slow compared to other JDBC methods, their frequent use can impair system performance. The guidelines in this section will help you optimize system performance when selecting and using database metadata.

Minimizing the use of database metadata methods

Compared to other JDBC methods, database metadata methods that generate ResultSet objects are relatively slow. Applications should cache information returned from result sets that generate database metadata methods so that multiple executions are not needed.

Although almost no JDBC application can be written without database metadata methods, you can improve system performance by minimizing their use. To return all result column information *mandated* by the JDBC specification, a JDBC driver may have to perform complex queries or multiple queries to return the necessary result set for a single call to a database metadata method. These particular elements of the SQL language are performance-expensive.

Applications should cache information from database metadata methods. For example, call getTypeInfo() once in the application and cache the elements of the result set that your application depends on. It is unlikely that any application uses all elements of the result set generated by a database metadata method, so the cache of information should not be difficult to maintain.

Avoiding search patterns

Using null arguments or search patterns in database metadata methods results in generating time-consuming queries. In addition, network traffic potentially increases due to unwanted results. Always supply as many non-null arguments as possible to result sets that generate database metadata methods.

Because database metadata methods are slow, invoke them in your applications as efficiently as possible. Many applications pass the fewest non-null arguments necessary for the function to return success. For example:

```
ResultSet WSrs = WSdbmd.getTables(null, null, "WSTable", null);
```

In this example, an application uses the getTables() method to determine if the WSTable table exists. A JDBC driver interprets the request as: return all tables, views, system tables, synonyms, temporary tables, and aliases named "WSTable" that exist in any database schema inside the database catalog.

In contrast, the following request provides non-null arguments as shown:

```
String[] tableTypes = {"TABLE"};
WSdbmd.getTables("cat1", "johng", "WSTable", "tableTypes");
```

Clearly, a JDBC driver can process the second request more efficiently than it can process the first request.

Sometimes, little information is known about the object for which you are requesting information. Any information that the application can send the driver when calling database metadata methods can result in improved performance and reliability.

Using a dummy query to determine table characteristics

Avoid using the getColumns() method to determine characteristics about a database table. Instead, use a dummy query with getMetadata().

Consider an application that allows the user to choose the columns to be selected. Should the application use getColumns() to return information about the columns to the user or instead prepare a dummy query and call getMetadata()?

Case 1: GetColumns() Method

```
ResultSet WSrc = WSc.getColumns(... "UnknownTable" ...);
// This call to getColumns will generate a query to
// the system catalogs... possibly a join
// which must be prepared, executed, and produce
// a result set
. . .
WSrc.next();
string Cname = getString(4);
. . .
// user must return N rows from the server
// N = # result columns of UnknownTable
// result column information has now been obtained
```

Case 2: GetMetadata() Method

```
// prepare dummy query
PreparedStatement WSps = WSc.prepareStatement
    ("SELECT * FROM UnknownTable WHERE 1 = 0");
// query is never executed on the server - only prepared
ResultSetMetaData WSsmd=WSps.getMetaData();
int numcols = WSrsmd.getColumnCount();
...
```

```
int ctype = WSrsmd.getColumnType(n)
...
// result column information has now been obtained
// Note we also know the column ordering within the
// table!  This information cannot be
// assumed from the getColumns example.
```

In both cases, a query is sent to the server. However, in Case 1, the potentially complex query must be prepared and executed, result description information must be formulated, and a result set of rows must be sent to the client. In Case 2, we prepare a simple query where we only return result set information. Clearly, Case 2 is the better performing model.

To somewhat complicate this discussion, let us consider a DBMS server that does not natively support preparing a SQL statement. The performance of Case 1 does not change but the performance of Case 2 improves slightly because the dummy query must be evaluated in addition to being prepared. Because the Where clause of the query always evaluates to FALSE, the query generates no result rows and should execute without accessing table data. For this situation, Case 2 still outperforms Case 1.

In summary, always use result set metadata to return table column information, such as column names, column data types, and column precision and scale. Only use the getColumns() method when the requested information cannot be obtained from result set metadata (for example, using the table column default values).

Returning data

To return data efficiently, return only the data that you need and choose the most efficient method of doing so. The guidelines in this section will help you optimize system performance when retrieving data with JDBC applications.

Returning long data

Because retrieving long data across a network is slow and resource intensive, applications should not request long data unless it is necessary.

Most users do not want to see long data. If the user does want to see these result items, then the application can query the database again, specifying only the long columns in the Select list. This method allows the average user to return the result set without having to pay a high performance penalty for network traffic.

Although the best method is to exclude long data from the Select list, some applications do not formulate the Select list before sending the query to the JDBC driver (that is, some applications `SELECT * FROM` *`table_name`* `...`). If the Select list contains long data, most drivers are forced to return that long data at fetch time, even if the application does not ask for the long data in the result set. When possible, the designer should attempt to implement a method that does not return all columns of the table.

For example, consider the following code:

```
ResultSet rs = stmt.executeQuery(
   "SELECT * FROM Employees WHERE SSID = '999-99-2222'");
rs.next();
string name = rs.getString(1);
```

Remember that a JDBC driver cannot interpret an application's final intention. When a query is executed, the driver has no way to know which result columns an application will use. A driver anticipates that an application can request any of the result columns that are returned. When the JDBC driver processes the rs.next request, it will probably return at least one, if not more, result rows from the database server across the network. In this case, a result row contains all the column values for each row, including an employee photograph if the Employees table contains such a column. If you limit the Select list to contain only the employee name column, it results in decreased network traffic and a faster performing query at runtime. For example:

```
ResultSet rs = stmt.executeQuery(
   "SELECT name FROM Employees WHERE SSID = '999-99-2222'");
rs.next();
string name = rs.getString(1);
```

Additionally, although the getClob() and getBlob() methods allow the application to control how long data is returned in the application, the designer must realize that in many cases, the JDBC driver emulates these methods due to the lack of true Large Object (LOB) locator support in the DBMS. In such cases, the driver must return all the long data across the network before exposing the getClob() and getBlob() methods.

Reducing the size of returned data

Sometimes long data must be returned. When this is the case, remember that most users do not want to see 100 KB, or more, of text on the screen.

To reduce network traffic and improve performance, you can reduce the size of any data being returned to some manageable limit by calling setMaxRows(), setMaxFieldSize(), and the driver-specific setFetchSize(). Another method of reducing the size of the data being returned is to decrease the column size.

In addition, be careful to return only the rows you need. If you return five columns when you only need two columns, performance is decreased, especially if the unnecessary rows include long data.

Choosing the right data type

Retrieving and sending certain data types can be expensive. When you design a schema, select the data type that can be processed most efficiently. For example, integer data is processed faster than floating-point data. Floating-point data is defined according to internal database-specific formats, usually in a compressed format. The data must be decompressed and converted into a different format so that it can be processed by the database wire protocol.

Retrieving result sets

Most JDBC drivers cannot implement scrollable cursors because of limited support for scrollable cursors in the database system. Unless you are certain that the database supports using a scrollable result set, rs, for example, do not call rs.last and rs.getRow() methods to find out how many rows the result set contains. For JDBC drivers that emulate scrollable cursors, calling rs.last results in the driver retrieving all results across the network to reach the last row. Instead, you can either count the rows by iterating through the result set or get the number of rows by submitting a query with a Count column in the Select clause.

In general, do not write code that relies on the number of result rows from a query because drivers must fetch all rows in a result set to know how many rows the query will return.

Selecting JDBC objects and methods

The guidelines in this section will help you to select which JDBC objects and methods will give you the best performance.

Using parameter markers as arguments to stored procedures

When calling stored procedures, always use parameter markers for argument markers instead of using literal arguments. JDBC drivers can call stored procedures on the database server either by executing the procedure as a SQL query or by optimizing the execution by invoking a Remote Procedure Call (RPC) directly on the database server. When you execute a stored procedure as a SQL query, the database server parses the statement, validates the argument types, and converts the arguments into the correct data types.

Remember that SQL is always sent to the database server as a character string, for example, `{call getCustName(12345)}`. In this case, even though the application programmer may have assumed that the only argument to getCustName() was an integer, the argument is actually passed inside a character string to the server. The database server parses the SQL query, isolates the single argument value `12345`, and converts the string `12345` into an integer value before executing the procedure as a SQL language event.

By invoking a RPC on the database server, the overhead of using a SQL character string is avoided. Instead, the JDBC driver constructs a network packet that contains the parameters in their native data type formats and executes the procedure remotely.

Case 1: Not Using a Server-Side RPC

In this example, the stored procedure getCustName() cannot be optimized to use a server-side RPC. The database server must treat the SQL request as a normal language event, which includes parsing the statement, validating the argument types, and converting the arguments into the correct data types before executing the procedure.

```
CallableStatement cstmt = conn.prepareCall("call getCustName(12345)");
ResultSet rs = cstmt.executeQuery();
```

Case 2: Using a Server-Side RPC

In this example, the stored procedure getCustName() can be optimized to use a server-side RPC. Because the application avoids literal arguments and calls the procedure by specifying all arguments as parameters, the JDBC driver can optimize the execution by invoking the stored procedure directly on the database as an RPC. The SQL language processing on the database server is avoided and execution time is greatly improved.

```
CallableStatement cstmt = conn.prepareCall("call getCustName(?)}");
cstmt.setLong(1,12345);
ResultSet rs = cstmt.executeQuery();
```

Using the statement object instead of the PreparedStatement object

JDBC drivers are optimized based on the perceived use of the functions that are being executed. Choose between the PreparedStatement object and the Statement object depending on how you plan to use the object. The Statement object is optimized for a single execution of a SQL statement. In contrast, the PreparedStatement object is optimized for SQL statements to be executed two or more times.

The overhead for the initial execution of a PreparedStatement object is high. The advantage comes with subsequent executions of the SQL statement. For example, suppose we are preparing and executing a query that returns employee information based on an ID. Using a PreparedStatement object, a JDBC driver would process the prepare request by making a network request to the database server to parse and optimize the query. The execute results in another network request. If the application will only make this request once during its life span, using a Statement object instead of a PreparedStatement object results in only a single network roundtrip to the database server. Reducing network communication typically provides the most performance gains.

This guideline is complicated by the use of prepared statement pooling because the scope of execution is longer. When using prepared statement pooling, if a query will only be executed once, use the Statement object. If a query will be executed infrequently, but may be executed again during the life of a statement pool inside a connection pool, use a PreparedStatement object. Under similar circumstances without statement pooling, use the Statement object.

Using batches instead of prepared statements

Updating large amounts of data typically is done by preparing an Insert statement and executing that statement multiple times, resulting in numerous network roundtrips. To reduce the number of JDBC calls and improve performance, you can send multiple queries to the database at a time using the addBatch method of the PreparedStatement object. For example, let us compare the following examples, Case 1 and Case 2.

Case 1: Executing Prepared Statement Multiple Times

```
PreparedStatement ps = conn.prepareStatement(
   "INSERT INTO employees VALUES (?, ?, ?)");
for (n = 0; n < 100; n++) {
   ps.setString(name[n]);
   ps.setLong(id[n]);
   ps.setInt(salary[n]);
   ps.executeUpdate();
}
```

Case 2: Using a Batch

```
PreparedStatement ps = conn.prepareStatement(
   "INSERT INTO employees VALUES (?, ?, ?)");
for (n = 0; n < 100; n++) {
   ps.setString(name[n]);
   ps.setLong(id[n]);
   ps.setInt(salary[n]);
   ps.addBatch();
}
ps.executeBatch();
```

In Case 1, a prepared statement is used to execute an Insert statement multiple times. In this case, 101 network roundtrips are required to perform 100 Insert operations: one roundtrip to prepare the statement and 100 additional roundtrips to execute its iterations. When the addBatch method is used to consolidate 100 Insert operations, as demonstrated in Case 2, only two network roundtrips are required—one to prepare the statement and another to execute the batch. Although more database CPU cycles are involved by using batches, performance is gained through the reduction of network roundtrips. Remember that the biggest gain in performance is realized by reducing network communication between the JDBC driver and the database server.

Choosing the right cursor

Choosing the appropriate type of cursor allows maximum application flexibility. This section summarizes the performance issues of three types of cursors: forward-only, insensitive, and sensitive.

A *forward-only cursor* provides excellent performance for sequential reads of all rows in a table. For retrieving table data, there is no faster way to return result rows than using a forward-only cursor; however, forward-only cursors cannot be used when the rows to be returned are not sequential.

Insensitive cursors are ideal for applications that require high levels of concurrency on the database server and require the ability to scroll forwards and backwards through result sets. The first request to an insensitive cursor fetches all the rows and stores them on the client. In most cases, the first request to an insensitive cursor fetches all the rows and stores them on the client. If a driver uses "lazy" fetching (fetch-on-demand), the first request may include many rows, if not all rows.The initial request is slow, especially when long data is returned. Subsequent requests do not require any network traffic (or, when a driver uses "lazy" fetching, requires limited network traffic) and are processed quickly.

Because the first request is processed slowly, insensitive cursors should not be used for a single request of one row. Developers should also avoid using insensitive cursors when long data or large result sets are returned because memory can be exhausted. Some insensitive cursor implementations cache the data in a temporary table on the database server and avoid the performance issue, but most cache the information local to the application.

Sensitive cursors, or keyset-driven cursors, use identifiers such as a ROWID that already exist in the database. When you scroll through the result set, the data for these identifiers is returned. Because each request generates network traffic, performance can be very slow. However, returning non-sequential rows does not further affect performance.

To illustrate this point further, consider an application that normally returns 1000 rows to an application. At execute time, or when the first row is requested, a JDBC driver does not execute the Select statement that was provided by the application. Instead, the JDBC driver replaces the Select list of the query with a key identifier, for example, ROWID. This modified query is then executed by the driver and all 1000 key values are returned by the database server and cached for use by the driver. Each request from the application for a result row directs the JDBC driver to look up the key value for the appropriate row in its local cache, construct an optimized query that contains a Where clause similar to WHERE ROWID=?, execute the modified query, and return the single result row from the server.

Sensitive cursors are the preferred scrollable cursor model for dynamic situations when the application cannot afford to buffer the data associated with an insensitive cursor.

Using get methods effectively

JDBC provides a variety of methods to return data from a result set (for example, getInt(), getString(), and getObject()). The getObject() method is the most generic and provides the worst performance when the non-default mappings are specified because the JDBC driver must perform extra processing to determine the type of the value being returned and generate the appropriate mapping. Always use the specific method for the data type.

To further improve performance, provide the column number of the column being returned, for example, `getString(1)`, `getLong(2)`, and `getInt(3)`, instead of the column name. If the column names are not specified, network traffic is unaffected, but costly conversions and lookups increase. For example, suppose you use:

```
getString("foo")...
```

The JDBC driver may need to convert foo to uppercase and then compare foo with all columns in the column list, which is costly. If the driver is able to go directly to result column 23, a large amount of processing is saved.

For example, suppose you have a result set that has 15 columns and 100 rows, and the column names are not included in the result set. You are interested in only three columns: EMPLOYEENAME (string), EMPLOYEENUMBER (long integer), and SALARY (integer). If you specify `getString("EmployeeName")`, `getLong("EmployeeNumber")`, and `getInt("Salary")`, each column name must be converted to the appropriate case of the columns in the database metadata and lookups would increase considerably. Performance improves significantly if you specify `getString(1)`, `getLong(2)`, and `getInt(15)`.

Retrieving auto-generated keys

Many databases have hidden columns (pseudo-columns) that represent a unique key for each row in a table. Typically, using these types of columns in a query is the fastest way to access a row because the pseudo-columns usually represent the physical disk address of the data. Prior to JDBC 3.0, an application could only return the value of the pseudo-columns by executing a Select statement immediately after inserting the data. For example:

```
//insert row
int rowcount = stmt.executeUpdate (
   "INSERT INTO LocalGeniusList (name)
   VALUES ('Karen')");
// now get the disk address – rowid -
// for the newly inserted row
ResultSet rs = stmt.executeQuery (
   "SELECT rowid FROM LocalGeniusList
   WHERE name = 'Karen'");
```

Retrieving pseudo-columns this way has two major flaws. First, retrieving the pseudo-column requires a separate query to be sent over the network and executed on the server. Second, because there may not be a primary key over the table, the search condition of the query may be unable to uniquely identify the row. In the latter case, multiple pseudo-column values can be returned, and the application may not be able to determine which value is actually the value for the most recently inserted row.

An optional feature of the JDBC 3.0 specification is the ability to return auto-generated key information for a row when the row is inserted into a table. For example:

```
int rowcount = stmt.executeUpdate(
   "INSERT INTO LocalGeniusList(name) VALUES('Karen')",
// insert row AND return key
Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
// key is automatically available
```

Now, the application contains a value that can be used in a search condition to provide the fastest access to the row and a value that uniquely identifies the row, even when a primary key doesn't exist on the table.

The ability to return keys provides flexibility to the JDBC developer and creates performance boosts when accessing data.

Managing connections and updates

The guidelines in this section will help you to manage connections and updates to improve system performance for your JDBC applications.

Managing connections

Connection management is important to application performance. Optimize your application by connecting once and using multiple Statement objects, instead of performing multiple connections. Avoid connecting to a data source after establishing an initial connection.

Although gathering driver information at connect time is a good practice, it is often more efficient to gather it in one step rather than two steps. For example, some applications establish a connection and then call a method in a separate component that reattaches and gathers information about the driver. Applications that are designed as separate entities should pass the established connection object to the data collection routine instead of establishing a second connection.

Another bad practice is to connect and disconnect several times throughout your application to perform SQL statements. Connection objects can have multiple Statement objects associated with them. Statement objects, which are defined to be memory storage for information about SQL statements, can manage multiple SQL statements.

You can improve performance significantly with connection pooling, especially for applications that connect over a network or through the World Wide Web. Connection pooling lets you reuse connections. Closing connections does not close the physical connection to the database. When an application requests a connection, an active connection is reused, thus avoiding the network round trips needed to create a new connection.

Typically, you can configure a connection pool to provide scalability for connections. The goal is to maintain a reasonable connection pool size while ensuring that each user who needs a connection has one available within an acceptable response time. To achieve this goal, you can configure the minimum and maximum number of connections that are in the pool at any given time, and how long idle connections stay in the pool. In addition, to help minimize the number of connections required in a connection pool, you can switch the user associated with a connection to another user, a process known as *reauthentication*. Not all databases support reauthentication.

In addition to connection pooling tuning options, JDBC also specifies semantics for providing a prepared statement pool. Similar to connection pooling, a prepared statement pool caches PreparedStatement objects so that they can be re-used from a cache without application intervention. For example, an application may create a PreparedStatement object similar to the following SQL statement:

```
SELECT name, address, dept, salary FROM personnel
WHERE empid = ? or name = ? or address = ?
```

When the PreparedStatement object is created, the SQL query is parsed for semantic validation and a query optimization plan is produced. The process of creating a prepared statement can be extremely expensive in terms of performance with some database systems. Once the prepared statement is closed, a JDBC 3.0-compliant driver places the prepared statement into a local cache instead of discarding it. If the application later attempts to create a prepared statement with the same SQL query, a common occurrence in many applications, the driver can simply retrieve the associated statement from the local cache instead of performing a network roundtrip to the server and an expensive database validation.

Connection and statement handling should be addressed before implementation. Thoughtfully handling connections and statements improves application performance and maintainability.

Managing commits in transactions

Committing transactions is slow because of the amount of disk I/O and potentially network round trips that are required. Always turn off Autocommit by using `Connection.setAutoCommit(false)`.

What does a commit actually involve? The database server must flush back to disk every data page that contains updated or new data. This is usually a sequential write to a journal file, but nevertheless, it involves disk I/O. By default, Autocommit is on when connecting to a data source, and Autocommit mode usually impairs performance because of the significant amount of disk I/O needed to commit every operation.

Furthermore, most database servers do not provide a native Autocommit mode. For this type of server, the JDBC driver must explicitly issue a COMMIT statement and a BEGIN TRANSACTION for every operation sent to the server. In addition to the large amount of disk I/O required to support Autocommit mode, a performance penalty is paid for up to three network requests for every statement issued by an application.

Although using transactions can help application performance, do not take this tip too far. Leaving transactions active can reduce throughput by holding locks on rows for longer than necessary, preventing other users from accessing the rows. Commit transactions in intervals that allow maximum concurrency.

Choosing the right transaction model

Many systems support distributed transactions; that is, transactions that span multiple connections. Distributed transactions are at least four times slower than normal transactions due to the logging and network round trips necessary to communicate between all the components involved in the distributed transaction (the JDBC driver, transaction monitor, and DBMS). Unless distributed transactions are required, avoid using them. Instead, use local transactions when possible. Many Java application servers provide a default transaction behavior that uses distributed transactions.

For the best system performance, design the application to run using a single Connection object.

Using updateXXX methods

Although programmatic updates do not apply to all types of applications, developers should attempt to use programmatic updates and deletes. Using the updateXXX methods of the ResultSet object allows the developer to update data without building a complex SQL statement. Instead, the developer simply supplies the column in the result set that is to be updated and the data that is to be changed. Then, before moving the cursor from the row in the result set, the updateRow() method must be called to update the database as well.

In the following code fragment, the value of the Age column of the ResultSet object rs is returned using the getInt() method, and the updateInt() method is used to update the column with an int value of 25. The updateRow() method is called to update the row in the database with the modified value.

```
int n = rs.getInt("Age");
// n contains value of Age column in the resultset rs
...
rs.updateInt("Age", 25);
rs.updateRow();
```

In addition to making the application more easily maintainable, programmatic updates usually result in improved performance. Because the database server is already positioned on the row for the Select statement in process, performance-expensive operations to locate the row that needs to be changed are unnecessary. If the row must be located, the server usually has an internal pointer to the row available (for example, ROWID).

Using getBestRowIdentifier

Use getBestRowIdentifier() to determine the optimal set of columns to use in the Where clause for updating data. Pseudo-columns often provide the fastest access to the data, and these columns can only be determined by using getBestRowIdentifier().

Some applications cannot be designed to take advantage of positioned updates and deletes. Some applications formulate the Where clause by calling getPrimaryKeys() to use all searchable result columns or by calling getIndexInfo() to find columns that may be part of a unique index. These methods usually work, but can result in fairly complex queries.

Consider the following example:

```
ResultSet WSrs = WSs.executeQuery
     ("SELECT first_name, last_name, ssn, address, city, state, zip FROM emp");
// fetchdata
...
WSs.executeQuery (
   "UPDATE emp SET address = ?
    WHERE first_name = ? AND last_name = ? AND ssn = ?
    AND address = ? AND city = ? AND state = ? AND zip = ?");
// fairly complex query
```

Applications should call getBestRowIdentifier() to return the optimal set of columns (possibly a pseudo-column) that identifies a specific record. Many databases support special columns that are not explicitly defined by the user in the table definition, but are "hidden" columns of every table (for example, ROWID and TID). These pseudo-columns generally provide the fastest access to the data because they typically are pointers to the exact location of the record. Because pseudo-columns are not part of the explicit table definition, they are not returned from getColumns(). To determine if pseudo-columns exist, call getBestRowIdentifier().

Consider the previous example again:

```
...
ResultSet WSrowid = getBestRowIdentifier()
   (... "emp", ...);
...
WSs.executeUpdate("UPDATE EMP SET ADDRESS = ? WHERE ROWID = ?");
// fastest access to the data!
```

If your data source does not contain special pseudo-columns, the result set of getBestRowIdentifier() consists of the columns of the most optimal unique index on the specified table (if a unique index exists). Therefore, your application does not need to call getIndexInfo() to find the smallest unique index.