SYBASE®

Programmers Reference

jConnect™ for JDBC™

7.0

DOCUMENT ID: DC39001-01-0700-01

LAST REVISED: October 2009

Contents

About This Book

The *Sybase jConnect for JDBC Programmers Reference* describes the jConnect™ for JDBC™ product and explains how to use it to access data stored in relational database management systems.

Audience

This manual is for database-application programmers who are familiar with the Java programming language, JDBC, and Transact-SQL®, the Sybase® version of Structured Query Language (SQL).

How to use this book

The information in this book is organized as follows:

- Chapter 1, "Introduction," describes jConnect for JDBC concepts and components.
- Chapter 2, "Programming Information," describes jConnect for JDBC programming requirements.
- Chapter 3, "Security," describes the security mechanisms that you can use with jConnect.
- Chapter 4, "Troubleshooting," describes solutions and workarounds for problems you might encounter when using jConnect.
- Chapter 5, "Performance and Tuning," describes how you can enhance the performance of an application using jConnect.
- Chapter 6, "Migrating jConnect Applications," describes how you can migrate your application to jConnect 7.x.
- Chapter 7, "Web Server Gateways," contains information about Web server gateways and explains how you can use these gateways with jConnect.
- Chapter A, "SQL Exception and Warning Messages," lists the SQL exception and warning messages that you may encounter when using jConnect.
- Chapter B, "jConnect Sample Programs," serves as a guide to the jConnect sample programs.

Related Documents

See these books for more information:

- The *Sybase jConnect for JDBC Release Bulletin* contains important last-minute information about jConnect.
- The *Software Developer's Kit Release Bulletin* for your platform contains important last-minute information about Software Developer's Kit (SDK).
- The *Software Developer's Kit and Open Server Installation Guide* contains information about installing SDK and its jConnect for JDBC component.
- The *Adaptive Server Enterprise Installation Guide* contains information about installing Adaptive Server.
- The *Adaptive Server Enterprise Release Bulletin* for your platform contains information about known problems and recent updates to the Adaptive Server® Enterprise.
- The javadoc documentation of jConnect extensions to JDBC. The Java Development Kit (JDK) from Java Software contains a javadoc script for extracting comments from source-code files. This script has been used to extract documentation of jConnect packages, classes, and methods from jConnect source files. When you install jConnect using the full installation or javadocs option, the javadoc information is placed in the *javadocs* directory *Installation_directory/docs/en/javadocs.*

Other sources of information

Use the Sybase Getting Started CD, the SyBooks™ CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

  Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.

  Refer to the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.

- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://www.sybase.com/support/manuals/.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ Finding the latest information on product certifications

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click Partner Certification Report.
3. In the Partner Certification Report filter select a product, platform, and timeframe and then click Go.
4. Click a Partner Certification Report title to display the report.

❖ Finding the latest information on component certifications

1. Point your Web browser to Availability and Certification Reports at http://certification.sybase.com/.
2. Either select the product family and product under Search by Base Product; or select the platform and product under Search by Platform.
3. Select Search to display the availability and certification report for the selection.

❖ Creating a personalized view of the Sybase Web site (including support pages)

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ Finding the latest information on EBFs and software maintenance

1. Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.
2. Select EBFs/Maintenance. If prompted, enter your MySybase user name and password.
3. Select a product.
4. Specify a time frame and click Go. A list of EBF/Maintenance releases is displayed.

   Padlock icons indicate that you do not have download authorization for certain EBF/Maintenance releases because you are not registered as a Technical Support Contact. If you have not registered, but have valid information provided by your Sybase representative or through your support contract, click Edit Roles to add the “Technical Support Contact” role to your MySybase profile.
5. Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Conventions

Table 1: Syntax conventions

Key	Definition
command	Command names, command option names, utility names, utility flags, and other keywords are in sans serif font.
variable	Variables, or words that stand for values that you fill in, are in *italics*.
{ }	Curly braces indicate that you choose at least one of the enclosed options. Do not include the braces in the command.
[ ]	Brackets mean choosing one or more of the enclosed items is optional. Do not include the braces in the command.
( )	Parentheses are to be typed as part of the command.
|	The vertical bar means you can select only one of the options shown.
,	The comma means you can choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.

Accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

jConnect for JDBC and the HTML documentation have been tested for compliance with U.S. government Section 508 Accessibility requirements. Documents that comply with Section 508 generally also meet non-U.S. accessibility guidelines, such as the World Wide Web Consortium (W3C) guidelines for Web sites.

The online help for this product is also provided in HTM, which you can navigate using a screen reader.

Note You might need to configure your accessibility tool for optimal use. Some screen readers pronounce text based on its case; for example, they pronounce ALL UPPERCASE TEXT as initials, and MixedCase Text as words. You might find it helpful to configure your tool to announce syntax conventions. Consult the documentation for your tool.

For information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1

Introduction

This chapter introduces you to jConnect for JDBC and describes its concepts and components.

What is JDBC?

Java Database Connectivity (JDBC), from the Java Software Division of Sun MicroSystems, Inc., is a specification for an application program interface (API) that allows Java applications to access multiple database management systems using Structured Query Language (SQL). The JDBC Driver Manager handles multiple drivers that connect to different databases.

A set of interfaces is included in the standard JDBC API and the JDBC Standard Extension API so you can open connections to databases, execute SQL commands, and process results. The interfaces are described in Table 1-1.

Table 1-1: JDBC interfaces

Interface	Description
java.sql.Driver	Locates the driver for a database URL
java.sql.Connection	Used to connect to a specific database
java.sql.Statement	Executes SQL statements
java.sql.PreparedStatement	Handles SQL statements with parameters
java.sql.CallableStatement	Handles database stored procedure calls
java.sql.ResultSet	Gets the results of SQL statements
java.sql.DatabaseMetaData	Used to access information about a connection to a database.
java.sql.ResultSetMetaData	Used to access information describing the attributes of a ResultSet.
javax.sql.Rowset	Handles JDBC RowSet implementations.
javax.sql.DataSource	Handles connection to a data source.
javax.sql.ConnectionPoolData Source	Handles connection pools.

Each relational database management system requires a driver to implement these interfaces. There are four types of JDBC drivers:

- *Type 1 JDBC-ODBC bridge* – translates JDBC calls into ODBC calls and passes them to an ODBC driver. Some ODBC software must reside on the client machine. Some client database code may also reside on the client machine.
- *Type 2 native-API partly-Java driver* – converts JDBC calls into database-specific calls. This driver, which communicates directly with the database server, also requires some binary code on the client machine.
- *Type 3 net-protocol all-Java driver* – communicates to a middle-tier server using a DBMS-independent net protocol. A middle-tier gateway then converts the request to a vendor-specific protocol.
- *Type 4 native-protocol all-Java driver* – converts JDBC calls to the vendor-specific DBMS protocol, allowing client applications direct communication with the database server.

For more information about JDBC and its specification, see the Sun Developer Network for Java at http://java.sun.com.

What is jConnect?

jConnect is the Sybase high-performance JDBC driver. jConnect is both:

- A net-protocol/all-Java driver within a three-tier environment, and
- A native-protocol/all-Java driver within a two-tier environment.

The protocol used by jConnect is TDS 5.0 (Tabular Data Stream™, version 5), the native protocol for Adaptive Server Enterprise and Open Server™ applications. jConnect implements the JDBC standard to provide optimal connectivity to the complete family of Sybase products, allowing access to over 25 enterprise and legacy systems, including:

- Adaptive Server Enterprise
- SQL Anywhere®
- Sybase® IQ
- Replication Server®
- DirectConnect™

In addition, jConnect for JDBC can access Oracle, AS/400, and other data sources using Sybase DirectConnect.

In some instances, the jConnect implementation of JDBC deviates from the JDBC specifications. For more information, see “Restrictions on and interpretations of JDBC standards” on page 102.

CHAPTER 2 Programming Information

This chapter describes the basic components and programming requirements that comprise jConnect for JDBC. It explains how to invoke the jConnect driver, set connection properties, and connect to a database server. It also contains information about using jConnect features.

The following topics are included in this chapter:

For information about JDBC programming, go to the resource page for Java developers at the Sun Developer Network at http://java.sun.com.

Setting up jConnect

This section describes the tasks you need to perform before you use jConnect.

Setting the jConnect version

The jConnect version property JCONNECT_VERSION determines the driver's behavior and the features activated. For example, Adaptive Server 15.5 supports both jConnect 6.05 and 7.0, however, these two versions differ in their handling of datetime and time data. When connecting to Adaptive Server 15.5, jConnect 7.0, which supports microsecond granularity for time data, uses bigdatetime or bigtime even if the target Adaptive Server columns are defined as datetime or time. jConnect 6.05, however, does not support microsecond granularity and always transfers datetime or time data when connecting to Adaptive Server 15.5.

You can set the jConnect version by using either the SybDriver.setVersion method or the JCONNECT_VERSION connection property.

Using SybDriver. setVersion

The setVersion method affects the jConnect default behavior for *all* connections created by the SybDriver object. You can call setVersion multiple times to change the version setting. New connections inherit the behavior associated with the version setting at the time the connection is made. Changing the version setting during a session does not affect current connections. You can use the com.sybase.jdbcx.SybDriver.VERSION_LATEST constant to ensure that you are always requesting the highest version value possible for the jConnect driver you are using. However, by setting the version to com.sybase.jdbcx.SybDriver.VERSION_LATEST, you may see behavior changes if you replace your current jConnect driver with a newer one.

This code sample shows how to load the jConnect driver and set its version:

```
import java.sql.DriverManager;
import com.sybase.jdbcx.SybDriver;
SybDriver sybDriver = (SybDriver)
   Class.forName("com.sybase.jdbc4.jdbc.SybDriver")
   .newInstance();
sybDriver.setVersion(com.sybase.jdbcx.SybDriver.
   VERSION_7);
DriverManager.registerDriver(sybDriver);
```

Using JCONNECT_ VERSION

You can use the JCONNECT_VERSION connection property to override the SybDriver version setting and specify a different version setting for a specific connection. Table 2-1 lists the valid JCONNECT_VERSION values and the jConnect characteristics associated with these values.

Table 2-1: jConnect version settings and their features

JCONNECT_ VERSION	Features
"7.0"	jConnect 7.0 behaves in the same way as jConnect 6.05, except that in this version: • jConnect requests support for the bigdatetime and bigtime SQL datatypes from the server. Versions of Adaptive Server earlier than 15.5 ignore this request. See "Microsecond granularity for time data" on page 78. • jConnect supports JDBC 4.0. See "JDBC 4.0 specifications support" on page 90 and "Restrictions on and interpretations of JDBC standards" on page 102. • The valid values of ENABLE_BULK_LOAD are False, True, ARRAY_INSERT_WITH_MIXED_STATEMENTS, ARRAY_INSERT, and BCP.
"6.05"	jConnect 6.05 behaves in the same way as jConnect 6.0, except that in this version: • jConnect supports computed columns, including their metadata information. • jConnect supports large identifiers. With large identifiers, you can use identifiers or object names with lengths of up to 255 bytes. The large identifier applies to most user-defined identifiers, including table name, column name, and index name. • jConnect supports JDBC 3.0. See "JDBC 3.0 specifications support" on page 91 and "Restrictions on and interpretations of JDBC standards" on page 102.
"6"	jConnect 6.0 behaves in the same way as jConnect 5.x, except that in this version: • jConnect requests support for the date and time SQL datatypes. Versions of Adaptive earlier than 12.5.1 ignore this request. See "Using date and time datatypes" on page 66. • jConnect requests support for the unichar and univarchar datatypes from the server. Versions of Adaptive Server earlier than 12.5.1 ignore this request. See "Using jConnect to pass Unicode data" on page 32. • jConnect requests support for wide tables from the server. Versions of Adaptive Server earlier than 12.5.1 ignore this request. "Using wide table support for Adaptive Server" on page 45. • The default value of DISABLE_UNICHAR_ SENDING is false.
"5"	jConnect 5.x behaves in the same way as jConnect 4.0.
"4"	jConnect 4.0 behaves in the same way as jConnect 3.0, except that in this version: • The default value of the LANGUAGE connection property is null. • The default behavior of Statement.cancel is to cancel only the Statement object on which it is invoked. This behavior is JDBC-compliant. Use CANCEL_ALL to set the behavior of Statement.cancel. • You can use JDBC 2.0 methods to store and retrieve Java objects as column data. See "Storing Java objects as column data in a table" on page 82.

JCONNECT_ VERSION	Features
"3"	jConnect 3.0 behaves in the same way as jConnect 2.0, except that in this version: • If the CHARSET connection property does not specify a character set, jConnect uses the default character set of the database. • The default value for CHARSET_CONVERTER is the CheckPureConverter class.
"2"	• The default value of the LANGUAGE connection property is us_english. • If the CHARSET connection property does not specify a character set, the default character set is iso_1. • The default value for CHARSET_CONVERTER is the TruncationConverter class, unless the CHARSET connection property specifies a multibyte or 8-bit character set, in which case the default CHARSET_CONVERTER is the CheckPureConverter class. See "jConnect character-set converters" on page 33. • The default behavior of Statement.cancel is to cancel the object it is invoked on and any other Statement objects that have begun to execute and are waiting for results. This behavior is *not* JDBC-compliant. Use CANCEL_ALL to set the behavior of Statement.cancel.

Invoking the jConnect driver

To register and invoke jConnect, add jConnect to the jdbc.drivers system property. At initialization, the DriverManager class attempts to load the drivers listed in jdbc.drivers. This is less efficient than calling Class.forName. You can list multiple drivers in this property, separated with a colon (:). The following code samples show how to add a driver to jdbc.drivers within a program:

```
Properties sysProps = System.getProperties();
String drivers = "com.sybase.jdbc4.jdbc.SybDriver";
String oldDrivers =
sysProps.getProperty("jdbc.drivers");
if (oldDrivers != null)
   drivers += ":" + oldDrivers;
 sysProps.put("jdbc.drivers", drivers.toString());
```

Note System.getProperties is not allowed for Java applets. Use the Class.forName method instead.

In Java 6 and JDBC 4, the instantiation of JDBC drivers has been simplified. You can use the Java system property jdbc.drivers to specify driver classes, for example:

```
java -Djdbc.drivers=com.sybase.jdbc4.jdbc.SybDriver UseDriver
```

There is no need for the UseDriver program to load the driver explicitly:

```
public class UseDriver
{
   public static void main(String[] args)
   {
      try {
         Connection conn = java.sql.DriverManager.getConnection
            ("jdbc:sybase:Tds:localhost:5000?USER=sa&PASSWORD=secret");
         // more code to use connection ...
      }
      catch (SQLException se){
         System.out.println("ERROR: SQLException "+se);
      }
   }
}
```

Configuring jConnect for J2EE servers

You can use the com.sybase.jdbc4.jdbc.SybConnectionPoolDataSource class to configure connection pools to an Adaptive Server server in application servers such as EAServer. The com.sybase.jdbc4.jdbc.SybConnectionPoolDataSource implementation of the javax.sql.ConnectionPoolDataSource interface provides getter and setter methods for every connection property.

You can also configure jConnect programmatically, for example:

```
private DataSource getDataSource ()
{
   SybConnectionPoolDataSource connectionPoolDataSource = new
      SybConnectionPoolDataSource();
   connectionPoolDataSource.setDatabaseName("pubs2");
   connectionPoolDataSource.setNetworkProtocol("Tds");
   connectionPoolDataSource.setServerName("localhost");
   connectionPoolDataSource.setPortNumber(5000);
   connectionPoolDataSource.setUser("sa");
   connectionPoolDataSource.setPassword(PASSWORD);
   return connectionPoolDataSource;
}
private void work () throws SQLException
{
   Connection conn = null;
   Statement stmt = null;
   DataSource ds = getDataSource();
   try {
      conn = ds.getConnection();
      stmt = conn.createStatement();
      // ...
   }
   finally {
      if (stmt != null) {
         try { stmt.close(); } catch (Exception ex) { /* ignore */ }
      }
      if (conn != null) {
         try { conn.close(); } catch (Exception ex) { /* ignore */ }
      }
   }
}
```

Establishing a connection

This section describes how to establish a connection to an Adaptive Server or SQL Anywhere database using jConnect.

Connection properties

Connection properties specify the information needed to log in to a server and define expected client and server behavior. Connection property names are not case sensitive.

Setting connection properties

Connection properties must be set before connecting to a server. You can set connection properties in two ways:

- Use the DriverManager.getConnection method in your application.

- Set the connection properties when you define the URL.

Note Driver connection properties set in the URL do not override any corresponding connection properties set in the application using the DriverManager.getConnection method.

Following is a sample code that uses the DriverManager.getConnection method. The sample programs provided with jConnect also contain examples of setting these properties.

```
Properties props = new Properties();
 props.put("user", "userid");
 props.put("password", "user_password");
/*
 * If the program is an applet that wants to access
 * a server that is not on the same host as the
 * web server, then it uses a proxy gateway.
 */
 props.put("proxy", "localhost:port");
/*
 * Make sure you set connection properties before
 * attempting to make a connection. You can also
 * set the properties in the URL.
 */
Connection con = DriverManager.getConnection
  ("jdbc:sybase:Tds:host:port", props);
```

Listing current connection settings

To list a driver's current connection settings, use Driver.getDriverPropertyInfo(String url, Properties props). This code returns an array of DriverPropertyInfo objects containing:

- Driver properties
- Current settings on which the driver properties are based
- The URL and properties passed in

List of jConnect connection properties

Table 2-2 lists the connection properties for jConnect and indicates their default values. These properties are not case-sensitive.

Table 2-2: Connection properties

Property	Description	Default value
ALTERNATE_ SERVER_NAME	Specifies the alternate server name used by the primary and secondary database in a mirrored SQL Anywhere environment. The primary and secondary database use the same alternate server name so that client applications can connect to the current primary server without knowing in advance which of the two servers is the primary server. The JDBC URL syntax is still `jdbc:sybase:Tds:<hostname>:<port#>/database?connection_property=value;`. However, when ALTERNATE_SERVER_NAME is set, jConnect ignores the values of the *hostname* and *port* variables. Instead, jConnect uses the SQL Anywhere UDP discovery protocol to determine the current primary server. For information about database mirroring, see the *SQL Anywhere Server - Database Administration*. **Note** You can also use ALTERNATE_SERVER_NAME with an SQL Anywhere that is not mirrored. However, you will always get the same host and port values from the singleton server.	Null
APPLICATIONNAME	Specifies an application name. This is a user-defined property. The server side can be programmed to interpret the value given to this property.	Null
BE_AS_JDBC_ COMPLIANT_AS_ POSSIBLE	Adjusts other properties to ensure that jConnect methods respond in a way that is as compliant as possible with the JDBC 3.0 standard. These properties are affected (and overridden) when this property is set to "true": • CANCEL_ALL (set to "false") • LANGUAGE CURSOR (set to “false”) • SELECT_OPENS_CURSOR (set to "true") • FAKE_METADATA (set to "true") • GET_BY_NAME_USES_COLUMN_LABEL (set to "false")	False

Property	Description	Default value
CACHE_COLUMN_ METADATA	If you repeatedly use PreparedStatement or CallableStatement objects that perform SELECT queries, setting CACHE_COLUMN_ METADATA to true can improve performance. When set to true, the statement remembers the ResultSet Metadata information associated with the SELECT query results from the first execution of the statement. On subsequent executions, the metadata is re-used without having to be reconstructed. This saves CPU time through the use of additional memory.	False
CANCEL_ALL	Specifies the behavior of the Statement.cancel method: • If CANCEL_ALL is false, invoking Statement.cancel cancels only the Statement object on which it is invoked. Thus, if stmtA is a Statement object, stmtA.cancel cancels the execution of the SQL statement contained in stmtA in the database, but no other statements are affected. stmtA is canceled whether it is in cache waiting to execute or has started to execute and is waiting for results. • If CANCEL_ALL is true, invoking Statement.cancel cancels not only the object on which it is invoked, but also any other Statement objects on the same connection that have executed and are waiting for results. The following example sets CANCEL_ALL to “false.” *props* is a Properties object for specifying connection properties: `props.put("CANCEL_ALL", "false");` **Note** To cancel the execution of all Statement objects on a connection, regardless of whether or not they have begun execution on the server, use the extension method SybConnection.cancel.	• True – for JCONNECT_ VERSION <= “3” • False – for JCONNECT_ VERSION >= “4”

Property	Description	Default value
CAPABILITY_TIME	Used only when JCONNECT_VERSION >= 6. When jConnect is connected to a server that supports the TIME datatype, and all parameters of type java.sql.Time or escape literals {t ...} are processed as TIME. Previous versions of jConnect treat such parameters as DATETIME and prepend '1970-01-01' to the java.sql.Time parameter. If the underlying datatype is datetime or smalldatetime the date part also gets stored in the database. In jConnect 6.0 or later, when TIME is processed, the server converts time to the underlying datatype and will prepend its own base year. This can result in incompatibilities between old and new data. If you are using datetime or smalldatetime datatypes for java.sql.Time, then for backward compatibility you should leave CAPABILITY_TIME as *false*. Leaving this property as *false* forces jConnect to process java.sql.Time parameters or escape literals {t ...} as DATETIME regardless of the server capability of handling TIME datatype. Setting this property to *true* will cause jConnect to process java.sql.Time parameters as TIME datatype when connected to Adaptive Server. Sybase recommends you leave this property as *false* if you are using smalldatetime or datetime columns to store time values.	False
CAPABILITY_ WIDETABLE	If you do not require JDBC ResultSetMetaData like Column name as a performance improvement, you can set this to “false.” The result is that less data is exchanged over the network and increases performance. Unless you are using EAServer, Sybase recommends that you use the default setting. See “Using wide table support for Adaptive Server” on page 45.	False
CHARSET	Specifies the character set for strings passed to the database. If the CHARSET value is null, jConnect uses the default character set of the server to send string data to the server. If you specify a CHARSET, the database must be able to handle characters in that format. If the database cannot do so, a message is generated indicating that character conversion cannot be properly completed. **Note** If you are using jConnect 6.05 or later and the DISABLE_UNICHAR_SENDING is set to false, jConnect detects when a client is trying to send characters to the server that cannot be represented in the character set that is being used for the connection. When that occurs, jConnect sends the character data to the server as unichar data, which allows clients to insert Unicode data into unichar/univarchar columns and parameters.	Null

Property	Description	Default value
CHARSET_ CONVERTER_CLASS	Specifies the character-set converter class you want jConnect to use. jConnect uses the version setting from SybDriver.setVersion, or the version passed in with the JCONNECT_VERSION property, to determine the default character-set converter class to use. See “Selecting a character-set converter” on page 34 for details.	Version-dependent. See Table 2-1 on page 7.
CLASS_LOADER	A property you set to a DynamicClassLoader object that you create. The DynamicClassLoader is used to load Java classes that are stored in the database but which are not in the CLASSPATH at application start-up time. See“Using dynamic class loading” on page 87 for more information.	Null
CONNECTION_ FAILOVER	Used with the Java Naming and Directory Interface (JNDI). See “CONNECTION_FAILOVER connection property” on page 29.	True
CRC	When this property is set to true, the update counts that are returned are cumulative counts that include updates directly affected by the statement executed and any triggers invoked as a result of the statement being executed.	false
DATABASE	Use this property to specify the database name for a connection when the connection information is obtained from a Sybase *interfaces* file. The URL of an *interfaces* file cannot supply the database name.	*null*
DEFAULT_QUERY_ TIMEOUT	When this connection property is set, it is used as the default query timeout for any statements created on this connection.	*0* (no timeout)
DISABLE_UNICHAR_ SENDING	When a client application sends unichar characters to the server (along with non-unichar characters), there is a slight performance hit for any character data sent to the database. This property defaults to false in jConnect 6.05 and later. Clients using older versions of jConnect who wish to send unichar data to the database must set this property to false. See “Using jConnect to pass Unicode data” on page 32.	Version-dependent
DISABLE_ UNPROCESSED_ PARAM_WARNINGS	Disables warnings. During results processing for a stored procedure, jConnect often reads return values other than row data. If you do not process the return value, jConnect raises a warning. To disable these warnings (which can help performance), set this property to "true."	False
DYNAMIC_PREPARE	Determines whether dynamic SQL prepared statements are precompiled in the database. See “DYNAMIC_PREPARE connection property” on page 147.	False

Property	Description	Default value
ENABLE_BULK_ LOAD	Specifies whether to use bulk load to insert rows to the database. Values: • False – disables bulk load. • True or ARRAYINSERT_WITH_MIXED_STATEMENTS – enables bulk load with row-level logging and allows your application to execute other statements during the bulk load operation. • ARRAYINSERT – enables bulk load with row-level logging, but your application cannot execute other statements during the bulk load operation. • BCP – enables bulk load with page-level logging; your application cannot execute other statements during the bulk load operation.	False
ENABLE_SERVER_ PACKETSIZE	Specifies if the connection packet size is set to the value suggested by the server. If set to true, the driver does not use PACKETSIZE connection property and the server is free to use any value between 512 and the maximum packet size. If set to false, the PACKETSIZE connection property is used.	True
ENCRYPT_ PASSWORD	Allows a secure login. When this property is set to true, both login and remote site passwords are encrypted before being sent to the server. Passwords are no longer sent in clear text. ENCRYPT_PASSWORD has precedence over RETRY_WITH_NO_ENCRYPTION. For more information about password encryption, see “Using password encryption” on page 79.	False
ESCAPE_ PROCESSING_ DEFAULT	Circumvents processing of JDBC function escapes in SQL statements. By default, jConnect parses all SQL statements submitted to the database for valid JDBC function escapes. If your application is not going to use JDBC function escapes in its SQL calls, you can set this connection property to “false” to avoid this processing. This can provide a slight performance benefit. Additionally, ESCAPE_PROCESSING_DEFAULT helps with backend servers such as Sybase IQ that use curly braces as part of the SQL syntax.	True
EXPIRESTRING	Contains the license expiration date. Expiration is set to Never except for evaluation copies of jConnect. This is a read-only property.	Never

Property	Description	Default value
FAKE_METADATA	Returns phony metadata. When you call the ResultSetMetaData methods getCatalogName, getSchemaName, and getTableName and this property is set to "true," the call returns empty strings ("") because the server does not supply useful metadata. When this property is set to "false," calling these methods throws a "Not Implemented" SQLException. **Note** If you have enabled wide tables and are using an Adaptive Server 12.5 or later, this property setting is ignored because the server supplies useful metadata.	False
GET_BY_NAME_ USES_COLUMN_ LABEL	Provides backward compatibility with versions of jConnect earlier than 6.0. With Adaptive Server version 12.5 and later, jConnect has access to more metadata than was previously available. Prior to version 12.5, column name and column alias meant the same thing. jConnect can now differentiate between the two when used with a 12.5 or later Adaptive Server with wide tables enabled. To preserve backward compatibility, set this property to "true." If you want calls to getByte, getInt, get* (String columnName) to look at the actual name for the column, set this property to "false."	True
GET_COLUMN_ LABEL_FOR_NAME	Maintains backward compatibility with jConnect 5.5 or earlier, where a call to ResultMetaData.getColumnName returns the column label rather than the column name. Values: • True – ResultMetaData.getColumnName returns column label • False – ResultMetaData.getColumnName returns column name	False
GSSMANAGER_ CLASS	Specifies a third-party implementation of the org.ietf.jgss.GSSManager class. This property can be set to a string or a GSSManager object. If the property is set to a string, the value should be the fully qualified class name of the third-party GSSManager implementation. If the property is set to an object, the object must extend the org.ietf.jgss.GSSManager class. See Chapter 3, "Security" for more information.	Null
HOSTNAME	Identifies the name of the current host.	None. The max length is 30 characters and, if exceeded, it is truncated to 30.

Property	Description	Default value
HOSTPROC	Identifies the application process on the host machine.	None
IGNORE_DONE_IN_ PROC	Determines that intermediate update results (as in stored procedures) are not returned, only the final result set.	False
IMPLICIT_CURSOR_ FETCH_SIZE	Use this property with the SELECT_OPENS_CURSOR property to force jConnect to open a read-only cursor on every select query that is sent to the database. The cursor has a fetch size of the value set in this property, unless overridden by the Statement.setFetchSize method.	*0*
INTERNAL_QUERY_ TIMEOUT	Use this property to set the query timeout that will be used by statements internally created and executed by jConnect. This may prevent application failures if internal commands do not complete in a reasonable time.	*0* (no timeout)
IS_CLOSED_TEST	Allows you to specify what query, if any, is sent to the database when Connection.isClosed is called. For additional information, see the "Using Connection.isClosed and IS_CLOSED_TEST" on page 103.	Null
J2EE_TCK_ COMPLIANT	When this property is set to true, the jConnect driver enables behavior that is compliant with the J2EE 1.4 technology compatibility kit (TCK) test suite, which causes some loss of performance. Therefore, Sybase recommends using the default value of false.	false
JCE_PROVIDER_ CLASS	Specifies the Java Cryptography Extension (JCE) provider used in RSA encryption algorithms.	The bundled JCE provider.
JCONNECT_VERSION	Sets version-specific characteristics. See "Using JCONNECT_ VERSION" on page 6.	7
LANGUAGE	Specifies the language in which messages from jConnect and the server appear. The setting must match a language in *syslanguages* because server messages are localized according to the language setting in your local environment. The languages supported are Chinese, U.S. English, French, German, Japanese, Korean, Polish, Portuguese, and Spanish.	Version dependent. See "Using JCONNECT_ VERSION" on page 6.
LANGUAGE_ CURSOR	Determines that jConnect uses "language cursors" instead of "protocol cursors." See "Cursor performance" on page 150.	False
LITERAL_PARAMS	When set to "true," any parameters set by the setXXX methods in the PreparedStatement interface are inserted literally into the SQL statement when it is executed. If set to "false," parameter markers are left in the SQL statement and the parameter values are sent to the server separately.	False
NEWPASSWORD	Specifies the new password used in password expiration handling.	Null

Property	**Description**	**Default value**
PACKETSIZE	Identifies the network packet size. If you are using Adaptive Server 15.0 or later, Sybase recommends that you do not set this property and allow jConnect and Adaptive Server to select the network packet size that is appropriate for your environment.	512
PASSWORD	Identifies the login password. Set automatically if using the getConnection(String, String, String) method, or explicitly if using getConnection(String, Props).	None
PRELOAD_JARS	Contains a comma-separated list of *.jar* file names that are associated with the CLASS_LOADER that you specify. These *.jar* files are loaded at connect time, and are available for use by any other connection using the same jConnect driver. See “Preloading .jar files” on page 90 for more information.	Null
PROMPT_FOR_ NEWPASSWORD	Specifies whether to perform a transparent password change or to prompt for the new password. Values: • True – prompts you to manually set the new password. • False – jConnect checks the value of NEWPASSWORD and, if it is not null, uses this value to replace the expired password.	False
PROTOCOL_ CAPTURE	Specifies a file for capturing TDS communication between an application and an Adaptive Server.	Null
PROXY	Specifies a gateway address. For the HTTP protocol, the URL is http://host:port. To use the HTTPS protocol that supports encryption, the URL is https://host:port/servlet_alias.	None
QUERY_TIMEOUT_ CANCELS_ALL	Forces jConnect to cancel all Statements on a Connection when a read timeout is encountered. This behavior can be used when a client has calls execute() and the timeout occurs because of a deadlock (for example, trying to read from a table that is currently being updated in another transaction). The default value is *false*.	False
REMOTEPWD	Contains remote server passwords for access through server-to-server remote procedure calls. See “Performing server-to-server remote procedure calls” on page 44.	None
REPEAT_READ	Determines whether the driver keeps copies of columns and output parameters so that columns can be read out of order or repeatedly. See “REPEAT_READ connection property” on page 142.	True

Property	Description	Default value
REQUEST_HA_ SESSION	Indicates whether the connecting client wants to begin a high availability (HA) failover session. See "Implementing high availability failover support" on page 39. You cannot reset the property once a connection has been made. If you want more flexibility for requesting failover sessions, code the client application to set REQUEST_HA_SESSION at runtime. **Note** Setting this property to "true" causes jConnect to attempt a failover login. If you do not set this connection property, a failover session does not start, even if the server is configured for failover.	False
REQUEST_ KERBEROS_SESSION	Determines whether jConnect uses Kerberos for authentication. If this property is set to "true," a value for the SERVICE_PRINCIPAL_NAME property must also be specified. You may also wish to provide a value for the GSSMANAGER_CLASS property. See Chapter 3, "Security," for more information.	False
RETRY_WITH_NO_ ENCRYPTION	Allows server to retry logging in using clear text passwords. When both ENCRYPT_PASSWORD and RETRY_WITH_NO_ENCRYPTION properties are set to true, jConnect first logs in using the encrypted password. If login fails, jConnect logs in using the clear text password. For more information about password encryption, see "Using password encryption" on page 79.	False
RMNAME	Sets the Resource Manager name when using distributed transactions (XA). This property overrides a Resource Manager name that may be set in an LDAP server entry. See "Distributed transaction management support" on page 101 for more information.	Null
SECONDARY_ SERVER_HOSTPORT	Sets the hostname and port for the secondary server when the client is using an HA failover session. The value for this property should be in the form of hostName:portNumber. This property is ignored unless you have also set REQUEST_HA_SESSION to "true." See "Implementing high availability failover support" on page 39 for more information.	Null

Property	Description	Default value
SELECT_OPENS_ CURSOR	Determines whether calls to Statement.executeQuery automatically generate a cursor when the query contains a FOR UPDATE clause. If you have previously called Statement.setFetchSize or Statement.setCursorName on the same statement, a setting of “true” for SELECT_OPENS_CURSOR has no effect. **Note** You may experience some performance degradation when SELECT_OPENS_CURSOR is set to “true.” See “Using cursors with result sets” on page 47 for more information on using cursors with jConnect.	False
SERIALIZE_ REQUESTS	Determines whether jConnect waits for responses from the server before sending additional requests.	False
SERVER_INITIATED_ TRANSACTIONS	Allows the server to control transactions. By default the property is set to true and jConnect lets the server start and control transactions by using the Transact-SQL command set chained on. If set to false, the transactions are started and controlled by jConnect by using the Transact-SQL command begin tran. Sybase recommends that you allow the server to control the transactions.	True
SERVICENAME	Indicates the name of a back-end database server that a DirectConnect gateway serves. Also used to indicate which database should be used upon connecting to SQL Anywhere.	None
SERVERTYPE	When connected to OpenSwitch set this property to “OSW.” This allows jConnect to send certain instructions to OpenSwitch that allows OpenSwitch to remember initial connection settings for example, isolation level, textsize, quoted identifier and autocommit when OpenSwitch redirects a connection to a different server instance.	None
SERVICE_ PRINCIPAL_NAME	Used when establishing a Kerberos connection to Adaptive Server. The value of this property should correspond both to the server entry in your Key Distribution Center (KDC) and to the server name under which your database is running. The value of the SERVICE_PRINCIPAL_NAME property is ignored if the REQUEST_KERBEROS_SESSION property is set to “false.” See Chapter 3, “Security,” for more information.	Null
SESSION_ID	A TDS session ID. When this property is set, jConnect assumes that an application is trying to resume communication on an existing TDS session held open by the TDS-tunnelling gateway. jConnect skips the login negotiations and forwards all requests from the application to the specified session ID.	Null

Property	**Description**	**Default value**
SESSION_TIMEOUT	Specifies the amount of time (in seconds) that an HTTP-tunnelled session (created using the jConnect TDS-tunnelling servlet) remains alive while idle. After the specified time, the connection is automatically closed. For more information about the TDS-tunnelling servlet, see "Using TDS tunnelling" on page 155.	Null
SQLINITSTRING	Defines a set of commands to be passed to the database server when a connection is opened. These must be SQL commands that can be executed using the Statement.executeUpdate method.	Null
STREAM_CACHE_ SIZE	Specifies the maximum size used to cache statement response streams.	Null (unlimited cache size)
SYBSOCKET_ FACTORY	Enables jConnect to use your custom socket implementation. Set SYBSOCKET_FACTORY either to: • The name of a class that implements com.sybase.jdbcx.SybSocketFactory; or • "DEFAULT," which instantiates a new java.net.Socket( ) Use this property to make an SSL connection to your database.	Null
TEXTSIZE	Allows you to set the text size. By default Adaptive Server and SQL Anywhere allow 32,627 bytes to be read from an image or text column. If you have the jConnect mda tables installed, jConnect changes that value to 2GB. However setting this value when connected to OpenSwitch allows the connection to remember the setting when OpenSwitch redirects a connection to a different server instance.	2GB
USE_METADATA	Creates and initializes a DatabaseMetaData object when you establish a connection. The DatabaseMetaData object is necessary to connect to a specified database. jConnect uses DatabaseMetaData for some features, including Distributed Transaction Management support (JTA/JTS) and dynamic class loading (DCL). If you receive error 010SJ, which indicates that your application requires metadata, install the stored procedures for returning metadata that come with jConnect. See "Installing Stored Procedures" in Chapter 3 of the *jConnect for JDBC Installation Guide*.	True
USER	Specifies the login ID. Set automatically if using the getConnection(String, String, String) method, or explicitly if using getConnection(String, Props).	None
VERSIONSTRING	Provides read-only version information for the JDBC driver.	jConnect driver version

Connecting to Adaptive Server

In your Java application, define a URL using the jConnect driver to connect to an Adaptive Server. The basic format of the URL is:

```
jdbc:sybase:Tds:host:port
```

where:

- jdbc:sybase identifies the driver.
- Tds is the Sybase communication protocol for Adaptive Server.
- *host:port* is the Adaptive Server host name and listening port. See *$SYBASE/interfaces* (UNIX) or *%SYBASE%\ini\sql.ini* (Windows) for the entry that your database or Open Server application uses. Obtain the *host:port* from the "query" entry.

You can connect to a specific database using this format:

```
jdbc:sybase:Tds:host:port/database
```

Note To connect to a specific database using SQL Anywhere or DirectConnect, use the SERVICENAME connection property to specify the database name instead of "/database."

Example

The following code creates a connection to an Adaptive Server on host "myserver" listening on port 3697:

```
SysProps.put("user","userid");
SysProps.put("password","user_password");
String url = "jdbc:sybase:Tds:myserver:3697";
Connection_con =
   DriverManager.getConnection(url,SysProps);
```

URL connection property parameters

You can specify the values for the jConnect driver connection properties when you define a URL.

Note Driver connection properties set in the URL do not override any corresponding connection properties set in the application using the DriverManager.getConnection method.

To set a connection property in the URL, append the property name and its value to the URL definition. Use this syntax:

```
jdbc:sybase:Tds:host:port/database?
   property_name=value
```

To set multiple connection properties, append each additional connection property and value, preceded by “&.” For example:

```
jdbc:sybase:Tds:myserver:1234/mydatabase?
   LITERAL_PARAMS=true&PACKETSIZE=512&HOSTNAME=myhost
```

If the value for one of the connection properties contains “&,” precede the “&” in the connection property value with a backslash (\). For example, if your host name is “a&bhost,” use this syntax:

```
jdbc:sybase:Tds:myserver:1234/mydatabase?
   LITERAL_PARAMS=true&PACKETSIZE=512&HOSTNAME=
   a\&bhost
```

Do not use quotes for connection property values, even if they are strings. For example, use:

```
HOSTNAME=myhost
```

not:

```
HOSTNAME="myhost"
```

Using the *sql.ini* and *interfaces* file directory services

You can use *sql.ini* file for Windows and the *interfaces* file for UNIX to provide server information for jConnect for JDBC. By using the *sql.ini* or *interfaces* file, enterprises can centralize the information about the services available in the enterprise networks including Adaptive Server information.

Use the connection string to identify the *sql.ini* or *interfaces* file. On jConnect for JDBC, you may only connect to a single Directory Services URL (DSURL).

Connection string for a single DSURL for jConnect

When connecting to a DSURL, you must specify the path to the *sql.ini* or *interfaces* file and the server name. Otherwise, jConnect will return an error.

This specifies the path to the *sql.ini* file:

```
String url = "jdbc:sybase:jndi:file://D:/syb1252/ini/mysql.ini?myaseISO1”
```

where:

- server name = myaseISO1
- *sql.ini* file path = *file://D:/syb1252/ini/sql.ini*

This specifies the path to the *interfaces* file:

```
String url = "jdbc:sybase:jndi:file:///work/sybase/interfaces?myase"
```

where:

- server name = myase
- *interfaces* file path = *file:///work/sybase/interfaces*

Format of the *sql.ini* and *interfaces* file for SSL

The following is the format for the *sql.ini* file for *SSL*:

```
[SYBSRV2]
master=nlwnsck,mango1,4100,ssl
query=nlwnsck,mango1,4100,ssl
query=nlwnsck,mango1,5000,ssl
```

The format for the interfaces file is:

```
sybsrv2
master tcp ether mango1 5000 ssl
query tcp ether mango1 4100 ssl
query tcp ether mango1 5000 ssl
```

Note jConnect supports multiple query entries under the same server name in the *sql.ini* or *interfaces* file. jConnect attempts to connect to values for host or port from the query entry in the sequence, as in the *sql.ini* or *interfaces* file. If jConnect finds a SSL in a query entry, it will require the application to be coded to handle SSL connections by specifying an application specific socket factory, or the connection may fail.

Connecting to a server using JNDI

In jConnect, you can use the Java Naming and Directory Interface (JNDI) to provide connection information, which offers:

- A centralized location where you can specify host names and ports for connecting to a server. You do not need to hard code a specific host and port number in an application.

- A centralized location where you can specify connection properties and a default database for all applications to use.
- The jConnect CONNECTION_FAILOVER property for handling unsuccessful connection attempts. When CONNECTION_FAILOVER is set to "true," jConnect attempts to connect to a sequence of host/port server addresses in the JNDI name space until one succeeds.

To use jConnect with JNDI, you need to make sure that certain information is available in any directory service that JNDI accesses and that required information is set in the javax.naming.Context class. This section covers the following topics:

- Connection URL for using JNDI
- Required directory service information
- CONNECTION_FAILOVER connection property
- Providing JNDI context information

Connection URL for using JNDI

To specify that jConnect should use JNDI to obtain connection information, place "jndi" as the URL protocol after "sybase":

```
jdbc:sybase:jndi:protocol-information-for-use-with-JNDI
```

Anything that follows "jndi" in the URL is handled through JNDI. For example, to use JNDI with the Lightweight Directory Access Protocol (LDAP), you might enter:

```
jdbc:sybase:jndi:ldap://LDAP_hostname:port_number/servername=
  Sybase11,o=MyCompany,c=US
```

This URL tells JNDI to obtain information from an LDAP server, gives the host name and port number of the LDAP server to use, and provides the name of a database server in an LDAP-specific form.

Required directory service information

When you use JNDI with jConnect, JNDI needs to return the following information for the target database server:

- A host name and port number to connect to
- The name of the database to use

- Any connection properties that individual applications are not allowed to set on their own

This information needs to be stored according to a fixed format in any directory service used for providing connection information. The required format consists of a numerical object identifier (OID), which identifies the type of information being provided (for example, the destination database), followed by the formatted information.

Note You can use the alias name to reference the attribute instead of the OID.

Table 2-3 shows the required formatting.

Table 2-3: Directory service information for JNDI

Attribute description	Alias	OID (object_id)
Interfaces entry replacement in LDAP directory services	sybaseServer	1.3.6.1.4.1.897.4.1.1
Collection point for sybaseServer LDAP attributes	sybaseServer	1.3.6.1.4.1.897.4.2
Version Attribute	sybaseVersion	1.3.6.1.4.1.897.4.2.1
Servername Attribute	sybaseServer	1.3.6.1.4.1.897.4.2.2
Service Attribute	sybaseService	1.3.6.1.4.1.897.4.2.3
Status Attribute	sybaseStatus	1.3.6.1.4.1.897.4.2.4
Address Attribute	sybaseAddress	1.3.6.1.4.1.897.4.2.5
Security Mechanism Attribute	sybaseSecurity	1.3.6.1.4.1.897.4.2.6
Retry Count Attribute	sybaseRetryCount	1.3.6.1.4.1.897.4.2.7
Loop Delay Attribute	sybaseRetryDelay	1.3.6.1.4.1.897.4.2.8
jConnect Connection Protocol	sybaseJconnectProtocol	1.3.6.1.4.1.897.4.2.9
jConnect Connection Property	sybaseJconnectProperty	1.3.6.1.4.1.897.4.2.10
Database Name	sybaseDatabasename	1.3.6.1.4.1.897.4.2.11
High Availability Failover Servername Attribute	sybaseHAservername	1.3.6.1.4.1.897.4.2.15
ResourceManager Name	sybaseResourceManager Name	1.3.6.1.4.1.897.4.2.16
ResourceManager Type	sybaseResourceManager Type	1.3.6.1.4.1.897.4.2.17
JDBCDataSource Interface	sybaseJdbcDataSource-Interface	1.3.6.1.4.1.897.4.2.18
ServerType	sybaseServerType	1.3.6.1.4.1.897.4.2.19

Note Attributes in italics are required.

The following examples show connection information entered for the database server "SYBASE11" under an LDAP directory service. Example 1 uses the attribute OID. Example 2 uses the attribute alias, which is not case sensitive. You can use either the OID or the alias.

Example 1

```
dn: servername=SYBASE11,o=MyCompany,c=US
 servername:SYBASE11
1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1266
1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1337
1.3.6.1.4.1.897.4.2.5:TCP#1#standby1 4444
1.3.6.1.4.1.897.4.2.10:REPEAT_READ=false&
    PACKETSIZE=1024
 1.3.6.1.4.1.897.4.2.10:CONNECTION_FAILOVER=true
 1.3.6.1.4.1.897.4.2.11:pubs2
 1.3.6.1.4.1.897.4.2.9:Tds
```

Example 2 `dn: servername=SYBASE11,o=MyCompany,c=US`

```
    servername:SYBASE11
   sybaseAddress:TCP#1#giotto 1266
   sybaseAddress:TCP#1#giotto 1337
   sybaseAddress:TCP#1#standby1 4444
   sybaseJconnectProperty:REPEAT_READ=false&
      PACKETSIZE=1024
   sybaseJconnectProperty:CONNECTION_FAILOVER=true
   sybaseDatabasename:pubs2
   sybaseJconnectProtocol:Tds
```

In these examples, SYBASE11 can be accessed through either port 1266 or port 1337 on host “giotto,” and it can be accessed through port 4444 on host “standby1.” Two connection properties, REPEAT_READ and PACKETSIZE, are set within one entry. The CONNECTION_FAILOVER connection property is set as a separate entry. Applications connecting to SYBASE11 are initially connected with the pubs2 database. You do not need to specify a connection protocol, but if you do, you must enter the attribute as “`Tds`”, not “`TDS`”.

CONNECTION_FAILOVER connection property

CONNECTION_FAILOVER is a Boolean-valued connection property you can use when jConnect uses JNDI to get connection information.

If CONNECTION_FAILOVER is set to True, jConnect makes multiple attempts to connect to a server. If one attempt to connect to a host and port number associated with a server fails, jConnect uses JNDI to get the next host and port number associated with the server and attempts to connect through them. Connection attempts proceed sequentially through all the hosts and ports associated with a server.

For example, if CONNECTION_FAILOVER is set to True, and a database server is associated with the following hosts and port numbers, as in the earlier LDAP example:

```
1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1266
1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1337
1.3.6.1.4.1.897.4.2.5:TCP#1#standby 4444
```

To get a connection to the server, jConnect tries to connect to the host “giotto” at port 1266. If this fails, jConnect tries port 1337 on “giotto.” If this fails, jConnect tries to connect to host “standby1” through port 4444.

The default for CONNECTION_FAILOVER is True.

If CONNECTION_FAILOVER is set to False, jConnect attempts to connect to an initial host and port number. If the attempt fails, jConnect throws a SQL exception and does not try again.

Providing JNDI context information

To use jConnect with JNDI, you should be familiar with the JNDI specification from Sun Microsystems at http://java.sun.com/products/jndi.

In particular, you need to make sure that required initialization properties are set in javax.naming.directory.DirContext when JNDI and jConnect are used together. These properties can be set either at the system level or at runtime.

Two key properties are:

- Context.INITIAL_CONTEXT_FACTORY

  This property takes the fully qualified class name of the initial context factory for JNDI to use. This determines the JNDI driver that is used with the URL specified in the Context.PROVIDER_URL property.

- Context.PROVIDER_URL

  This property takes the URL of the directory service that the driver (for example, the LDAP driver) is to access. The URL should be a string, such as “ldap://ldaphost:427”.

The following example shows how to set context properties at runtime and how to get a connection using JNDI and LDAP. In the example, the INITIAL_CONTEXT_FACTORY context property is set to invoke the Sun Microsystem implementation of an LDAP service provider. The PROVIDER_URL context property is set to the URL of an LDAP directory service located on the host “ldap_server1” at port 389.

```
Properties props = new Properties();

 /* We want to use LDAP, so INITIAL_CONTEXT_FACTORY is set to the
 * class name of an LDAP context factory. In this case, the
 * context factory is provided by Sun’s implementation of a
 * driver for LDAP directory service.
 */
 props.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.ldap.LdapCtxFactory");

 /* Now, we set PROVIDER_URL to the URL of the LDAP server that
 * is to provide directory information for the connection.
 */
 props.put(Context.PROVIDER_URL, "ldap://ldap_server1:389");

 /* Set up additional context properties, as needed. */
 props.put("user", "xyz");
 props.put("password", "123");

 /* get the connection */
 Connection con = DriverManager.getConnection
   ("jdbc:sybase:jndi:ldap://ldap_server1:389" +
   "/servername=Sybase11,o=MyCompany,c=US",props);
```

The connection string passed to getConnection contains LDAP-specific information, which the developer must provide.

When JNDI properties are set at runtime, as in the preceding example, jConnect passes them to JNDI to be used in initializing a server, as in the following jConnect code:

```
javax.naming.directory.DirContext ctx =
   new javax.naming.directory.InitialDirContext(props);
```

jConnect then obtains the connection information it needs from JNDI by invoking DirContext.getAtributes, as in the following example, where *ctx* is a DirContext object:

```
javax.naming.directory.Attributes attrs =
   ctx.getAttributes("ldap://ldap_server1:389/servername=" +
     "Sybase11", SYBASE_SERVER_ATTRIBUTES);
```

In the example, SYBASE_SERVER_ATTRIBUTES is an array of strings defined within jConnect. The array values are the OIDs for the required directory information listed in Table 2-3.

Handling internationalization and localization

This section discusses internationalization and localization issues relevant to jConnect.

Using jConnect to pass Unicode data

In Adaptive Server version 12.5 and later, database clients can take advantage of the unichar and univarchar datatypes. The two datatypes allow for the efficient storage and retrieval of Unicode data.

Quoting from the Unicode Standard, version 2.0:

"The Unicode Standard is a fixed-width, uniform encoding scheme for encoding characters and text. The repertoire of this international character code for information processing includes characters for the major scripts of the world, as well as technical symbols in common. The Unicode character encoding treats alphabetic characters, ideographic characters, and symbols identically, which means they can be used in any mixture and with equal facility. The Unicode Standard is modeled on the ASCII character set, but uses a 16-bit encoding to support full multilingual text."

This means that the user can designate database table columns to store Unicode data, regardless of the default character set of the server.

Note In Adaptive Server version 12.5 through 12.5.0.3, the server had to have a default character set of utf-8 in order to use the Unicode datatypes. However, in Adaptive Server 12.5.1 and later, database users can use unichar and univarchar without having to consider the default character set of the server.

When the server accepts unichar and univarchar data, jConnect behaves as follows:

- For all character data that a client wishes to send to the server—for example, using PreparedStatement.setString (int column, String value)—jConnect determines if the string can be converted to the default character set of the server.
- If jConnect determines that the characters cannot be converted to the character set of the server (for example, some characters cannot be represented in the character set of the server), it sends the data to the server encoded as unichar/univarchar data.

For example, if a client attempts to send a Unicode Japanese character to an Adaptive Server 12.5.1 that has iso_1 as the default character set, jConnect detects that the Japanese character cannot be converted to an iso_1 character. jConnect then sends the string as Unicode data.

There is a performance penalty when a client sends unichar/univarchar data to a server. This is because jConnect must perform character-to-byte conversion twice for all strings and characters that do not map directly to the default character set of the server.

If you are using a jConnect version that is earlier than 6.05 and you wish to use the unichar and univarchar datatypes, you must perform the following tasks:

1. Set the JCONNECT_VERSION = 6 or later. See "Using JCONNECT_ VERSION" on page 6 for more information.
2. You need to set the DISABLE_UNICHAR_SENDING connection property to false. Starting with jConnect 6.05 this property is set to false by default. See "Setting connection properties" on page 10 for more information.

Note For more information on support for unichar and univarchar datatypes, see the Adaptive Server Enterprise manuals.

jConnect character-set converters

jConnect uses special classes for all character-set conversions. By selecting a character-set converter class, you specify how jConnect handles single-byte and multibyte character-set conversions, and what performance impact the conversions have on your applications.

There are two character-set conversion classes. The conversion class that jConnect uses is based on the JCONNECT_VERSION, CHARSET, and CHARSET_CONVERTER_CLASS connection properties.

- The TruncationConverter class works only with single-byte character sets that use ASCII characters such as iso_1 and cp850. It does not work with multibyte character sets or single-byte character sets that use non-ASCII characters. The TruncationConverter class is the default converter when JCONNECT_VERSION is set to 2.

  Using the TruncationConverter class, jConnect 7 handles character sets in the same manner as jConnect version 2.2. The TruncationConverter class is the default converter when the JCONNECT_VERSION = 2.

- The PureConverter class is a pure Java, multibyte character-set converter. jConnect uses this class if the JCONNECT_VERSION = 4 or later. jConnect also uses this converter when JCONNECT_VERSION = 2 if it detects a character set specified in the CHARSET connection property that is not compatible with the TruncationConverter class.

  Although it enables multibyte character-set conversions, the PureConverter class may negatively impact jConnect driver performance. If driver performance is a concern, see “Improving character-set conversion performance” on page 35.

Selecting a character-set converter

jConnect uses the JCONNECT_VERSION to determine the default character-set converter class to use. For JCONNECT_VERSION = 2 or higher, the default is PureConverter the default is TruncationConverter. For JCONNECT_VERSION = 4 or higher, the default is PureConverter.

You can also set the CHARSET_CONVERTER_CLASS connection property to specify which character-set converter you want jConnect to use. This is useful if you want to use a character-set converter other than the default for your jConnect version.

For example, if you set JCONNECT_VERSION = 4 or later but want to use the TruncationConverter class rather than the multibyte PureConverter class, you can set CHARSET_CONVERTER_CLASS:

```
...
 props.put("CHARSET_CONVERTER_CLASS",
   "com.sybase.jdbc4.utils.TruncationConverter")
```

Setting the CHARSET connection property

You can specify the character set to use in your application by setting the CHARSET driver property. If you do not set the CHARSET property:

- For JCONNECT_VERSION = 2, jConnect uses iso_1 as the default character set.
- For JCONNECT_VERSION = 3, and later, jConnect uses the default character set of the database, and adjusts automatically to perform any necessary conversions on the client side.

- For jConnect versions starting with 6.05, if jConnect cannot successfully convert the user data to the negotiated charset, it will send unconverted unicode characters to the server, if the server supports the unicode characters, else it will throw an exception.

You can also use the -J *charset* command line option for the IsqlApp application to specify a character set.

To determine which character sets are installed on your Adaptive Server, issue the following SQL query on your server:

```
select name from syscharsets
 go
```

For the PureConverter class, if the designated CHARSET does not work with the client Java Virtual Machine (VM), the connection fails with a SQLException, indicating that you must set CHARSET to a character set that is supported by both Adaptive Server and the client.

When the TruncationConverter class is used, character truncation is applied regardless of whether the designated CHARSET is 7-bit ASCII or not. Therefore, if your application needs to handle non-ascii data (for instance any asian languages), you should not use TruncationConverter, as this causes data corruption.

Improving character-set conversion performance

If you use multibyte character sets and need to improve driver performance, you can use the SunIoConverter class provided with the jConnect samples. See “SunIoConverter character-set conversion” on page 143 for details.

In addition, you can use TruncationConverter to improve performance if your application deals with only 7-bit ASCII data.

Supported character sets

Table 2-4 lists the Sybase character sets that are supported by jConnect. The table also lists the corresponding JDK byte converter for each supported character set.

Although jConnect supports UCS-2, currently no Sybase databases or Open Servers support UCS-2.

Adaptive Server versions 12.5 and later support a version of Unicode known as the UTF-16 encoding.

Table 2-4: Supported Sybase character sets

SybCharset name	JDK byte converter
ascii_7	ASCII
big5	Big5
big5hk (see note)	Big5_HKSCS
cp037	Cp037
cp437	Cp437
cp500	Cp500
cp850	Cp850
cp852	Cp852
cp855	Cp855
cp857	Cp857
cp860	Cp860
cp863	Cp863
cp864	Cp864
cp866	Cp866
cp869	Cp869
cp874	Cp874
cp932	MS932
cp936	GBK
cp950	Cp950
cp1250	Cp1250
cp1251	Cp1251
cp1252	Cp1252
cp1253	Cp1253
cp1254	Cp1254
cp1255	Cp1255
cp1256	Cp1256
cp1257	Cp1257
cp1258	Cp1258
deckanji	EUC_JP
eucgb	EUC_CN
eucjis	EUC_JP
eucksc	EUC_KR
GB18030	GB18030

SybCharset name	JDK byte converter
ibm420	Cp420
ibm918	Cp918
iso_1	ISO8859_1
iso88592	ISO8859-2
is088595	ISO8859_5
iso88596	ISO8859_6
iso88597	ISO8859_7
iso88598	ISO8859_8
iso88599	ISO8859_9
iso15	ISO8859_15_FDIS
koi8	KOI8_R
mac	Macroman
mac_cyr	MacCyrillic
mac_ee	MacCentralEurope
macgreek	MacGreek
macturk	MacTurkish
sjis	MS932
tis620	MS874
utf8	UTF8

European currency symbol support

jConnect supports the use of the European currency symbol, or "*euro*," and its conversion to and from UCS-2 Unicode.

The *euro* has been added to the following Sybase character sets: cp1250, cp1251, cp1252, cp1253, cp1254, cp1255, cp1256, cp1257, cp1258, cp874, iso885915, and utf8.

To use the *euro* symbol:

- Use the PureConvertor or CheckPureConverter class, pure Java, multibyte character-set converter. See "jConnect character-set converters" on page 33 for more information.
- Verify that the new character sets are installed on the server.

  The *euro* symbol is supported on Adaptive Server and SQL Anywhere.
- Select the appropriate character set on the client. See "Setting the CHARSET connection property" on page 34 for more information.

Unsupported character sets

The following Sybase character sets are not supported in jConnect because no JDK byte converters are analogous to the Sybase character sets:

- cp1047
- euccns
- greek8
- roman8
- roman9
- turkish8

You can use these character sets with the TruncationConverter class as long as the application uses only the 7-bit ASCII subsets of these characters.

Working with databases

This section discusses database issues relevant to jConnect and includes these topics:

- Implementing high availability failover support
- Performing server-to-server remote procedure calls
- Using wide table support for Adaptive Server
- Accessing database metadata
- Using cursors with result sets
- Using Transact-SQL queries with COMPUTE clause
- Support for batch updates
- Updating a database from a result set of a stored procedure
- Working with datatypes

Implementing high availability failover support

jConnect supports the Adaptive Server failover feature.

Note Sybase failover in a high availability system is a different feature than connection failover. Sybase strongly recommends that you read this section *very carefully* if you want to use both.

Overview

Sybase failover allows you to configure two Adaptive Servers as companions. If the primary companion fails, the devices, databases, and connections for that server can be taken over by the secondary companion.

You can configure a high availability system either asymmetrically or symmetrically:

- An *asymmetric* configuration includes two Adaptive Servers that are physically located on different machines but are connected so that if one of the servers is brought down, the other assumes its workload. The secondary Adaptive Server acts as a “hot standby” and does not perform any work until failover occurs.
- A *symmetric* configuration also includes two Adaptive Servers running on separate machines. However, if failover occurs, either Adaptive Server can act as a primary or secondary companion for the other Adaptive Server. In this configuration, each Adaptive Server is fully functional with its own system devices, system databases, user databases, and user logins.

In either setup, the two machines are configured for dual access, which makes the disks visible and accessible to both machines.

You can enable failover in jConnect and connect a client application to an Adaptive Server configured for failover. If the primary server fails over to the secondary server, the client application also automatically switches to the second server and reestablishes network connections.

Note Refer to *Adaptive Server Enterprise Using Sybase Failover in High Availability Systems Manual for Adaptive Server* for more detailed information.

Requirements, dependencies, and restrictions

- You must have two Adaptive Servers configured for failover.
- Only changes that were committed to the database before failover are retained when the client fails over.
- You must set the REQUEST_HA_SESSION jConnect connection property to “true” (see “Connection properties” on page 10).
- jConnect event notification does not work when failover occurs. See “Using event notification” on page 70.
- Close all statements when they are no longer used. jConnect stores information on statements to enable failover. Unclosed statements result in memory leaks.

Implementing failover in jConnect

Implement failover support in jConnect using one of the following two methods:

- Use the two connection properties, REQUEST_HA_SESSION and SECONDARY_SERVER_HOSTPORT, and set the following:
  - Set REQUEST_HA_SESSION to "true."
  - Set the SECONDARY_SERVER_HOSTPORT to the host name and port number where your secondary server is listening. See “Connection properties” on page 10, and the 'SECONDARY_SERVER_HOSTPORT' connection property.
- Use JNDI to connect to the server. See “Connecting to a server using JNDI”. Include an entry for the primary server and a separate entry for the secondary server in the directory service information file required by JNDI. The primary server entry has an attribute (the HA OID) that refers to the entry for the secondary server.

  Using LDAP as the service provider for JNDI, there are three possible forms that this HA attribute can have:

  a *Relative Distinguished Name (RDN)* – this form assumes that the search base (typically provided by the java.naming.provider.url attribute) combined with the value of this attribute is enough to identify the secondary server. For example, assume the primary server is at “hostname:4200” and the secondary server is at “hostname:4202”:

```
dn: servername=haprimary, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4200
1.3.6.1.4.1.897.4.2.15: servername=hasecondary
objectclass: sybaseServer

dn: servername=hasecondary, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4202
objectclass: sybaseServer
```

b *Distinguished Name (DN)* – this form assumes that the value of the HA attribute uniquely identifies the secondary server, and may or may not duplicate values found in the search base. For example:

```
dn: servername=haprimary, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4200
1.3.6.1.4.1.897.4.2.15: servername=hasecondary,
   o=Sybase, c=US ou=Accounting
objectclass: sybaseServer

dn: servername=hasecondary, o=Sybase, c=US, ou=Accounting
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4202
objectclass: sybaseServer
```

Notice that `hasecondary` is located in a different branch of the tree (see the additional `ou=Accounting` qualifier).

c *Full LDAP URL* – this form assumes nothing about the search base. The HA attribute is expected to be a fully-qualified LDAP URL that is used to identify the secondary (it may even point to a different LDAP server). For example:

```
dn: servername=hafailover, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4200
1.3.6.1.4.1.897.4.2.15: ldap://ldapserver: 386/servername=secondary,
   o=Sybase, c=US ou=Accounting
objectclass: sybaseServer

dn: servername=secondary, o=Sybase, c=US, ou=Accounting
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4202
objectclass: sybaseServer
```

d In the directory service information file required by JNDI, set the REQUEST_HA_SESSION connection property to "true" to enable a failover session every time you make a connection.

Use the REQUEST_HA_SESSION connection property to indicate that the connecting client wants to begin a failover session with Adaptive Server that is configured for failover. Setting this property to true instructs jConnect to attempt a failover login. If you do not set this connection property, a failover session does not start, even if the server is configured correctly. The default value for REQUEST_HA_SESSION is false.

Set the connection property like any other connection property. You cannot reset the property once a connection has been made.

If you want more flexibility for requesting failover sessions, code the client application to set REQUEST_HA_SESSION at runtime.

The following example shows connection information entered for the database server "SYBASE11" under an LDAP directory service, where "tahiti" is the primary server, and "moorea" is the secondary companion server:

```
dn: servername=SYBASE11,o=MyCompany,c=US
1.3.6.1.4.1.897.4.2.5:TCP#1#tahiti 3456
1.3.6.1.4.1.897.4.2.10:REPEAT_READ=false&PACKETSIZE=1024
1.3.6.1.4.1.897.4.2.10:CONNECTION_FAILOVER=false
1.3.6.1.4.1.897.4.2.11:pubs2
1.3.6.1.4.1.897.4.2.9:Tds
1.3.6.1.4.1.897.4.2.15:servername=SECONDARY
1.3.6.1.4.1.897.4.2.10:REQUEST_HA_SESSION=true



dn:servername=SECONDARY, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1#moorea 6000
```

- Request a connection using JNDI and LDAP:
  - jConnect uses the directory of the LDAP server to determine the name and location of the primary and secondary servers:

```
/* get the connection */
Connection con = DriverManager.getConnection
   ("jdbc:sybase:jndi:ldap://ldap_server1:389" +
    "/servername=Sybase11,o=MyCompany,c=US",props);
```

or

  - Specify a searchbase:

```
props.put(Context.PROVIDER_URL,
   "ldap://ldap_server1:389/ o=MyCompany, c=US");
```

```
Connection con=DriverManager.getConnection
   ("jdbc:sybase:jndi:servername=Sybase11", props);
```

Logging in to the primary server

If an Adaptive Server is not configured for failover or cannot grant a failover session, the client cannot log in, and the following warning displays:

```
'The server denied your request to use the high-
availability feature.

Please reconfigure your database, or do not request a
high-availability session.'
```

Failing over to the secondary server

When failover occurs, the SQL exception JZ0F2 is thrown:

```
‘Sybase high-availability failover has occurred. The
current transaction is aborted, but the connection is
still usable. Retry your transaction.’
```

The client then automatically reconnects to the secondary database using JNDI.

Note that:

- The identity of the database to which the client was connected and any committed transactions are retained.
- Partially read result sets, cursors, and stored procedure invocations are lost.
- When failover occurs, your application may need to restart a procedure or go back to the last completed transaction or activity.

Failing back to the primary server

At some point, the client fails back from the secondary server to the primary server. When failback occurs is determined by the System Administrator who issues sp_failback on the secondary server. Afterward, the client can expect the same behavior and results on the primary server as documented in “Failing over to the secondary server” on page 43.

Performing server-to-server remote procedure calls

A Transact-SQL language command or stored procedure running on one server can execute a stored procedure located on another server. The server to which an application has connected logs in to the remote server, and executes a server-to-server remote procedure call.

An application can specify a “universal” password for server-to-server communication, that is, a password used in all server-to-server connections. Once the connection is open, the server uses this password to log in to any remote server. By default, jConnect uses the password of the current connection as the default password for server-to-server communications.

However, if the passwords are different on two servers for the same user, and that user is performing server-to-server remote procedure calls, the application must explicitly define passwords for each server it plans to use.

jConnect includes a property that enables you to set a universal “remote” password or different passwords on several servers. jConnect allows you to set and configure the property using the setRemotePassword method in the SybDriver class:

```
Properties connectionProps = new Properties();

public final void setRemotePassword(String serverName,
    String password, Properties connectionProps)
```

To use this method, the application must import the SybDriver class, then call the method:

```
import com.sybase.jdbcx.SybDriver;
SybDriver sybDriver = (SybDriver)
    Class.forName("com.sybase.jdbc4.jdbc.SybDriver").n
ewInstance();
sybDriver.setRemotePassword
    (serverName, password, connectionProps);
```

Note To set different remote passwords for various servers, repeat the preceding call for each server.

This call adds the given server name-password pair to the given Properties object, which can be passed by the application to DriverManager in DriverManager.getConnection (*server_url, props*).

If serverName is null, the universal password is set to password for subsequent connections to all servers except the ones specifically defined by previous calls to setRemotePassword.

When an application sets the REMOTEPWD property, jConnect no longer sets the default universal password.

Using wide table support for Adaptive Server

Adaptive Server offers limits and parameters that are larger than in previous versions of the database server. For example:

- Tables can contain 1024 columns.
- Varchar and varbinary columns can contain more than 255 bytes of data.
- You can send and retrieve up to 2048 parameters when invoking stored procedures or inserting data into tables.

To ensure that jConnect requests wide table support from the database, the default setting of JCONNECT_VERSION must be 6 or later.

Note jConnect continues to work with an Adaptive Server version 12.5 and later if you set JCONNECT_VERSION to earlier than 6. However, if you try selecting from a table that requires wide table support to fully retrieve the data, you may encounter unexpected errors or data truncation.

You can also set JCONNECT_VERSION to 6 or later when you access data from a Sybase server that does not support wide tables. In this case, the server simply ignores your request for wide table support.

In addition to the larger number of columns and parameters, wide table support provides extended result set metadata. For example, in versions of jConnect earlier than 6.0, the ResultSetMetaData methods getCatalogName, getSchemaName, and getTableName all returned “Not Implemented” SQLExceptions because that metadata was not supplied by the server. When you enable wide table support, the server now sends back this information, and the three methods return useful information.

Accessing database metadata

To support JDBC DatabaseMetaData methods, Sybase provides a set of stored procedures that jConnect can call for metadata about a database. These stored procedures must be installed on the server for the JDBC metadata methods to work.

If the stored procedures for providing metadata are not already installed in a Sybase server, you can install them using stored procedure scripts provided with jConnect:

- *sql_server12.sql* installs stored procedures on Adaptive Server databases of version 12.0.x.
- *sql_server12.5.sql* installs stored procedures on Adaptive Server databases of version 12.5.x.
- *sql_server15.0.sql* installs stored procedures for Adaptive Server 15.x or later.
- *sql_asa.sql* – installs stored procedures on the SQL Anywhere 9.x database
- *sql_asa10.sql* – installs stored procedures on the SQL Anywhere 10.x database
- *sql_asa11.sql* – installs stored procedures on the SQL Anywhere 11.x database

---

Note The most recent versions of these scripts are compatible with all versions of jConnect.

See the *Sybase jConnect for JDBC Installation Guide* and *Sybase jConnect for JDBC Release Bulletin* for complete instructions on installing stored procedures.

In addition, to use the metadata methods, you must set the USE_METADATA connection property to "true" (its default value) when you establish a connection.

You cannot get metadata about temporary tables in a database.

Note The DatabaseMetaData.getPrimaryKeys method finds primary keys declared in a table definition (CREATE TABLE) or with alter table (ALTER TABLE ADD CONSTRAINT). It does not find keys defined using sp_primarykey.

Server-side metadata installation

Metadata support can be implemented in either the client (ODBC, JDBC) or in the data source (server stored procedures). jConnect provides metadata support on the server, which results in the following benefits:

- Keeps jConnect small in size, which ensures that the driver can be quickly downloaded from the Internet.
- Gains runtime efficiency from preloaded stored procedures on the data source.
- Provides flexibility—jConnect can connect to a variety of databases.

Using cursors with result sets

jConnect implements many JDBC 2.0 cursor and update methods. These methods make it easier to use cursors and to update rows in a table based on values in a result set.

In JDBC 2.0, ResultSets are characterized by their type and their concurrency. The type and concurrency values are part of the java.sql.ResultSet interface and are described in its javadocs.

Table 2-5 identifies the characteristics of java.sql.ResultSet that are available in jConnect. When requested, jConnect opens server side scrollable cursors when the server is Adaptive Server 15.0 or later.

Table 2-5: java.sql.ResultSet options available in jConnect

	Type		
Concurrency	**TYPE_FORWARD_ ONLY**	**TYPE_SCROLL_ INSENSITIVE**	**TYPE_SCROLL_ SENSITIVE**
CONCUR_READ_ONLY	Supported	Supported	Not available
CONCUR_UPDATABLE	Supported	Not available	Not available

This section includes the following topics:

- Creating a cursor
- Using JDBC 1.x methods for positioned updates and deletes
- Using JDBC 2.0 methods for positioned updates and deletes
- Using a cursor with a PreparedStatement object
- Using TYPE_SCROLL_INSENSITIVE result sets in jConnect

Creating a cursor

There are two methods for creating a cursor using jConnect:

- SybStatement.setCursorName

  Use SybStatement.setCursorName, to explicitly assign the cursor a name. The signature for SybStatement.setCursorName is:

  ```
  void setCursorName(String name) throws SQLException;
  ```

- SybStatement.setFetchSize

  Use SybStatement.setFetchSize to create a cursor and specify the number of rows returned from the database in each fetch. The signature for SybStatement.setFetchSize is:

  ```
  void setFetchSize(int rows) throws SQLException;
  ```

  When you use setFetchSize to create a cursor, the jConnect driver names the cursor. To get the name of the cursor, use ResultSet.getCursorName.

Another way you can create cursors is to specify the kind of ResultSet you want returned by the statement, using the following JDBC method on the connection:

```
Statement createStatement(int resultSetType, int
resultSetConcurrency)throws SQL Exception
```

The type and concurrencies correspond to the types and concurrencies found on the ResultSet interface listed in Table 2-5. If you request an unsupported ResultSet, a SQL warning is chained to the connection. When the returned Statement is executed, you receive the kind of ResultSet that is most like the one you requested. See the JDBC specification for more details on the behavior of this method.

If you do not use createStatement, the default types of ResultSet are:

- If you call only Statement.executeQuery, then the ResultSet returned is a SybResultSet that is TYPE_FORWARD_ONLY and CONCUR_READ_ONLY.
- If you call setFetchSize or setCursorName, then the ResultSet returned from executeQuery is a SybCursorResultSet that is TYPE_FORWARD_ONLY and CONCUR_UPDATABLE.

To verify that the kind of ResultSet object is what you intended, use the following two ResultSet methods:

```
int getConcurrency() throws SQLException;
int getType() throws SQLException;
```

❖ **Creating and using a cursor**

1 Create the cursor using Statement.setCursorName or SybStatement.setFetchSize.

2 Invoke Statement.executeQuery to open the cursor for a statement and return a cursor result set.

3 Invoke ResultSet.next to fetch rows and position the cursor in the result set.

The following example uses each of the two methods for creating cursors and returning a result set. It also uses ResultSet.getCursorName to get the name of the cursor created by SybStatement.setFetchSize.

```
// With conn as a Connection object, create a
// Statement object and assign it a cursor using
// Statement.setCursorName().
Statement stmt = conn.createStatement();
stmt.setCursorName("author_cursor");

// Use the statement to execute a query and return
// a cursor result set.
ResultSet rs = stmt.executeQuery("SELECT au_id,
      au_lname, au_fname FROM authors
      WHERE city = 'Oakland'");
while(rs.next())
{
...
}

// Create a second statement object and use
// SybStatement.setFetchSize()to create a cursor
// that returns 10 rows at a time.
SybStatement syb_stmt = conn.createStatement();
syb_stmt.setFetchSize(10);

// Use the syb_stmt to execute a query and return
// a cursor result set.
SybCursorResultSet rs2 =
      (SybCursorResultSet)syb_stmt.executeQuery
      ("SELECT au_id, au_lname, au_fname FROM
authors
       WHERE city = 'Pinole'");
while(rs2.next())
{
...
}
```

```
// Get the name of the cursor created through the
// setFetchSize() method.
String cursor_name = rs2.getCursorName();
 ...
// For jConnect 6.0, create a third statement
// object using the new method on Connection,
// and obtain a SCROLL_INSENSITIVE ResultSet.
// Note: you no longer have to downcast the
// Statement or the ResultSet.
Statement stmt = conn.createStatement(
                 ResultSet.TYPE_SCROLL_INSENSITIVE,
                 ResultSet.CONCUR_READ_ONLY);
ResultSet rs3 = stmt.executeQuery
   ("SELECT ... [whatever]");
// Execute any of the JDBC 2.0 methods that
// are valid for read only ResultSets.
rs3.next();
rs3.previous();
rs3.relative(3);
rs3.afterLast();
...
```

Using JDBC 1.x methods for positioned updates and deletes

The following example shows how to use methods in JDBC 1.x to do a positioned update. The example creates two Statement objects, one for selecting rows into a cursor result set, and the other for updating the database from rows in the result set.

```
// Create two statement objects and create a cursor
// for the result set returned by the first
// statement, stmt1. Use stmt1 to execute a query
// and return a cursor result set.
Statement stmt1 = conn.createStatement();
Statement stmt2 = conn.createStatement();
stmt1.setCursorName("author_cursor");
ResultSet rs = stmt1.executeQuery("SELECT
    au_id,au_lname, au_fname
    FROM authors WHERE city = 'Oakland'
    FOR UPDATE OF au_lname");

// Get the name of the cursor created for stmt1 so
// that it can be used with stmt2.
String cursor = rs.getCursorName();
```

```
// Use stmt2 to update the database from the
// result set returned by stmt1.
String last_name = new String("Smith");
while(rs.next())
{
    if (rs.getString(1).equals("274-80-9391"))
     {
       stmt2.executeUpdate("UPDATE authors "+
       "SET au_lname = "+last_name +
       "WHERE CURRENT OF " + cursor);
    }
}
```

Deletions in a result set

The following example uses Statement object *stmt2*, from the preceding code, to perform a positioned deletion:

```
stmt2.executeUpdate("DELETE FROM authors
          WHERE CURRENT OF " + cursor);
```

Using JDBC 2.0 methods for positioned updates and deletes

This section discusses JDBC 2.0 methods for updating columns in the current cursor row and updating the database from the current cursor row in a result set. Each is followed by an example.

Updating columns in a result set

JDBC 2.0 specifies a number of methods for updating column values from a result set in memory, on the client. The updated values can then be used to perform an update, insert, or delete operation on the underlying database. All of these methods are implemented in the SybCursorResultSet class.

Examples of some of the JDBC 2.0 update methods available in jConnect are:

```
void updateAsciiStream(String columnName, java.io.InputStream x, int length)
   throws SQLException;
void updateBoolean(int columnIndex, boolean x) throws SQLException;
void updateFloat(int columnIndex, float x) throws SQLException;
void updateInt(String columnName, int x) throws SQLException;
void updateInt(int columnIndex, int x) throws SQLException;
void updateObject(String columnName, Object x) throws SQLException;
```

Methods for updating the database from a result set

JDBC 2.0 specifies two methods for updating or deleting rows in the database, based on the current values in a result set. These methods are simpler in form than Statement.executeUpdate in JDBC 1.x and do not require a cursor name. They are implemented in SybCursorResultSet:

```
void updateRow() throws SQLException;
void deleteRow() throws SQLException;
```

Note The concurrency of the result set must be CONCUR_UPDATABLE. Otherwise, the above methods raise an exception. For insertRow, all table columns that require non-null entries must be specified.

Methods provided on DatabaseMetaData dictate when these changes are visible.

Example

The following example creates a single Statement object that is used to return a cursor result set. For each row in the result set, column values are updated in memory and then the database is updated with the new column values for the row.

```
// Create a Statement object and set fetch size to
// 25. This creates a cursor for the Statement
// object Use the statement to return a cursor
// result set.
SybStatement syb_stmt =
(SybStatement)conn.createStatement();
syb_stmt.setFetchSize(25);
SybCursorResultSet syb_rs =
(SybCursorResultSet)syb_stmt.executeQuery(
    "SELECT * from T1 WHERE ...")

// Update each row in the result set according to
// code in the following while loop. jConnect
// fetches 25 rows at a time, until fewer than 25
// rows are left. Its last fetch takes any
// remaining rows.
while(syb_rs.next())
{
  // Update columns 2 and 3 of each row, where
// column 2 is a varchar in the database and
// column 3 is an integer.
  syb_rs.updateString(2, "xyz");
syb_rs.updateInt(3,100);
//Now, update the row in the database.
```

```
  syb_rs.updateRow();
}
// Create a Statement object using the
// JDBC 2.0 method implemented in jConnect 6.0
Statement stmt = conn.createStatement
(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
// In jConnect 6.0, downcasting to SybCursorResultSet is not
// necessary. Update each row in the ResultSet in the same
// manner as above
while (rs.next())
{
rs.updateString(2, "xyz");
rs.updateInt(3,100);
  rs.updateRow();
// Use the Statement to return an updatable ResultSet
ResultSet rs = stmt.executeQuery("SELECT * FROM T1 WHERE...");
}
```

Deleting a row from a ResultSet

To delete a row from a cursor result set, you can use SybCursorResultSet.deleteRow as follows:

```
while(syb_rs.next())
{
    int col3 = getInt(3);
    if (col3 >100)
    {
    syb_rs.deleteRow();
    }
}
```

Inserting a row into a ResultSet

The following example illustrates how to do inserts using the JDBC 2.0 API. There is no need to downcast to a SybCursorResultSet.

```
// prepare to insert
rs.moveToInsertRow();
// populate new row with column values
rs.updateString(1, "New entry for col 1");
rs.updateInt(2, 42);
// insert new row into db
rs.insertRow();
// return to current row in result set
rs.moveToCurrentRow();
```

Using a cursor with a PreparedStatement object

Once you create a PreparedStatement object, you can use it multiple times with the same or different values for its input parameters. If you use a cursor with a PreparedStatement object, you must close the cursor after each use and then reopen the cursor to use it again. A cursor is closed when you close its result set (ResultSet.close). It is opened when you execute its prepared statement (PreparedStatement.executeQuery).

The following example shows how to create a PreparedStatement object, assign it a cursor, and execute the PreparedStatement object twice, closing and then reopening the cursor.

```
// Create a prepared statement object with a
// parameterized query.
PreparedStatement prep_stmt =
conn.prepareStatement(
"SELECT au_id, au_lname, au_fname "+
"FROM authors WHERE city = ? "+
"FOR UPDATE OF au_lname");

//Create a cursor for the statement.
prep_stmt.setCursorName("author_cursor");

// Assign the parameter in the query a value.
// Execute the prepared statement to return a
// result set.
prep_stmt.setString(1, "Oakland");
ResultSet rs = prep_stmt.executeQuery();

//Do some processing on the result set.
while(rs.next())
{
    ...
}

// Close the result, which also closes the cursor.
rs.close();

// Execute the prepared statement again with a new
// parameter value.
prep_stmt.setString(1,"San Francisco");
rs = prep_stmt.executeQuery();
// reopens cursor
```

Using TYPE_SCROLL_INSENSITIVE result sets in jConnect

jConnect supports only TYPE_SCROLL_INSENSITIVE result sets.

jConnect uses the Tabular Data Stream (TDS)—the Sybase proprietary protocol—to communicate with Sybase database servers. Adaptive Server 15.0 or later supports TDS scrollable cursors. For servers that do not support TDS scrollable cursors, jConnect caches the row data on demand, on the client, on each call to ResultSet.next. However, when the end of the result set is reached, the entire result set is stored in the client memory. Because this may cause a performance strain, Sybase recommends that you use TYPE_SCROLL_INSENSITIVE result sets only with Adaptive Server 15.0 or when the result set is reasonably small.

Note When you use TYPE_SCROLL_INSENSITIVE ResultSets in jConnect, and the server does not support TDS scrollable cursors, you can only call the isLast method after the last row of the ResultSet has been read. Calling isLast before the last row is reached causes an UnimplementedOperationException to be thrown.

jConnect provides the ExtendResultSet in the *sample2* directory; this sample provides a limited TYPE_SCROLL_INSENSITIVE ResultSet using JDBC 1.0 interfaces.

This implementation uses standard JDBC 1.0 methods to produce a scroll-insensitive, read-only result set, that is, a static view of the underlying data that is not sensitive to changes made while the result set is open. ExtendedResultSet caches all of the ResultSet rows on the client. Be cautious when you use this class with large result sets.

The sample.ScrollableResultSet interface:

- Is an extension of JDBC 1.0 java.sql.ResultSet.
- Defines additional methods that have the same signatures as the JDBC 2.0 java.sql.ResultSet.
- Does *not* contain all of the JDBC 2.0 methods. The missing methods deal with modifying the ResultSet.

The methods from the JDBC 2.0 API are:

```
boolean previous() throws SQLException;
boolean absolute(int row) throws SQLException;
boolean relative(int rows) throws SQLException;
boolean first() throws SQLException;
boolean last() throws SQLException;
```

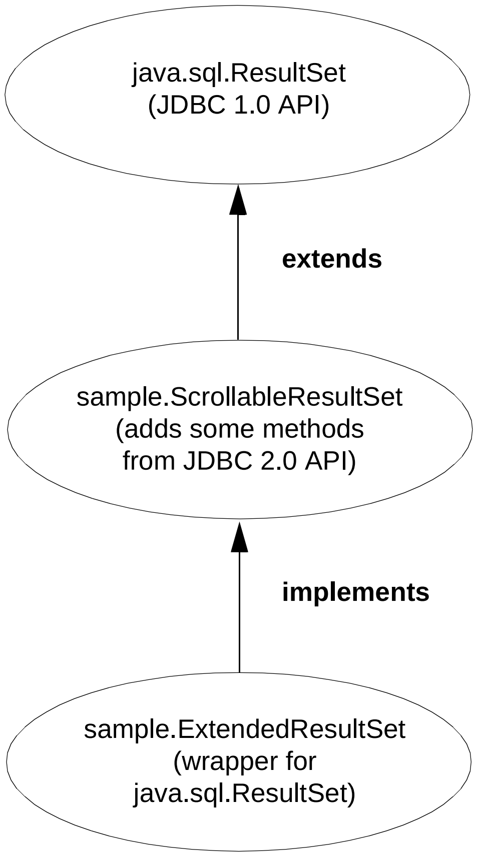
```
void beforeFirst() throws SQLException;
void afterLast() throws SQLException;
boolean isFirst() throws SQLException;
boolean isLast() throws SQLException;
boolean isBeforeFirst() throws SQLException;
boolean isAfterLast() throws SQLException;
int getFetchSize() throws SQLException;
void setFetchSize(int rows) throws SQLException;
int getFetchDirection() throws SQLException;
void setFetchDirection(int direction) throws
SQLException;
int getType() throws SQLException;
int getConcurrency() throws SQLException;
int getRow() throws SQLException;
```

To use the sample classes, create an ExtendedResultSet using any JDBC 1.0 java.sql.ResultSet. Below are the relevant pieces of code (assume a Java 1.1 environment):

```
// import the sample files
import sample.*;
//import the JDBC 1.0 classes
import java.sql.*;
// connect to some db using some driver;
// create a statement and a query;
// Get a reference to a JDBC 1.0 ResultSet
ResultSet rs = stmt.executeQuery(_query);
// Create a ScrollableResultSet with it
ScrollableResultSet srs = new ExtendedResultSet(rs);
// invoke methods from the JDBC 2.0 API
srs.beforeFirst();
// or invoke methods from the JDBC 1.0 API
if (srs.next())
  String column1 = srs.getString(1);
```

Figure 2-1 is a class diagram that shows the relationships between the sample classes and the JDBC API.

Figure 2-1: Class diagram



See the JDBC 2.0 API at http://java.sun.com/products/jdbc/jdbcse2.html for more details.

Using Transact-SQL queries with COMPUTE clause

jConnect for JDBC supports Transact-SQL queries that include a COMPUTE clause. A COMPUTE clause allows you to display detail and summary results in one select statement. The summary row appears following the detail rows of a specific group. For example:

```
select type, price, advance
   from titles
   order by type
   compute sum(price), sum(advance) by type

type          price      advance
------------  ---------  ----------
UNDECIDED     NULL       NULL

Compute Result:
------------------------ ------------------------
NULL                     NULL

type          price      advance
------------  ---------  ----------
business      2.99       10,125.00
business      11.95      5,000.00

business      19.99      5,000.00
business      19.99      5,000.00

Compute Result:
------------------------ ------------------------
54.92                    25,125.00

...
...

(24 rows affected)
```

When jConnect executes a select statement that includes a COMPUTE clause, jConnect returns multiple result sets to the client, the number of result sets depends on the number of unique groupings available. Each group contains one result set for the detail rows and one result set for the summary. The client must process all result sets to fully process the rows returned; if it does not, only the detail rows of the first group of data are included in the first result set returned.

For more information about the COMPUTE clause, see the *Adaptive Server Enterprise Transact-SQL Users Guide*. For more information about processing multiple result sets, see the JDBC API documentation found on the Sun Microsystems Web site.

Support for batch updates

Batch updates allow a Statement object to submit multiple update commands as one unit (batch) to an underlying database for processing together.

Note To use batch updates, you must install the latest metadata scripts provided in the *sp* directory under your jConnect installation directory.

See *BatchUpdates.java* in the *sample2* subdirectories for an example of using batch updates with Statement, PreparedStatement, and CallableStatement.

jConnect also supports dynamic PreparedStatements in batch.

Implementation notes

jConnect implements batch updates as specified in the JDBC 2.0 API, except as described here:

- If the JDBC 2.0 standard for implementing BatchUpdateException.getUpdateCounts is modified or relaxed in the future, jConnect continues to implement the original standard by having BatchUpdateException.getUpdateCounts return an int[ ] length of M < N, indicating that the first M statements in the batch succeeded, that the M+1 statement failed, and M+2..N statements were not executed. Here, “N” equals the total statements in the batch.
- To call stored procedures in batch (unchained) mode, you must create the stored procedure in unchained mode. For more information, see “Stored procedure executed in unchained transaction mode” on page 139.

- If an Adaptive Server encounters an error during batch execution, BatchUpdateException.getUpdateCounts returns only an int[ ] length of zero. The entire transaction is rolled back if an error is encountered, resulting in zero successful rows.

  Note The transaction is not rolled back if the error is a duplicate key row insert.

- In Adaptive Server, a duplicate key row insertion does not result in the termination and rollback of batch statements. The server continues processing the statements in the batch until you issue a cancel or the batch completes or encounters an error other than a duplicate key row insertion. Because jConnect sends a cancel to the server when it detects any exception (including duplicate key row insertion) during batch processing, it is impossible to determine exactly how much of the batch the server executed before receiving the cancel. Therefore, Sybase strongly recommends that in accordance with the JDBC specification, you should execute batches inside of transactions with autoCommit set to "false." This way, you can roll back your transactions and return the database to a known state before retrying the batch.
- Batch updates in databases that do not support batch updates: jConnect carries out batch updates in an executeUpdate loop even if your database does not support batch updates. This allows you to use the same batch code, regardless of the database to which you are pointing.

For details on batch updates see the JDBC API documentation at http://java.sun.com.

Updating a database from a result set of a stored procedure

jConnect includes update and delete methods that allow you to get a cursor on the result set returned by a stored procedure. You can then use the position of the cursor to update or delete rows in the underlying table that provided the result set. The methods are in SybCursorResultSet:

```
void updateRow(String tableName) throws SQLException;
void deleteRow(String tableName) throws SQLException;
```

The *tableName* parameter identifies the database table that provided the result set.

To get a cursor on the result set returned by a stored procedure, you need to use either SybCallableStatement.setCursorName or SybCallableStatement.setFetchSize before you execute the callable statement that contains the procedure. The following example shows how to create a cursor on the result set of a stored procedure, update values in the result set, and then update the underlying table using the SybCursorResultSet.update method:

```
// Create a CallableStatement object for executing the stored
// procedure.
CallableStatement sproc_stmt =
   conn.prepareCall("{call update_titles}");

// Set the number of rows to be returned from the database with
// each fetch. This creates a cursor on the result set.
(SybCallableStatement)sproc_stmt.setFetchSize(10);

//Execute the stored procedure and get a result set from it.
SybCursorResultSet sproc_result = (SybCursorResultSet)
   sproc_stmt.executeQuery();

// Move through the result set row by row, updating values in the
// cursor’s current row and updating the underlying titles table
// with the modified row values.
while(sproc_result.next())
{
   sproc_result.updateString(...);
   sproc_result.updateInt(...);
   ...
   sproc_result.updateRow(titles);
}
```

Working with datatypes

This section documents use of numeric, image, text, date, time, and char data.

Sending numeric data

The SybPreparedStatement extension supports the way Adaptive Server handles the NUMERIC datatype where precision (total digits) and scale (digits after the decimal) can be specified.

The corresponding datatype in Java—java.math.BigDecimal—is slightly different, and these differences can cause problems when jConnect applications use the setBigDecimal method to control values of an input/output parameter. Specifically, there are cases where the precision and scale of the parameter must precisely match that precision and scale of the corresponding SQL object, whether it is a stored procedure parameter or a column.

The SybPreparedStatement extension used with the following method gives jConnect applications more control over setBigDecimal:

```
public void setBigDecimal (int parameterIndex, BigDecimal X, int scale,
  int precision) throws SQLException
```

See the *SybPrepExtension.java* sample in the */sample2* subdirectories under your jConnect installation directory for more information.

Updating image data in the database

jConnect has a TextPointer class with sendData methods for updating an image column in an Adaptive Server or SQL Anywhere database. In earlier versions of jConnect, you had to send image data using the setBinaryStream method in java.sql.PreparedStatement. Now the TextPointer.sendData methods use java.io.InputStream and greatly improve performance when you send image data to an Adaptive Server database.

Warning! Using the TextPointer class with sendData() method may affect the application as TextPointer is not a standard JDBC form.

Sybase recommends you use PreparedStatement.setBinaryStream(int paramIndex, InputStream image), a standard JDBC form to send image data. However, setBinaryStream() may consume much more memory on procedure cache than the TextPointer class when large image data is handled.

Until a replacement for the TextPointer class is implemented, Sybase will continue supporting it.

To obtain instances of the TextPointer class, you can use either of two getTextPtr methods in SybResultSet:

```
public TextPointer getTextPtr(String columnName)
public TextPointer getTextPtr(int columnIndex)
```

Public methods in the TextPointer class

The com.sybase.jdbcx package contains the TextPointer class. Its public method interface is:

```
public void sendData(InputStream is, boolean log)
  throws SQLException

public void sendData(InputStream is, int length,
  boolean log) throws SQLException

public void sendData(InputStream is, int offset,
  int length, boolean log) throws SQLException

public void sendData(byte[] byteInput, int offset,
  int length, boolean log) throws SQLEXception
```

where:

- sendData(InputStream *is*, boolean *log)* updates an image column with data in the specified input stream.
- sendData(InputStream *is*, int *length*, boolean *log)* updates an image column with data in the specified input stream. *length* is the number of bytes being sent.
- sendData(InputStream *is*, int *offset*, int *length*, boolean *log)* updates an image column with data in the specified input stream, starting at the byte offset given in the *offset* parameter and continuing for the number of bytes specified in the *length* parameter.
- sendData(byte[ ] *byteInput*, int *offset*, int *length*, boolean *log)* updates a column with image data contained in the byte array specified in the *byteInput* parameter. The update starts at the byte offset given in the *offset* parameter and continues for the number of bytes specified in the *length* parameter.
- *log* is a parameter for each method that specifies whether image data is to be fully logged in the database transaction log. If the *log* parameter is set to "true," the entire binary image is written into the transaction log. If the *log* parameter is set to "false," the update is logged, but the image itself is not included in the log.

❖ Updating an *image* column with *TextPointer.sendData*

To update a column with image data:

1. Get a TextPointer object for the row and column that you want to update.
2. Use TextPointer.sendData to execute the update.

The next two sections illustrate these steps with an example. In the example, image data from the file *Anne_Ringer.gif* is sent to update the pic column of the au_pix table in the pubs2 database. The update is for the row with author ID 899-46-2035.

Getting a TextPointer object

text and image columns contain timestamp and page-location information that is separate from their text and image data. When data is selected from a text or image column, this extra information is "hidden" as part of the result set.

A TextPointer object for updating an image column requires this hidden information but does not need the image portion of the column data. To get this information, you need to select the column into a ResultSet object and then use SybResultSet.getTextPtr, which extracts text-pointer information, ignores image data, and creates a TextPointer object. See the following code for an example.

When a column contains a significant amount of image data, selecting the column for one or more rows and waiting to get all the data is likely to be inefficient, since the data is not used. To shortcut this process, use the set textsize command to minimize the amount of data returned in a packet. The following code example for getting a TextPointer object includes the use of set textsize for this purpose.

```
/*
 * Define a string for selecting pic column data for author ID
 * 899-46-2035.
 */
 String getColumnData = "select pic from au_pix where au_id = '899-46-2035'";

 /*
 * Use set textsize to return only a single byte of column data
 * to a Statement object. The packet with the column data will
 * contain the "hidden" information necessary for creating a
 * TextPointer object.
 */
 Statement stmt= connection.createStatement();
 stmt.executeUpdate("set textsize 1");

 /*
 * Select the column data into a ResultSet object--cast the
 * ResultSet to SybResultSet because the getTextPtr method is
 * in SybResultSet, which extends ResultSet.
 */
 SybResultSet rs = (SybResultSet)stmt.executeQuery(getColumnData);
```

```
/*
* Position the result set cursor on the returned column data
* and create the desired TextPointer object.
*/
rs.next();
TextPointer tp = rs.getTextPtr("pic");

/*
* Now, assuming we are only updating one row, and won’t need
* the minimum textsize set for the next return from the server,
* we reset textsize to its default value.
*/
stmt.executeUpdate("set textsize 0");
```

Executing the update with *TextPointer.sendData*

The following code uses the TextPointer object from the preceding section to update the pic column with image data in the file *Anne_Ringer.gif*.

```
/*
*First, define an input stream for the file.
*/
FileInputStream in = new FileInputStream("Anne_Ringer.gif");

/*
* Prepare to send the input stream without logging the image data
* in the transaction log.
*/
boolean log = false;

/*
* Send the image data in Anne_Ringer.gif to update the pic
* column for author ID 899-46-2035.
*/
tp.sendData(in, log);
```

See the *TextPointers.java* sample in the *sample2* subdirectories under your jConnect installation directory for more information.

Using *text* data

In earlier versions, jConnect used a TextPointer class with sendData methods for updating a text column in an Adaptive Server or SQL Anywhere database.

The TextPointer class has been deprecated, that is, it is no longer recommended and may cease to exist in a future version of Java.

If your data server is Adaptive Server or SQL Anywhere, use the standard JDBC form to send text data:

```
PreparedStatement.setAsciiStream(int paramIndex,
   InputStream text, int length)
```

or

```
PreparedStatement.setUnicodeStream(int paramIndex,
   InputStream text, int length)
```

or

```
PreparedStatement.setCharacterStream(int paramIndex,
   Reader reader, int length)
```

Using *date* and *time* datatypes

Adaptive Server supports the SQL datetime, smalldatetime, date, and time datatypes. date and time provide the following advantages:

- Date values can be between Jan. 1, 0001 and Dec. 31, 9999, exactly matching the allowable values in java.sql.Date.
- A direct mapping exists between java.sql.Date and the date datatype, as well as between java.sql.Time and the time datatype.

Implementation notes

- If you select from a table that contains a date or time column, and you have not enabled date/time support in jConnect (by setting the version), the server tries to convert the date or time to a datetime value before returning it. This can cause problems if the date to be returned is prior to 1/1/1753. In that case, a conversion error occurs, and the database informs you of the error.
- SQL Anywhere supports a date and time datatype, but the date and time datatypes are not yet directly compatible with those in Adaptive Server version 12.5.1 and later. Using jConnect, you should continue to use the datetime and smalldatetime datatypes when communicating with SQL Anywhere.
- The maximum value in a datetime column in SQL Anywhere is 1-1-7911 00:00:00.

- Using jConnect, you receive conversion errors if you attempt to insert dates prior to 1/1/1753 into datetime columns or parameters.
- Refer to the Adaptive Server manuals for more information on the date and time datatypes; of special note is the section on allowable implicit conversions.
- If you use getObject with an Adaptive Server date, time, or datetime column, the value returned is, respectively, a java.sql.Date, java.sql.Time, or java.sql.Timestamp datatype.

Using c*har/varchar/text* datatypes and *getByte*

Do not use rs.getByte on a char, univarchar, unichar, varchar, or text field unless the data is hex, octal, or decimal.

Implementing advanced features

This section describes how to use advanced jConnect features and contains the following topics:

- Using BCP insert
- Supported Adaptive Server Cluster Edition features
- Using event notification
- Handling error messages
- Microsecond granularity for time data
- Using password encryption
- Storing Java objects as column data in a table
- Using dynamic class loading
- JDBC 4.0 specifications support
- JDBC 3.0 specifications support
- JDBC 2.0 optional package extensions support

Using BCP insert

jConnect supports large insertions of rows to Adaptive Server version 12.5.2 and later using bulk load inserts. Although this feature does not require special configuration on the server, a larger page size, network packet size, and max memory size significantly improves performance. Depending on the client memory, use of larger batches also improves performance.

To enable this feature, set ENABLE_BULK_LOAD to True. When you use prepared statements and ENABLE_BULK_LOAD is True, jConnect uses the BULK routines to insert a batch of records to the Sybase databases.

BCP insert does not support:

- unsigned types, bigint, and unitext
- encrypted columns and computed columns

Supported Adaptive Server Cluster Edition features

jConnect supports the Adaptive Server Cluster Edition environment, where multiple Adaptive Servers connect to a shared set of disks and a high-speed private interconnection. This allows Adaptive Server to scale using multiple physical and logical hosts.

For more information about Cluster Edition, see the *Adaptive Server Enterprise Users Guide to Clusters*.

Login redirection

At any given time, some servers within a Cluster Edition environment are usually more loaded with work than others. When a client application attempts to connect to a busy server, the login redirection feature helps balance the load of the servers by allowing the server to redirect the client connection to less busy servers within the cluster. The login redirection occurs during the login sequence and the client application does not receive notification that it was redirected. Login redirection is enabled automatically when a client application connects to a server that supports this feature.

Note When a client application connects to a server that is configured to redirect clients, the login time may increase because the login process is restarted whenever a client connection is redirected to another server.

Connection migration

The connection migration feature allows a server in a Cluster Edition environment to dynamically distribute load, and seamlessly migrate an existing client connection and its context to another server within the cluster. This feature enables the Cluster Edition environment to achieve optimal resource utilization and decrease computing time. Because migration between servers is seamless, the connection migration feature also helps create a highly available, zero-downtime environment. Connection migration is enabled automatically when a client application connects to a server that supports this feature.

Note Command execution time may increase during server migration. Sybase recommends that you increase the command timeouts accordingly.

Connection failover

Connection failover allows a client application to switch to an alternate Adaptive Server if the primary server becomes unavailable due to an unplanned event, like power outage or a socket failure. In a cluster environment, client applications can fail over numerous times to multiple servers using dynamic failover addresses.

With high availability option enabled, the client application does not need to be configured to know the possible failover targets. Adaptive Server keeps the client updated with the best failover list based on cluster membership, logical cluster usage, and load distribution. During failover, the client refers to the ordered failover list while attempting to reconnect. If the driver successfully connects to a server, the driver internally updates the list of host values based on the list returned. Otherwise, the driver throws a connection failure exception.

Enabling connection failover

You can use the connection string to enable connection failover by setting REQUEST_HA_SESSION to true. For example:

```
URL="jdbc:sybase:Tds:server1:port1,server2:port2,...,
serverN:portN/mydb?REQUEST_HA_SESSION=true"
```

where server1:port1, server2:port2, ... , serverN:portN is the ordered failover list.

In establishing a connection, jConnect attempts to connect to the first host and port specified in the failover list. If unsuccessful, goes through the list until a connection is established or until the end of the list is reached.

Note The list of alternate servers specified in the connection string is used only during initial connection. After the connection is established with any available instance, and if the client supports high availability, the client receives an updated list of the best possible failover targets from the server. This new list overrides the specified list.

Using event notification

You can use the jConnect event notification feature to have your application notified when an Open Server procedure is executed.

To use this feature, you must use the SybConnection class, which extends the Connection interface. SybConnection contains a regWatch method for turning event notification on and a regNoWatch method for turning event notification off.

Your application must also implement the SybEventHandler interface. This interface contains one public method, void event(String proc_name, ResultSet params), which is called when the specified event occurs. The parameters of the event are passed to event, which tells the application how to respond.

To use event notification in your application, call SybConnection.regWatch( ) to register your application in the notification list of a registered procedure. Use this syntax:

SybConnection.regWatch(*proc_name*,*eventHdlr*,*option*)

where:

- *proc_name* is a string that is the name of the registered procedure that generates the notification.
- *eventHdler* is an instance of the SybEventHandler class that you implement.
- *option* is either NOTIFY_ONCE or NOTIFY_ALWAYS. Use NOTIFY_ONCE if you want the application to be notified only the first time a procedure executes. Use NOTIFY_ALWAYS if you want the application to be notified every time the procedure executes.

Whenever an event with the designated *proc_name* occurs on the Open Server, jConnect calls eventHdlr.event from a separate thread. The event parameters are passed to eventHdlr.event when it is executed. Because it is a separate thread, event notification does not block execution of the application.

If *proc_name* is not a registered procedure, or if Open Server cannot add the client to the notification list, the call to regWatch throws a SQL exception.

To turn off event notification, use this call:

```
SybConnection.regNoWatch(proc_name)
```

Warning! When you use Sybase event notification extensions, the application needs to call the close method on the connection to remove a child thread created by the first call to regWatch. Failing to do so may cause the Virtual Machine to hang when exiting the application.

Event notification example

The following example shows how to implement an event handler and then register an event with an instance of your event handler, once you have a connection:

```
public class MyEventHandler implements SybEventHandler
{
  // Declare fields and constructors, as needed.
  ...
  public MyEventHandler(String eventname)
  {
    ...
  }

  // Implement SybEventHandler.event.
  public void event(String eventName, ResultSet params)
  {
    try
    {
      // Check for error messages received prior to event
      // notification.
      SQLWarning sqlw = params.getWarnings();
      if sqlw != null
      {
        // process errors, if any
        ...
      }
      // process params as you would any result set with
      // one row.
      ResultSetMetaData rsmd = params.getMetaData();
      int numColumns = rsmd.getColumnCount();
      while (params.next())          // optional
      {
        for (int i = 1; i <= numColumns; i++)
        {
          System.out.println(rsmd.getColumnName(i) + " =
            " + params.getString(i));
        }
        // Take appropriate action on the event. For example,
        // perhaps notify application thread.
        ...
      }
    }
    catch (SQLException sqe)
    {
      // process errors, if any
      ...
    }
  }
}

public class MyProgram
{
  ...
  // Get a connection and register an event with an instance
  // of MyEventHandler.
  Connection conn = DriverManager.getConnection(...);
  MyEventHandler myHdlr = new  MyEventHandler("MY_EVENT");

  // Register your event handler.
  ((SybConnection)conn).regWatch("MY_EVENT", myHdlr,
    SybEventHandler.NOTIFY_ALWAYS);
  ...
 conn.regNoWatch("MY_EVENT");
  conn.close();
}
```

Handling error messages

jConnect provides two classes for returning Sybase-specific error information, SybSQLException and SybSQLWarning, as well as a SybMessageHandler interface that allows you to customize the way jConnect handles error messages received from the server.

Handling numeric errors returned as warnings

In Adaptive Server12.0 through 12.5, numeric errors are handled by default as severity 10. A severity-level 10 message is classified as a status information message, not as an error, and its content is transferred in a SQLWarning object. The following code excerpt illustrates this processing:

```
static void processWarnings(SQLWarning warning)
{
if (warning != null)
 {
  System.out.println ("\n -- Warning received -- \n");
 }//end if
  while (warning != null)
 {
  System.out.println ("Message: " +
  warning.getMessage());
  System.out.println("SQLState: " +
  warning.getSQLState());
  System.out.println ("ErrorCode: " +
  warning.getErrorCode());
  System.out.println ("---------------------------");
  warning = warning.getNextWarning();
 }//end while
}//end processWarnings
```

When a numeric error occurs, the ResultSet object returned contains no result set data, and the relevant information concerning the error must be obtained from the SQLWarning. Therefore, in a JDBC application, the code that checks for and processes a SQLWarning should not depend on there being a result set. For example, the following code checks for and processes SQLWarning data both inside and outside the result set processing while loop:

```
while (rs.next())
{
 String value = rs.getString(1);
 System.out.println ("Fetched value: " + value);

 // Check for SQLWarning on the result set.
```

```
     processWarnings (rs.getWarnings());

    }//end while

     // Check for SQLWarning on the result set.
     processWarnings (rs.getWarnings());
```

Here the code checks for SQLWarning even if there is no result set data (rs.next( ) is false). The following example is output for a program properly written to detect and report numeric errors. The error is a division by zero:

```
-- Warning received --

Message: Divide by zero occurred.
SQLState: 01012
ErrorCode: 3607
```

Retrieving Sybase-specific error information

jConnect provides an EedInfo interface that specifies methods for obtaining Sybase-specific error information. The EedInfo interface is implemented in SybSQLException and SybSQLWarning, which extend the SQLException and SQLWarning classes.

SybSQLException and SybSQLWarning contain the following methods:

- public ResultSet getEedParams, which returns a one-row result set containing any parameter values that accompany the error message.
- public int getStatus, which returns a "1" if there are parameter values, returns a "0" if there are no parameter values in the message.
- public int getLineNumber, which returns the line number of the stored procedure or query that caused the error message.
- public String getProcedureName, which returns the name of the procedure that caused the error message.
- public String getServerName, which returns the name of the server that generated the message.
- public int getSeverity, which returns the severity of the error message.
- public int getState, which returns information about the internal source of the error message in the server. (For use by Sybase Technical Support only.)
- public int getTranState, which returns one of the following transaction states:

- 0 The connection is currently in an extended transaction.
- 1 The previous transaction committed successfully.
- 3 The previous transaction aborted.

Some error messages can be SQLException or SQLWarning messages without being SybSQLException or SybSQLWarning messages. Your application should check the type of exception it is handling before it downcasts to SybSQLException or SybSQLWarning.

Customizing error-message handling

You can use the SybMessageHandler interface to customize the way jConnect handles error messages generated by the server. Implementing SybMessageHandler in your own class for handling error messages can provide the following benefits:

- "Universal" error handling

  Error-handling logic can be placed in your error-message handler, instead of being repeated throughout your application.

- "Universal" error logging

  Your error-message handler can contain the logic for handling all error logging.

- Remapping of error-message severity, based on application requirements

  Your error-message handler can contain logic for recognizing specific error messages, and downgrading or upgrading their severity based on application considerations rather than the severity rating of the server. For example, during a cleanup operation that deletes old rows, you might want to downgrade the severity of a message that a row does not exist. However, you may want to upgrade the severity in other circumstances.

Note Error-message handlers implementing the SybMessageHandler interface only receive server-generated messages. They do not handle messages generated by jConnect.

When jConnect receives an error message, it checks to see if a SybMessageHandler class has been registered for handling the message. If so, jConnect invokes the messageHandler method, which accepts a SQL exception as its argument. jConnect then processes the message based on what value is returned from messageHandler. The error-message handler can:

- Return the SQL exception as is.
- Return a null. As a result, jConnect ignores the message.
- Create a SQL warning from a SQL exception, and return it. This results in the warning being added to the warning-message chain.
- If the originating message is a SQL warning, messageHandler can evaluate the SQL warning as urgent and create and return a SQL exception to be thrown once control is returned to jConnect.

Installing an error-message handler

You can install an error-message handler implementing SybMessageHandler by calling the setMessageHandler method from SybDriver, SybConnection, or SybStatement. If you install an error-message handler from SybDriver, all subsequent SybConnection objects inherit it. If you install an error-message handler from a SybConnection object, it is inherited by all SybStatement objects created by that SybConnection.

This hierarchy only applies from the time the error-message handler object is installed. For example, if you create a SybConnection object called "myConnection," and then call SybDriver.setMessageHandler to install an error-message handler object, "myConnection" cannot use that object.

To return the current error-message handler object, use getMessageHandler.

Error-message-handler example

```
import java.io.*;
import java.sql.*;
import com.sybase.jdbcx.SybMessageHandler;
import com.sybase.jdbcx.SybConnection;
import com.sybase.jdbcx.SybStatement;
import java.util.*;

public class MyApp
{
  static SybConnection conn = null;
  static SybStatement stmt = null
  static ResultSet rs = null;
  static String user = "guest";
  static String password = "sybase";
  static String server = "jdbc:sybase:Tds:192.138.151.39:4444";
  static final int AVOID_SQLE = 20001;
  public MyApp()
  {
     try
     {
             Class.forName("com.sybase.jdbc4.jdbc.SybDriver").newInstance();
             Properties props = new Properties();
             props.put("user", user);
             props.put("password", password);
             conn = (SybConnection)
             DriverManager.getConnection(server, props);
             conn.setMessageHandler(new NoResultSetHandler());
             stmt =(SybStatement) conn.createStatement();
             stmt.executeUpdate("raiserror 20001 'your error'");

             for (SQLWarning sqw = _stmt.getWarnings();
             sqw != null;
             sqw = sqw.getNextWarning());
             {
                if (sqw.getErrorCode() == AVOID_SQLE);
                {
                 System.out.println("Error" +  sqw.getErrorCode()+
                 " was found in the Statement’s warning list.");
                 break;
                }
             }
             stmt.close();
             conn.close();
  }
  catch(Exception e)
  {
    System.out.println(e.getMessage());
    e.printStackTrace();
  }
}

class NoResultSetHandler implements SybMessageHandler
{
  public SQLException messageHandler(SQLException sqe)
  {
    int code = sqe.getErrorCode();
    if (code == AVOID_SQLE)
    {
      System.out.println("User " + _user + " downgrading " +
        AVOID_SQLE + " to a warning");
      sqe = new SQLWarning(sqe.getMessage(),
        sqe.getSQLState(),sqe.getErrorCode());
    }
    return sqe;
  }
}

public static void main(String args[])
{
  new MyApp();
}
```

Microsecond granularity for time data

jConnect for JDBC provides microsecond-level precision for time data by supporting the SQL datatypes bigdatetime and bigtime.

bigdatetime and bigtime function similarly to and have the same data mappings as the SQL datetime and time datatypes:

- bigdatetime corresponds to the Adaptive Server bigdatetime datatype and indicates the number of microseconds that have passed since January 1, 0000 0:00:00.000000. The range of legal bigdatetime values is from January 1, 0001 00:00:00.000000 to December 31, 9999 23:59:59.999999.
- bigtime corresponds to the Adaptive Server bigtime datatype and indicates the number of microseconds that have passed since the beginning of the day. The range of legal bigtime values is from 00:00:00.000000 to 23:59:59.999999.

Usage

- When connecting to Adaptive Server 15.5, jConnect for JDBC transfers data using the bigdatetime and bigtime datatypes even if the receiving Adaptive Server columns are defined as datetime and time.

  This means that Adaptive Server may silently truncate the values from jConnect for JDBC to fit Adaptive Server columns. For example, a bigtime value of 23:59:59.999999 is saved as 23:59:59.996 in an Adaptive Server column with datatype time.
- When connecting to Adaptive Server 15.0.x and earlier, jConnect for JDBC transfers data using the datetime and time datatypes.

Using password encryption

By default, jConnect for JDBC sends plain text passwords over the network to Adaptive Server for authentication. However, jConnect also supports symmetrical and asymmetrical password encryption and can encrypt passwords before they are sent over the network. The symmetrical encryption mechanism uses the same key to encrypt and decrypt the password whereas an asymmetrical encryption mechanism uses one key (the public key) to encrypt the password and another key (the private key) to decrypt the password. Because the private key is not shared across the network, asymmetrical encryption is considered more secure than symmetrical encryption. When password encryption is enabled, and the server supports asymmetric encryption, this format is used instead of symmetric encryption.

Note To use the asymmetric password encryption feature, you must have a server that supports password encryption, such as Adaptive Server 15.0.2.

Enabling password encryption

The ENCRYPT_PASSWORD connection property specifies whether the password is transmitted in encrypted format. This same property is used to enable asymmetric key encryption. When password encryption is enabled and the server supports asymmetric key encryption, this format is used instead of the symmetric key encryption.

Set the ENCRYPT_PASSWORD connection property to true to enable password encryption. The default value is false.

Note If the server is configured to require clients to use an encrypted password, entering a plain text password causes user login to fail.

Enabling login retry with a clear text password

Server login fails when the ENCRYPT_PASSWORD property is set to True, and the server does not support password encryption. To use a clear text password for servers that do not support password encryption, set the RETRY_WITH_NO_ENCRYPTION connection property to True.

When both ENCRYPT_PASSWORD and RETRY_WITH_NO_ENCRYPTION properties are set to True, jConnect first logs in using the encrypted password. If login fails, jConnect logs in using the clear text password.

Setting up the Java Cryptography Extension (JCE) provider

The asymmetric password encryption mechanism uses RSA encryption algorithms to encrypt the password being transmitted. To perform this RSA encryption, configure your JRE with a suitable Java Cryptography Extension (JCE) provider. The configured JCE provider should be capable of supporting the “RSA/NONE/OAEPWithSHA1AndMGF1Padding” transformation.

The Sun JCE provider included with Sun JREs may not be capable of handling the “RSA/NONE/OAEPWithSHA1AndMGF1Padding” transformation. To use the extended password encryption feature in this case, configure an external JCE provider that includes support for this transformation. If the JCE cannot handle the required transformation, you receive an error message at login.

You can use the JCE_PROVIDER_CLASS connection property to specify the JCE provider. There are a number of commercial and open source JCE providers that you can choose from. For example, the “Bouncy Castle Crypto APIs for Java” is a popular open source Java JCE provider. If you choose not to specify the JCE_PROVIDER_CLASS property, jConnect attempts to use any bundled JCE.

To specify a JCE provider:

- Set the JCE_PROVIDER_CLASS property to the fully qualified class name of the provider you want to use. For example, to use the Bouncy Castle JCE:

```
String url = "jdbc:sybase:Tds:myserver:3697";
Properties props = new Properties();
props.put("ENCRYPT_PASSWORD ", “true”);
props.put("JCE_PROVIDER_CLASS",
"org.bouncycastle.jce.provider.BouncyCastleProvider
");

/* Set up additional connnection properties as
needed */
props.put("user", "xyz");
props.put("password", "123");

/* get  the connection */
```

```
Connection con = DriverManager.getConnection(url,
props);
```

- Configure the JCE provider before using it. This can be done by one of two ways:
    - Copy the JCE provider *jar* file into the JRE standard extension directory:
        - For UNIX / Mac OS X platforms:
          ${JAVA_HOME}/jre/lib/ext
        - For Windows:
          %JAVA_HOME%\jre\lib\ext
    - If you cannot copy the JCE *jar* file to the appropriate directory, refer to the Sun JCE Reference Guide at http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html for instructions on setting up an external JCE provider.

If jConnect cannot use the specified JCE provider, it attempts to use the JCE providers configured in the JRE security profile. If no other JCE providers are configured, or if configured providers do not support the required transformation and password encryption is enabled, the connection fails.

Using GSE-J to perform RSA password encryption

You can use the Certicom Security Builder GSE-J to perform RSA password encryption. Certicom Security Builder GSE-J is a FIPS 140-2 compliant JCE provider that is included in the jConnect driver. This provider contains two JAR files, *EccpressoFIPS.jar* and *EccpressoFIPSJca.jar*, that are both accessible from the *$JDBC_HOME/classes* and the *$JDBC_HOME/devclasses* directories.

To use the Certicom Security Builder GSE-J provider, set the value of JCE_PROVIDER_CLASS connection property to "com.certicom.ecc.jcae.Certicom" and add the *EccpressoFIPS.jar* and *EccpressoFIPSJca.jar* files to the CLASSPATH. See the "Setting up the Java Cryptography Extension (JCE) provider" on page 80 for details.

Note If you enable password encryption by setting the ENCRYPT_PASSWORD connection property but not the JCE_PROVIDER_CLASS connection property, jConnect attempts to locate and load the Certicom Security Builder GSE-J provider. This succeeds only if *EccpressoFIPS.jar* and *EccpressoFIPSJca.jar* are located in the same directory as the jConnect JAR file—*jconn4.jar* or *jconn4d.jar*— in use.

Storing Java objects as column data in a table

Some database products enable you to directly store Java objects as column data in a database. In such databases, Java classes are treated as datatypes, and you can declare a column with a Java class as its datatype.

jConnect supports storing Java objects in a database by implementing the setObject methods defined in the PreparedStatement interface and the getObject methods defined in the CallableStatement and ResultSet interfaces. This allows you to use jConnect with an application that uses native JDBC classes and methods to directly store and retrieve Java objects as column data.

Note To use getObject and setObject, set the jConnect version to com.sybase.jdbcx.SybDriver.VERSION_4 or later. See "Using JCONNECT_ VERSION" on page 6.

The following sections describe the requirements and procedures for storing objects in a table and retrieving them using JDBC with jConnect:

- Prerequisites for storing java objects as column data
- Sending Java objects to a database

- Receiving Java objects from the database

Note Adaptive Server version 12.0 and later and SQL Anywhere version 6.0.x and later can store Java objects in a table, with some limitations. See the *jConnect for JDBC Release Bulletin* for more information.

Prerequisites for storing java objects as column data

To store Java objects belonging to a user-defined Java class in a column, three requirements must be met:

- The class must implement the java.io.Serializable interface. This is because jConnect uses native Java serialization and deserialization to send objects to a database and receive them back from the database.
- The class definition must be installed in the destination database, *or* you must be using the DynamicClassLoader (DCL) to load a class directly from an SQL Anywhere or an Adaptive Server server and use it as if it were present in the local CLASSPATH. See “Using dynamic class loading” on page 87 for more information.
- The client system must have the class definition in a *.class* file that is accessible through the local CLASSPATH environment variable.

Sending Java objects to a database

To send an instance of a user-defined class as column data, use one of the following setObject methods, as specified in the PreparedStatement interface:

```
void setObject(int parameterIndex, Object x, int targetSqlType,
   int scale) throws SQLException;
void setObject(int parameterIndex, Object x, int targetSqlType)
   throws SQLException;
void setObject(int parameterIndex, Object x) throws SQLException;
```

In jConnect, to send a Java object, you can use the java.sql.Types.JAVA_OBJECT target sql.Type, or you can use java.sql.Types.OTHER.

The following example defines an Address class, shows the definition of a Friends table that has an Address column whose datatype is the Address class, and inserts a row into the table.

```
public class Address implements Serializable
{
  public String   streetNumber;
   public String   street;
   public String   apartmentNumber;
   public String   city;
   public int   zipCode;
  //Methods
  ...
}

/* This code assumes a table with the following structure
**  Create table Friends:
**  (firstname varchar(30)  ,
**   lastname varchar(30),
**  address Address,
**  phone varchar(15))
*/

// Connect to the database containing the Friends table.
Connection conn =
   DriverManager.getConnection("jdbc:sybase:Tds:localhost:5000",
     "username", "password");

// Create a Prepared Statement object with an insert statement
//for updating the Friends table.
PreparedStatement ps = conn.prepareStatement("INSERT INTO
   Friends values (?,?,?,?)");

// Now, set the values in the prepared statement object, ps.
// set firstname to "Joan."
ps.setString(1, "Joan");

// Set last name to "Smith."
ps.setString(2, "Smith");

// Assuming that we already have "Joan_address" as an instance
// of Address, use setObject(int parameterIndex, Object x) to
// set the address column to "Joan_address."
ps.setObject(3, Joan_address);

// Set the phone column to Joan's phone number.
ps.setString(4, "123-456-7890");

// Perform the insert.
ps.executeUpdate();
```

Receiving Java objects from the database

A client JDBC application can receive a Java object from the database in a result set or as the value of an output parameter returned from a stored procedure.

If a result set contains a Java object as column data, use one of the following getObject methods in the ResultSet interface to retrieve the object:

```
Object getObject(int columnIndex) throws SQLException;
Object getObject(String columnName) throws SQLException;
```

If an output parameter from a stored procedure contains a Java object, use the following getObject method in the CallableStatement interface to retrieve the object:

```
Object getObject(int parameterIndex) throws SQLException;
```

The following example illustrates the use of ResultSet.getObject(int parameterIndex) to assign an object received in a result set to a class variable. The example uses the Address class and Friends table used in the previous section and presents a simple application that prints a name and address on an envelope.

```
/*
 ** This application takes a first and last name, gets the
 ** specified person’s address from the Friends table in the
 ** database, and addresses an envelope using the name and
 ** retrieved address.
 */
 public class Envelope
 {
   Connection conn = null;
   String firstName = null;
   String lastName = null;
   String street = null;
   String city = null;
   String zip = null;

   public static void main(String[] args)
   {
     if (args.length < 2)
     {
     System.out.println("Usage: Envelope <firstName>
       <lastName>");
     System.exit(1);
     }
     // create a 4" x 10" envelope
```

```
    Envelope e = new Envelope(4, 10);
    try
    {
      // connect to the database with the Friends table.
      conn = DriverManager.getConnection(
        "jdbc:sybase:Tds:localhost:5000", "username",
          "password");
      // look up the address of the specified person
      firstName = args[0];
      lastName = args[1];
      PreparedStatement ps = conn.prepareStatement(
        "SELECT address FROM friends WHERE " +
          "firstname = ? AND lastname = ?");
      ps.setString(1, firstName);
      ps.setString(2, lastName);
      ResultSet rs = ps.executeQuery();
      if (rs.next())
      {
        Address a = (Address) rs.getObject(1);
        // set the destination address on the envelope
        e.setAddress(firstName, lastName, a);
      }
      conn.close();
    }
    catch (SQLException sqe)
    {
      sqe.printStackTrace();
      System.exit(2);
    }
    // if everything was successful, print the envelope
    e.print();
  }
  private void setAddress(String fname, String lname, Address a)
  {
    street = a.streetNumber + " " + a.street + " " +
      a.apartmentNumber;
    city = a.city;
    zip = "" + a.zipCode;
  }
  private void print()
  {
    // Print the name and address on the envelope.
    ...
  }
}
```

You can find a more detailed example of HandleObject.java in the *sample2* subdirectory under your jConnect installation directory.

Using dynamic class loading

SQL Anywhere and Adaptive Server allow you to specify Java classes as:

- Datatypes of SQL columns
- Datatypes of Transact-SQL variables
- Default values for SQL columns

In earlier versions, only classes that appeared in the jConnect CLASSPATH were accessible, that is, if a jConnect application attempted to access an instance of a class that was not in the local CLASSPATH, a java.lang.ClassNotFound exception would result.

jConnect version 6.05 and later implements DynamicClassLoader (DCL) to load a class directly from an SQL Anywhere or Adaptive Server server and use it as if it were present in the local CLASSPATH.

All security features present in the superclass are inherited. The loader delegation model implemented in Java 2 is followed—first jConnect attempts to load a requested class from the CLASSPATH; if that fails, jConnect tries the DynamicClassLoader.

See *Java in Adaptive Server* for more detailed information about using Java and Adaptive Server.

Using *DynamicClassLoader*

To use DCL functionality:

1. Create and configure a class loader. The code for your jConnect application should look similar to this:

```
Properties props = new Properties();

// URL of the server where the classes live.
String classesUrl = "jdbc:sybase:Tds:myase:1200";

// Connection properties for connecting to above server.
props.put("user", "grinch");
props.put("password", "meanone");
...

// Ask the SybDriver for a new class loader.
```

```
DynamicClassLoader loader = driver.getClassLoader(classesUrl, props);
```

2 Use the CLASS_LOADER connection property to make the new class loader available to the statement that executes the query. Once you create the class loader, pass it to subsequent connections as shown (continuing from the code example in step 1):

```
// Stash the class loader so that other connection(s)
// can know about it.
props.put("CLASS_LOADER", loader);

// Additional connection properties
props.put("user", "joeuser");
props.put("password", "joespassword");

// URL of the server we now want to connect to.
String url = "jdbc:sybase:Tds:jdbc.sybase.com:4446";

// Make a connection and go.
Connection conn = DriverManager.getConnection(url, props);
```

Assume the Java class definition is as follows:

```
class Addr {
      String street;
      String city;
      String state;
}
```

Assume the SQL table definition is as follows:

```
create table employee (char(100) name, int empid, Addr address)
```

3 Use the following client-side code in the absence of an Addr class in the client application CLASSPATH:

```
Statement stmnt = conn.createStatement();
// Retrieve some rows from the table that has a Java class
// as one of its fields.
ResultSet rs = stmnt.executeQuery(
      "select * from employee where empid = '19'");
if (rs.next() {
      // Even though the class is not in our class path,
      // we should be able to access its instance.
      Object obj = rs.getObject("address");
      // The class has been loaded from the server, so let's take a look.
      Class c = obj.getClass();
    // Some Java Reflection can be done here to access the fields of obj.
      ...
}
```

The CLASS_LOADER connection property provides a convenient mechanism for sharing one class loader among several connections.

You should ensure that sharing a class loader across connections does not result in class conflicts. For example, if two different, incompatible instances of class org.foo.Bar exist in two different databases, problems can arise if you use the same loader to access both classes. The first class is loaded when examining a result set from the first connection. When it is time to examine a result set from the second connection, the class is already loaded. Consequently, the second class is never loaded, and there is no direct way for jConnect to detect this situation.

However, Java has a built-in mechanism for ensuring that the version of a class matches the version information in a deserialized object. The above situation is at least detected and reported by Java.

Classes and their instances do not need to reside in the same database or server, but there is no reason why both the loader and subsequent connections cannot refer to the same database or server.

Using deserialization

The following example illustrates how to deserialize an object from a local file. The serialized object is an instance of a class that resides on a server and does not exist in the CLASSPATH.

SybResultSet.getObject( ) makes use of DynamicObjectInputStream, which is a subclass of ObjectInputStream that loads a class definition from DynamicClassLoader, rather than the default system (“boot”) class loader.

```
// Make a stream on the file containing the
//serialized object.
FileInputStream fileStream = new FileInputStream("serFile");
// Make a "deserializer" on it. Notice that, apart
//from the additional parameter, this is the same
//as ObjectInputStreamDynamicObjectInputStream
stream = new DynamicObjectInputStream(fileStream, loader);
// As the object is deserialized, its class is
//retrieved through the loader from our server.
Object obj = stream.readObject();stream.close();
```

Preloading *.jar* files

jConnect version 6.05 or later has a connection property called PRELOAD_JARS. When defined as a comma-delimited list of *.jar* file names, the *.jar* files are loaded in their entirety. In this context, "JAR" refers to the "retained JARname" used by the server. This is the *.jar* file name specified in the install Java program, for example:

```
install java new jar 'myJarName' from file '/tmp/mystuff.jar'
```

If you set PRELOAD_JARS, the *.jar* files are associated with the class loader, so it is unnecessary to preload them with every connection. You should only specify PRELOAD_JARS for one connection. Subsequent attempts to preload the same *.jar* files may result in performance problems as the *.jar* file data is retrieved from the server unnecessarily.

Note SQL Anywhere cannot return a *.jar* file as one entity, so jConnect iteratively retrieves each class in turn. However, Adaptive Server retrieves the entire *.jar* file and loads each class that it contains.

Advanced features

There are various public methods in DynamicClassLoader. For more information, see the javadocs information in *JDBC_HOME/docs/en/javadocs*.

Additional features include the ability to keep the database connection of a loader "alive" when a series of class loads is expected, and to explicitly load a single class by name.

Public methods inherited from java.lang.ClassLoader can also be used. Methods in java.lang.Class that deal with loading classes are also available; however, use these methods with caution because some of them make assumptions about which class loader gets used. In particular, you should use the 3-argument version of Class.forName, otherwise the system ("boot") class loader is used. See "Handling error messages" on page 73.

JDBC 4.0 specifications support

These JDBC 4.0 specifications are supported:

- Connection management
- Automatic SQL driver loading

- Database metadata
- National character set conversion
- Wrapper pattern
- Scalar functions CHAR_LENGTH, CHARACTER_LENGTH, CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, EXTRACT, and OCTET_LENGTH, POSITION

See the Sun Developer Network at http://developers.sun.com/ for information about the JDBC 4.0 specifications.

JDBC 3.0 specifications support

This section describes the JDBC 3.0 features that are supported in the current release of jConnect 7.0.

Savepoint support

Adds the Savepoint interface, which contains methods to set, release, or roll back a transaction to designated savepoints.

Using Savepoints in your transactions

The transaction support in JDBC 2.0 allowed you to have control over a transaction and roll back every change in a transaction. In JDBC 3.0, you are given more control with savepoints: the Savepoint interface allows you to partition a transaction into logical breakpoints, providing control over how much of the transaction gets rolled back.

Setting and rolling back to a Savepoint

The JDBC 3.0 API adds the method Connection.setSavepoint, which sets a savepoint within the current transaction and returns a Savepoint object. The Connection.rollback method is overloaded to take a Savepoint object argument.

Releasing a Savepoint

The Connection.releaseSavepoint method takes a Savepoint object as a parameter and removes it from the current transaction. After a Savepoint has been released, if you try to reference it in a rollback operation, a SQLException occurs.

Any savepoints that you create in a transaction are automatically released and become invalid when the transaction is committed or when the entire transaction is rolled back. If you roll a transaction back to a savepoint, it automatically releases and invalidates any other savepoints that were created after the savepoint in question.

Note You can use the DatabaseMetaData.supportsSavepoints method to determine whether a JDBC API implementation supports savepoints.

Retrieval of parameter metadata

Adds the interface ParameterMetaData, which describes the number, type, and properties of parameters to prepared statements, and supports the new and modified DatabaseMetaData methods.

Retrieval of auto-generated keys

Adds a way to retrieve values from columns that contain automatically generated values. JDBC 3.0 addresses the common need to obtain the value of an auto-generated or auto-incremented key.

Determine the value of a generated key

To inform the driver that you want to retrieve the auto-generated keys, pass the constant Statement.RETURN_GENERATED_KEYS as the second parameter of the Statement.execute() method. After you have executed the statement, call Statement.getGeneratedKeys() to retrieve the generated keys. The result set will contain a row for each generated key retrieved.

Note Adaptive Server cannot return a result set of generated keys. If you execute a batch of insert commands, invoking Statement.getGeneratedKeys() will return the value of the last generated key only.

For more information about retrieving auto-generated keys, including a sample code, search for “retrieving automatically generated keys” on the Sun Microsystems Web site.

Ability to have multiple open ResultSet objects

Adds getMoreResults(int), which takes an argument that specifies whether ResultSet objects returned by a Statement object should be closed before returning any subsequent ResultSet objects.

As a part of the changes, the JDBC 3.0 specification allows the Statement interface to support multiple open ResultSets, which removes the limitation of the JDBC 2 specification that statements returning multiple results must have only one ResultSet open at any given time. To support multiple open results, the Statement interface adds an overloaded version of the method getMoreResults(). The getMoreResults(int) method takes an integer flag that specifies the behavior of previously opened ResultSets when the getResultSet() method is called. The interface defines the flags as follows:

- CLOSE_ALL_RESULTS – all previously opened ResultSet objects are closed when calling getMoreResults().
- CLOSE_CURRENT_RESULT – the current ResultSet object are closed when calling getMoreResults().
- KEEP_CURRENT_RESULT – the current ResultSet object is not closed when calling getMoreResults().

Passing parameters to *CallableStatement* objects by name

Adds methods to allow a string to identify the parameter to be set for a CallableStatement object.

You can use the CallableStatement interface to specify parameters by their names and not the previous method of specifying the parameter's index. You will find this useful when a procedure has many parameters with default values. You can use named parameters to specify only the values that have no default value.

Holdable cursor support

Adds the ability to specify the holdability of a ResultSet object. A holdable cursor, or result, is one that does not automatically close when the transaction that contains the cursor is committed. JDBC 3.0 adds support for specifying cursor holdability. For you to specify the holdability of your ResultSet, you must do so when you prepare a statement using the createStatement(), prepareStatement(), or prepareCall() methods. The holdability may be one of the following constants:

- HOLD_CURSORS_OVER_COMMIT – ResultSet objects (cursors) are not closed; they are held open when a commit operation is implicitly or explicitly performed.
- CLOSE_CURSORS_AT_COMMIT – ResultSet objects (cursors) are closed when a commit operation is implicitly or explicitly performed.

If you close a cursor when a transaction is committed, it usually results in better performance. Unless you require the cursor after the transaction, it is recommended that you close the cursor when the commit operation is carried out. Because the specification does not define the default holdability of a ResultSet, its behavior will depend on the implementation.

JDBC 2.0 optional package extensions support

The *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*) defined several features that JDBC 2.0 drivers could implement. jConnect version 6.05 and later have implemented the following optional package extension features:

- JNDI for naming databases
  (works with any Sybase DBMS supported by jConnect)
- Connection pooling
  (works with any Sybase DBMS supported by jConnect)
- Distributed transaction management support works only with Adaptive Server.

Note Sybase recommends that you use JNDI 1.2, which is compatible with Java 1.1.6 and later.

JNDI for naming databases

Reference

The *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*), Chapter 5, “JNDI and the JDBC API.”

Related interfaces

- javax.sql.DataSource
- javax.naming.Referenceable
- javax.naming.spi.ObjectFactory

This feature provides JDBC clients with an alternative to the standard approach for obtaining database connections. Instead of invoking Class.forName ("com.sybase.jdbc4.jdbc.SybDriver"), then passing a JDBC URL to the DriverManager's getConnection( ) method, clients can access a JNDI name server using a logical name to retrieve a javax.sql.DataSource object. This object is responsible for loading the driver and establishing the connection to the physical database it represents. The client code is simpler and reusable because the vendor-specific information has been placed within the DataSource object.

The Sybase implementation of the DataSource object is com.sybase.jdbcx.SybDataSource (see the javadocs for details). This implementation supports the following standard properties using the design pattern for JavaBean components:

- databaseName
- dataSourceName
- description
- networkProtocol
- password
- portNumber
- serverName
- user

Note roleName is not supported.

---

jConnect provides an implementation of the javax.naming.spi.ObjectFactory interface so the DataSource object can be constructed from the attributes of a name server entry. When given a javax.naming.Reference, or a javax.naming.Name and a javax.naming.DirContext, this factory can construct com.sybase.jdbcx.SybDataSource objects. To use this factory, set the java.naming.object.factory system property to include com.sybase.jdbc4.SybObjectFactory.

Usage

You can use DataSource in different ways, in different applications. All options are presented in the following subsections with some code examples to guide you through the process. For more information, see the *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*), and the JNDI documentation on the Sun Web site.

1a. Configuration by administrator: LDAP

jConnect has supported LDAP connectivity since version 4.0. As a result, the recommended approach, which requires no custom software, is to configure DataSources as LDAP entries using the LDAP Data Interchange Format (LDIF). For example:

```
dn:servername:myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
```

1b. Access by client

This is the typical JDBC client application. The only difference is that you access the name server to obtain a reference to a DataSource object, instead of accessing the DriverManager and providing a JDBC URL. Once you obtain the connection, the client code is identical to any other JDBC client code. The code is very generic and references Sybase only when setting the object factory property, which can be set as part of the environment.

The jConnect installation contains the sample program *sample2/SimpleDataSource.java* to illustrate the use of DataSource. This sample is provided for reference only, that is, you cannot run the sample unless you configure your environment and edit the sample appropriately. *SimpleDataSource.java* contains the following critical code:

```
import javax.naming.*;
import javax.sql.*;
import java.sql.*;

// set necessary JNDI properties for your environment (same as above)
Properties jndiProps = new Properties();

// used by JNDI to build the SybDataSource
jndiProps.put(Context.OBJECT_FACTORIES,
    "com.sybase.jdbc4.jdbc.SybObjectFactory");

// nameserver that JNDI should talk to
jndiProps.put(Context.PROVIDER_URL, "ldap: some_ldap_server:238/" +
"o=MyCompany,c=Us");

// used by JNDI to establish the naming context
```

```
jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.ldap.LdapCtxFactory");

// obtain a connection to your name server
Context ctx = new InitialContext(jndiProps);
DataSource ds = (DataSource) ctx.lookup("servername=myASE");

// obtains a connection to the server as configured earlier.
// in this case, the default username and password will be used
Connection conn = ds.getConnection();

// do standard JDBC methods
...
```

Explicitly passing the Properties to the InitialContext constructor is not required if the properties have already been defined within the virtual machine, that is, passed when Java was either set as part of the browser properties, or by using the following:

```
java -Djava.naming.object.factory=com.sybase.jdbc4.jdbc.SybObjectFactory
```

See your Java VM documentation for more information about setting environment properties.

2a. Programmatic configuration

This phase is typically done by the person who performs database system administration or application integration for their company. The purpose is to define a data source, then deploy it under a logical name to a name server. If the server needs to be reconfigured (for example, moved to another machine, port, and so on), then the administrator runs this configuration utility (outlined as follows) and reassigns the logical name to the new data source configuration. As a result, the client code does not change, since it knows only the logical name.

```
import javax.sql.*;
import com.sybase.jdbcx.*;
.....

// create a SybDataSource, and configure it
SybDataSource ds = new com.sybase.jdbc4.jdbc.SybDataSource();
ds.setUser("my_username");
ds.setPassword("my_password");
ds.setDatabaseName("my_favorite_db");
ds.setServerName("db_machine");
ds.setPortNumber(4000);
ds.setDescription("This DataSource represents the Adaptive Server
    Enterprise server running on db_machine at port 2638.  The default
    username and password have been set to 'me' and 'mine' respectively.
```

```
        Upon connection, the user will access the my_favorite_db database on
        this server.");
    Properties props = newProperties()
    props.put("REPEAT_READ","false");
    props.put("REQUEST_HA_SESSION","true");
    ds.setConnectionProperties(props);
    // store the DataSource object. Typically this is
    // done by setting JNDI properties specific to the
    // type of JNDI service provider you are using.
    // Then, initialize the context and bind the object.
    Context ctx = new InitialContext();
    ctx.bind("java:comp/env/jdbc/myASE", ds);
```

Once you set up your DataSource, you decide where and how you want to store the information. To assist you, SybDataSource is both java.io.Serializable and javax.naming.Referenceable, but it is still up to the administrator to determine how the data is stored, depending on what service provider you are using for JNDI.

2b. Access by client

The client retrieves the DataSource object by setting its JNDI properties the same way the DataSource was deployed. The client needs to have an object factory available that can transform the object as it is stored (for example, serialized) into a Java object.

```
    Context ctx = new InitialContext();
    DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/myASE");
    Connection conn = ds.getConnection();
```

Connection pooling

Reference

The *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*), Chapter 6, “Connection Pooling.”

Related interfaces

- javax.sql.ConnectionPoolDataSource
- javax.sql.PooledConnection

Overview

Traditional database applications create one connection to a database that you use for each session of an application. However, a Web-based database application may need to open and close a new connection several times when using the application.

An efficient way to handle Web-based database connections is to use connection pooling, which maintains open database connections and manages connection sharing across different user requests to maintain performance and to reduce the number of idle connections. On each connection request, the connection pool first determines if there is an idle connection in the pool. If there is, the connection pool returns that connection instead of making a new connection to the database.

The com.sybase.jdbc4.jdbc.ConnectionPoolDataSource class is provided to interact with connection pooling implementations. When you use ConnectionPoolDataSource, pool implementations listen to the PooledConnection. The implementation is notified when you close the connection, or if you have an error that destroys the connection. At this point, the pool implementation decides what to do with the PooledConnection.

Without connection pooling, a transaction:

1. Creates a connection to the database.
2. Sends the query to the database.
3. Gets back the result set.
4. Displays the result set.
5. Destroys the connection.

With connection pooling, the sequence looks more like this:

1. Sees if an unused connection exists in the "pool" of connections.
2. If so, uses it; otherwise creates a new connection.
3. Sends the query to the database.
4. Gets back the result set.
5. Displays the result set.
6. Returns the connection to the "pool." The user still calls "close( )", but the connection remains open, and the pool is notified of the close request.

It is less costly to reuse a connection than to create a new one every time a client needs to establish a connection to a database.

To enable a third party to implement the connection pool, the jConnect implementation has the ConnectionPoolDataSource interface produce PooledConnections, similar to the way the DataSource interface produces Connections.

The pool implementation creates "real" database connections, using the getPooledConnection( ) methods of ConnectionPoolDataSource. Then, the pool implementation registers itself as a listener to the PooledConnection.

Currently, when a client requests a connection, the pool implementation invokes getConnection( ) on an available PooledConnection. When the client finishes with the connection and calls close, the pool implementation is notified through the ConnectionEventListener interface that the connection is free and available for reuse.

The pool implementation is also notified through the ConnectionEventListener interface if the client somehow corrupts the database connection, so that the pool implementation can remove that connection from the pool.

For more information, refer to Appendix B in the *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*).

Configuration by administrator: LDAP

This approach is the same as "1a. Configuration by administrator: LDAP" described in "JNDI for naming databases," except that you enter an additional line to your LDIF entry. In the following example, the added line of code is bolded for your reference.

```
dn:servername=myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
1.3.6.1.4.1.897.4.2.18:ConnectionPoolDataSource
```

Access by middle-tier clients

This procedure initializes three properties (INITIAL_CONTEXT_FACTORY, PROVIDER_URL, and OBJECT_FACTORIES as shown on page 78), and retrieves a ConnectionPoolDataSource object. For a more complete code example, see *sample2/SimpleConnectionPool.java*. The fundamental difference is:

```
...
ConnectionPoolDatabase cpds = (ConnectionPoolDataSource)
    ctx.lookup("servername=myASE");
PooledConnection pconn = cpds.getPooledConnection();
```

Distributed transaction management support

This feature provides a standard Java API for performing distributed transactions with either Adaptive Server.

Note This feature is designed for use in a large multitier environment.

Reference

See Chapter 7, "Distributed Transactions," in the *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*).

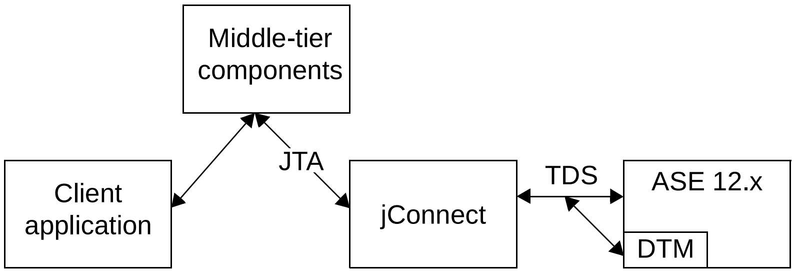
Related interfaces

- javax.sql.XADataSource
- javax.sql.XAConnection
- javax.transaction.xa.XAResource

Background and system requirements

- Because jConnect is communicating directly with the resource manager within Sybase Adaptive Server version 12.0 and later, the installation must have Distributed Transaction Management support.
- Any user who wants to participate in a distributed transaction must have the "dtm_tm_role" granted, or the transactions fail.
- To use distributed transactions, you must install the stored procedures in the */sp* directory. Refer to "Installing Stored Procedures" in Chapter 1 of the *jConnect for JDBC Installation Guide*.

Figure 2-2: Distributed transaction management support with version 12.x



Configuration by administrator: LDAP

This approach is the same as "1a. Configuration by administrator: LDAP" described in "JNDI for naming databases" on page 94, except that you enter an additional line to the LDIF entry. In the following example, the added line of code is displayed in bold.

```
dn:servername:myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
1.3.6.1.4.1.897.4.2.18:XADataSource
```

Access by middle-tier clients

This procedure initializes three properties (INITIAL_CONTEXT_FACTORY, PROVIDER_URL, and OBJECT_FACTORIES), and retrieves a XADataSource object. For example:

```
...
XADataSource xads = (XADatasource) ctx.lookup("servername=myASE");
XAConnection xaconn = xads.getXAConnection();
```

or override the default settings for the user name and password:

```
...
XADataSource xads = (XADatasource) ctx.lookup("servername=myASE");
XAConnection xaconn = xads.getXAConnection("my_username","my_password");
```

Restrictions on and interpretations of JDBC standards

This section discusses how the jConnect implementation of JDBC deviates from the JDBC standards. The following topics are covered:

- Unsupported JDBC 4.0 specification requirements

- Using Connection.isClosed and IS_CLOSED_TEST
- Using Statement.close with unprocessed results
- Making adjustments for multithreading
- Using ResultSet.getCursorName
- Using setLong with large parameter values
- Executing stored procedures

Unsupported JDBC 4.0 specification requirements

These are not supported in this release:

- java.sql.RowID
- BLOB, CLOB, or NCLOB methods
- XML APIs introduced in JDBC 4.0

Using Connection.isClosed and IS_CLOSED_TEST

According to section 11.1 of the JDBC 2.1 specification:

"The Connection.isClosed method is only guaranteed to return true after Connection.close has been called. Connection.isClosed cannot be called, in general, to determine if a database connection is valid or invalid. A typical client can determine that a connection is invalid by catching the exception that is thrown when an operation is attempted."

jConnect offers a default interpretation of the isClosed method that is different from the behavior that is defined in the spec. When you call Connection.isClosed, jConnect first verifies that Connection.close has been called on this connection. If close has been called, jConnect returns true for isClosed.

However, if Connection.close has not been called, jConnect next tries to execute the sp_mda stored procedure on the database. The sp_mda stored procedure is part of the standard metadata that jConnect users must install when they use jConnect with a database.

The purpose of calling sp_mda is so that jConnect can try to execute a procedure that is known (or at least, expected) to reside on the database server. If the stored procedure executes normally, then jConnect returns false for isClosed because we have verified that the database connection is valid and working. However, if the call to sp_mda results in a SQLException being thrown, jConnect catches the exception and returns true for isClosed because it appears that there is something wrong with the connection.

If you intend to force jConnect to more closely follow the standard JDBC behavior for isClosed(), you can do so by setting the IS_CLOSED_TEST connection property to the special value "INTERNAL." The INTERNAL setting means that jConnect returns true for isClosed only when Connection.close has been called, or when jConnect has detected an IOException that has disabled the connection.

You can also specify a query other than sp_mda to use when isClosed is called. For example, if you intend for jConnect to attempt a select 1 when isClosed is called, you can set the IS_CLOSED_TEST connection property to select 1.

Using Statement.close with unprocessed results

The JDBC specification is somewhat vague on how a driver should behave when you call Statement.execute and later call close on that same statement object without processing all of the results (update counts and ResultSets) returned by the Statement.

For example, assume that there is a stored procedure on the database that does seven row inserts. An application then executes that stored procedure using a Statement.execute. In this case, a Sybase database returns seven update counts (one for each inserted row) to the application. In normal JDBC application logic, you would process those update counts in a loop using the getMoreResults, getResultSet and getUpdateCount methods. These are clearly explained on the Sun Web page for Java developers at http://java.sun.com/ in the javadocs for the java.sql.* package.

However, an application programmer might incorrectly choose to call Statement.close before reading through all of the returned update counts. In this case, jConnect sends a cancel to the database, which can have unexpected and unwanted side effects.

In this particular example, if the application calls Statement.close before the database has completed the inserts, the database might not execute all of the inserts. It might stop, for example, after only five rows are inserted because the cancel is processed on the database before the stored procedure completes.

The missing inserts would not be reported to you in this case. Future releases of jConnect may throw a SQLException when you try to close a Statement when there are still unprocessed results, but until then, jConnect programmers are strongly advised to adhere to the following guidelines:

- When you call Statement.close, a cancel is sent to the server if not all the results (update counts and ResultSets) have been completely processed by you. In cases where you only executed select statements, this is fine. However, in cases where you executed insert/update/delete operations, this can result in not all of those operations completing as expected.
- Therefore, you should never call close with unprocessed results when you have executed anything but pure select statements.
- Instead, if you call Statement.execute, be sure your code processes all the results by using the getUpdateCount, getMoreResults and getResultSet methods.

Making adjustments for multithreading

If several threads simultaneously call methods on the same Statement instance, CallableStatement, or PreparedStatement—which Sybase does not recommend— you must manually synchronize the calls to the methods on the Statement; jConnect does not do this automatically.

For example, if you have two threads operating on the same Statement instance—one thread sending a query and the other thread processing warnings—you must synchronize the calls to the methods on the Statement or conflicts may occur.

Using *ResultSet.getCursorName*

Some JDBC drivers generate a cursor name for any SQL query so that a string can always be returned. However, jConnect does not return a name when ResultSet.getCursorName is called, unless you either:

- Called setFetchSize or setCursorName on the corresponding Statement, or
- Set the SELECT_OPENS_CURSOR connection property to “true,” and your query was in the form of SELECT... FOR UPDATE. For example:

```
select au_id from authors for update
```

If you do not call setFetchSize or setCursorName on the corresponding Statement, or set the SELECT_OPENS_CURSOR connection property to "true," null is returned.

According to the JDBC 2.0 API documentation (see Chapter 11, "Clarifications"), all other SQL statements do not need to open a cursor and return a name.

For more information on how to use cursors in jConnect, see "Using cursors with result sets" on page 47.

Using *setLong* with large parameter values

Implementations of the PreparedStatement.setLong method set a parameter value to a SQL BIGINT datatype. Most Adaptive Server databases do not have an 8-byte BIGINT datatype. If a parameter value requires more than 4 bytes of a BIGINT, using setLong can result in an overflow exception.

Datatypes supported

jConnect supports the following Adaptive Server datatypes:

- bigint – An exact numeric datatype designed to be used when the range of the existing int types is insufficient.
- unsigned int – Unsigned versions of the exact numeric integer datatypes: unsignedsmallint, unsignedint, and unsignedbigint.
- unitext – A variable-length datatype for Unicode characters.

B*igint* datatype

Sybase supports bigint, which is a 64-bit integer datatype that is supported as a native Adaptive Server datatype. In Java, this datatype maps to java datatype long. To use this as a parameter, you can call PreparedStatement.setLong(int index, long value) and jConnect sends the data as bigint to Adaptive Server. When retrieving from a bigint column, you can use the ResultSet.getLong(int index) method.

Unitext datatypes

There are no API changes in jConnect for using the unitext datatype. jConnect can internally handle storage and retrieval of data from Adaptive Server when unitext columns are used.

Unsigned int datatypes

Adaptive Server supports *unsigned bigint*, *unsigned int*, and *unsigned smallint* as native Adaptive Server datatypes. Because, there are no corresponding unsigned datatypes in Java, you must set and get the next higher integer if you want to process the data correctly. For example, if you are retrieving data from an *unsigned int*, using the Java datatype int is too small to contain positive large values, and as a result, ResultSet.getInt (int index) might return incorrect data or throw an exception. To process the data correctly, you should get the next higher integer value ResultSet.getLong(). You can use the following table to set or get data.

Adaptive Server datatype	Java datatype
unsigned smallint	setInt(), getInt()
unsigned int	setLong(), getLong()
unsigned bigint	setBigDecimal(), getBigDecimal()

Executing stored procedures

If you execute a stored procedure in a CallableStatement object that represents parameter values as question marks, you get better performance than if you use both question marks and literal values for parameters. Also, if you mix literals and question marks, you cannot use output parameters with a stored procedure.

The following example creates *sp_stmt* as a CallableStatement object for executing the stored procedure MyProc:

```
CallableStatement sp_stmt = conn.prepareCall(
   "{call MyProc(?,?)}");
```

The two parameters in MyProc are represented as question marks. You can register one or both of them as output parameters using the registerOutParameter methods in the CallableStatement interface.

In the following example, *sp_stmt2* is a CallableStatement object for executing the stored procedure MyProc2.

```
CallableStatement sp_stmt2 = conn.prepareCall(
   {"call MyProc2(?,'javelin')}");
```

In *sp_stmt2*, one parameter value is given as a literal value and the other as a question mark. You cannot register either parameter as an output parameter.

To execute stored procedures with RPC commands using name-binding for parameters, use either of the following procedures:

- Use language commands, passing input parameters to them directly from Java variables using the PreparedStatement class. This is illustrated in the following code fragment:

```
// Prepare the statement
System.out.println("Preparing the statement...");
String stmtString = "exec " + procname + " @p3=?, @p1=?";
PreparedStatement pstmt = con.preparedStatement(stmtString);

// Set the values
pstmt.setString(1, "xyz");
pstmt.setInt(2, 123);

// Send the query
System.out.println("Executing the query...");
ResultSet rs = pstmt.executeQuery();
```

- With jConnect version 6.05 and later, use the com.sybase.jdbcx.SybCallableStatement interface, illustrated in this example:

```
import com.sybase.jdbcx.*;
....
// prepare the call for the stored procedure to execute as an RPC
String execRPC = "{call " + procName + " (?, ?)}";
SybCallableStatement scs = (SybCallableStatement)
con.prepareCall(execRPC);

// set the values and name the parameters
// also (optional) register for any output parameters
scs.setString(1, "xyz");
scs.setParameterName(1, "@p3");
scs.setInt(2, 123);
scs.setParameterName(2, "@p1");

// execute the RPC
// may also process the results using getResultSet()
// and getMoreResults()
// see the samples for more information on processing results
ResultSet rs = scs.executeQuery();
```

CHAPTER 3

Security

This chapter describes security issues for jConnect.

Overview

jConnect provides the following options for securing client-server communications:

- *SSL* – Use SSL to encrypt communications, including the login exchange, between client and server applications.
- *Kerberos* – Use Kerberos to authenticate Java applications or users of Java applications to Adaptive Server without sending user names or passwords over a network. Also use Kerberos to set up a Single Sign-On (SSO) environment and provide mutual authentication between the digital identity of a Java application and that of Adaptive Server Enterprise.

  Note Kerberos may be used to encrypt communications and provide data integrity checking, but these have not been implemented for jConnect.

Kerberos and SSL may also be used together, providing the advantage of both SSO and encryption of data transferred between client and server applications.

Restrictions

Kerberos and SSL can be used with Adaptive Server; SQL Anywhere does not currently support either SSL or Kerberos security.

Sybase recommends that you read related documentation about SSL and Kerberos before attempting to use either with jConnect. The information in this chapter assumes that the servers you intend to use have been configured to work properly with SSL, with Kerberos, or with both.

For further information on Kerberos, SSL, and configuring Adaptive Server Enterprise, see “Related documents” on page 129. Also, see the white paper on setting up Kerberos. The URL for this document can be found in the *jConnect for JDBC Release Bulletin*.

Implementing custom socket plug-ins

This section discusses how to plug a custom socket implementation into an application to customize the communication between a client and server. javax.net.ssl.SSLSocket is an example of a socket that you could customize to enable encryption.

com.sybase.jdbcx.SybSocketFactory is a Sybase extension interface that contains the createSocket(String, int, Properties) method that returns a java.net.Socket. To use a custom socket factory in jConnect, an application must implement this interface by defining the createSocket() method.

jConnect uses the new socket for its subsequent input/output operations. Classes that implement SybSocketFactory create sockets and provide a general framework for the addition of public socket-level functionality, as shown:

```
/**
 * Returns a socket connected to a ServerSocket on the named host,
 * at the given port.
 * @param host  the server host
 * @param port  the server port
 * @param props  Properties passed in through the connection
 * @returns Socket
 * @exception IOException, UnknownHostException
 */
public java.net.Socket createSocket(String host, int port, Properties props)
   throws IOException, UnknownHostException;
```

Passing in properties allows instances of SybSocketFactory to use connection properties to implement an intelligent socket.

When you implement SybSocketFactory to produce a socket, the same application code can use different kinds of sockets by passing the different kinds of factories or pseudo-factories that create sockets to the application.

You can customize factories with parameters used in socket construction. For example, you can customize factories to return sockets with different networking timeouts or security parameters already configured. The sockets returned to the application can be subclasses of java.net.Socket to directly expose new APIs for features such as compression, security, record marking, statistics collection, or firewall tunnelling (javax.net.SocketFactory).

Note SybSocketFactory is intended to be an overly simplified javax.net.SocketFactory, enabling applications to bridge from java.net.* to javax.net.*

❖ **Using a custom socket with jConnect**

1. Provide a Java class that implements com.sybase.jdbcx.SybSocketFactory. See "Creating and configuring a custom socket" on page 113.
2. Set the SYBSOCKET_FACTORY connection property so that jConnect can use your implementation to obtain a socket.

   To use a custom socket with jConnect, set the SYBSOCKET_FACTORY connection property to one of the following:

   - The class name that implements com.sybase.jdbcx.SybSocketFactory
   - DEFAULT (this instantiates a new java.net.Socket)

   See "Connection properties" on page 10 for instructions on how to set SYBSOCKET_FACTORY.

Creating and configuring a custom socket

Once jConnect obtains a custom socket, it uses the socket to connect to a server. Any configuration of the socket must be completed before jConnect obtains it.

This section explains how to plug in an SSL socket implementation, such as javax.net.ssl.SSLSocket, with jConnect.

The following example shows how an implementation of SSL can create an instance of SSLSocket, configure it, and then return it. In the example, the MySSLSocketFactory class implements SybSocketFactory and extends javax.net.ssl.SSLSocketFactory to implement SSL. It contains two createSocket methods—one for SSLSocketFactory and one for SybSocketFactory—that:

- Create an SSL socket
- Invoke SSLSocket.setEnableCipherSuites to specify the cipher suites available for encryption
- Return the socket to be used by jConnect

Example

```
public class MySSLSocketFactory extends SSLSocketFactory
   implements SybSocketFactory
 {
 /**
 * Create a socket, set the cipher suites it can use, return
 * the socket.
 * Demonstrates how cither suites could be hard-coded into the
 * implementation.
 *
 * See javax.net.SSLSocketFactory#createSocket
 */
public Socket createSocket(String host, int port)
   throws IOException, UnknownHostException
 {
   // Prepare an array containing the cipher suites that are to
   // be enabled.
   String enableThese[] =
   {
       "SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA",
       "SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5",
       "SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA"
   }
   ;
   Socket s =
       SSLSocketFactory.getDefault().createSocket(host, port);
   ((SSLSocket)s).setEnabledCipherSuites(enableThese);
   return s;
 }
/**
 * Return an SSLSocket.
 * Demonstrates how to set cipher suites based on connection
 * properties like:
 * Properties _props = new Properties();
 * Set other url, password, etc. properties.
 * _props.put(("CIPHER_SUITES_1",
 *    "SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA");
 * _props.put("CIPHER_SUITES_2",
 *    "SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5");
 * _props.put("CIPHER_SUITES_3",
 *     "SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA");
 * _conn = _driver.getConnection(url, _props);
 *
 * See com.sybase.jdbcx.SybSocketFactory#createSocket
 */
public Socket createSocket(String host, int port,
   Properties props)
   throws IOException, UnknownHostException
 {
   // check to see if cipher suites are set in the connection
   // properites
   Vector cipherSuites = new Vector();
   String cipherSuiteVal = null;
   int cipherIndex = 1;
   do
   {
       if((cipherSuiteVal = props.getProperty("CIPHER_SUITES_"
           + cipherIndex++)) == null)
       {
           if(cipherIndex <= 2)
           {
               // No cipher suites available
               // return what the object considers its default
               // SSLSocket, with cipher suites enabled.
               return createSocket(host, port);
           }
           else
           {
               // we have at least one cipher suite to enable
               // per request on the connection
               break;
           }
           else
           }
               // add to the cipher suit Vector, so that
               // we may enable them together
               cipherSuites.addElement(cipherSuiteVal);
           }
```

```
        }
        while(true);
       // lets you create a String[] out of the created vector
        String enableThese[] = new String[cipherSuites.size()];
        cipherSuites.copyInto(enableThese);
        Socket s =
            SSLSocketFactory.getDefault().createSocket
              (host, port);
        // enable the cipher suites
        ((SSLSocket)s).setEnabledCipherSuites(enableThese);
       // return the SSLSocket
        return s;
    }
   // other methods
  }
```

Because jConnect requires no information about the kind of socket it is, you must complete any configuration before you return a socket.

For additional information, see:

- *EncryptASE.java* – located in the *sample2* subdirectory of your jConnect installation, this sample shows you how to use the SybSocketFactory interface with jConnect applications.
- *MySSLSocketFactoryASE.java* – also located in the *sample2* subdirectory of your jConnect installation, this is a sample implementation of the SybSocketFactory interface that you can plug in to your application and use.

Kerberos

Kerberos is a network authentication protocol that uses encryption for authentication of client-server applications. Kerberos provides advantages for users and system administrators, including the following:

- A Kerberos database can serve as a centralized storehouse for users.
- Kerberos facilitates the single-sign-on (SSO) environment, in which a user system login provides the credentials necessary to access a database.
- Kerberos is an IETF standard. Interoperability is possible between different implementations of Kerberos.

Configuring jConnect applications for Kerberos

Before attempting to configure Kerberos for jConnect, make sure you have the following:

- JDK 1.6 or later
- A Java Generic Security Services (GSS) Manager:
  - a The default Sun GSS Manager, which is part of the JDK, or
  - b Wedgetail JCSI Kerberos version 2.6 or later, or
  - c CyberSafe TrustBroker Application Security Runtime Library version 3.1.0 or later, or
  - d A GSS Manager implementation from another vendor.
- A KDC that is supported and interoperable at the server side with your GSS library and at the client side with your GSSManager.

To enable Kerberos login with jConnect, use the following procedure.

❖ **Configuring Kerberos for jConnect**

1. Set the REQUEST_KERBEROS_SESSION property to "true."
2. Set the SERVICE_PRINCIPAL_NAME property to the name that your Adaptive Server Enterprise is running under. In general, this is the name set with the -s option when the server is started. The service principal name must also be registered with the KDC. If you do not set a value for the SERVICE_PRINCIPAL_NAME property, jConnect defaults to using the host name of the client machine.
3. Optionally, set the GSSMANAGER_CLASS property.

For more information on the REQUEST_KERBEROS_SESSION and SERVICE_PRINCIPAL_NAME properties, see Chapter 2, “Programming Information.” For more information on the GSSMANAGER_CLASS property, see “GSSMANAGER_CLASS connection property.”

GSSMANAGER_CLASS connection property

When using Kerberos, jConnect relies on several Java classes that implement the Generic Security Services (GSS) API. Much of this functionality is provided by the org.ietf.jgss.GSSManager class.

Vendor implementations

Java allows vendors to provide their own implementations of the GSSManager class. Examples of vendor-supplied GSSManager implementations are those provided by Wedgetail Communications and CyberSafe Limited. Users can configure a vendor-written GSSManager class to work in a particular Kerberos environment. Vendor-supplied GSSManager classes may also offer more interoperability with Windows than the standard Java GSSManager class provides.

Before using a vendor-supplied implementation of GSSManager, be sure to read the vendor documentation. Vendors use system property settings other than the standard Java system properties used for Kerberos and may locate realm names and Key Distribution Center (KDC) entries without using configuration files.

Setting GSSMANAGER_CLASS

You can use a vendor implementation of GSSManager with jConnect by setting the GSSMANAGER_CLASS connection property. There are two ways to set this property:

- Create an instance of GSSManager, and set this instance as the value of the GSSMANAGER_CLASS property.
- Set the value of the GSSMANAGER_CLASS property as a string specifying the fully qualified class name of the GSSManager object. jConnect uses this string to call `Class.forName().newInstance()` and casts the returned object as a GSSManager class.

In either case, the application CLASSPATH variable must include the location of the classes and *.jar* files for the vendor implementation.

Note If you do not set the GSSMANAGER_CLASS connection property, jConnect uses the org.ietf.jgss.GSSManager.getInstance method to load the default Java GSSManager implementation.

When you use the GSSMANAGER_CLASS connection property to pass in a fully qualified class name, jConnect calls the no-argument constructor for the GSSManager. This instantiates a GSSManager that is in the default configuration for the vendor implementation, so you do not have control over the exact configuration of the GSSManager object. If you create your own instance of GSSManager, you can use constructor arguments to set configuration options.

How jConnect uses GSSMANAGER_CLASS

First, jConnect checks the value of GSSMANAGER_CLASS for a GSSManager class object to use in Kerberos authentication.

If the value of GSSMANAGER_CLASS has been set to a string instead of a class object, jConnect uses the string to create an instance of the specified class and uses the new instance in Kerberos authentication.

If the value of GSSMANAGER_CLASS is set to something that is neither a GSSManager class object nor a string, or if jConnect encounters a ClassCastException, jConnect throws a SQLException indicating the problem.

Examples

The following examples illustrate how to create your own instance of GSSManager and how to let jConnect create a GSSManager object when the GSSMANAGER_CLASS connection property is set to a fully qualified class name. Both examples use the Wedgetail GSSManager.

❖ **Example: Creating your own instance of GSSManager**

1 Instantiate a GSSManager in your application code. For example:

```
GSSManager gssMan =  new com.dstc.security.kerberos.gssapi.GSSManager();
```

This example uses the default constructor with no arguments. You can use other vendor-supplied constructors, which allow you to set various configuration options.

2 Pass the new GSSManager instance into the GSSMANAGER_CLASS connection property. For example:

```
Properties props = new Properties();
props.put("GSSMANAGER_CLASS", gssMan);
```

3 Use these connection properties, including GSSMANAGER_CLASS, in your connection. For example:

```
Connection conn = DriverManager.getConnection (url, props);
```

❖ **Example: Passing a string to GSSMANAGER_CLASS**

1 In your application code, create a string specifying the fully qualified class name of the GSSManager object. For example:

```
String gssManClass = "com.dstc.security.kerberos.gssapi.GSSManager";
```

2 Pass the string to the GSSMANAGER_CLASS connection property. For example:

```
Properties props = new Properties();
props.put("GSSMANAGER_CLASS", gssManClass);
```

3 Use these connection properties, including GSSMANAGER_CLASS, in your connection. For example,

```
Connection conn = DriverManager.getConnection (url, props);
```

Setting up the Kerberos environment

This section provides suggestions for setting up the environment to use jConnect with three different implementations of Kerberos:

- CyberSafe
- MIT
- Microsoft Active Directory

Note Before reading this section, see the Kerberos white paper at http://www.sybase.com/detail?id=1029260

CyberSafe

Encryption keys

Specify a Data Encryption Standard (DES) key when creating a principal to be used by Java in the CyberSafe KDC. The Java reference implementation does not support Triple Data Encryption Standard (3DES) keys.

Note You can use 3DES keys if you are using CyberSafe GSSManager with a CyberSafe KDC and have set the GSSMANAGER_CLASS property.

Address mapping and realm information

CyberSafe Kerberos does not use a *krb5.conf* configuration file. By default, CyberSafe uses DNS records to locate KDC address mapping and realm information. Alternately, CyberSafe locates KDC address mapping and realm information in the *krb.conf* and *krb.realms* files, respectively. Refer to CyberSafe documentation for more information.

If you are using the standard Java GSSManager implementation, you must still create a *krb5.conf* file for use by Java. The CyberSafe *krb.conf* file is formatted differently from the *krb5.conf* file. Create a *krb5.conf* file as specified in the Sun manual pages or in the MIT documentation. You do not need a *krb5.conf* file if using the CyberSafe GSSManager.

For examples of the *krb5.conf* file, see white paper on setting up Kerberos. The URL for this document can be found in the *jConnect for JDBC Release Bulletin*.

Solaris

When using CyberSafe client libraries on Solaris, make sure your library search path includes the CyberSafe libraries before any other Kerberos libraries.

MIT

Encryption keys

Specify a DES key when creating a principal to be used by Java in the MIT KDC. The Java reference implementation does not support 3DES keys.

If you plan to use only the standard Java GSSManager implementation, specify an encryption key of type `des-cbc-crc` or `des-cbc-md5`. Specify the encryption type as follows:

```
des-cbc-crc:normal
```

Here `normal` is the type of key salt. It may be possible to use other salt types.

Note If you are using Wedgetail GSSManager, you can create principals in an MIT KDC of type `des3-cbc-sha1-kd`.

Microsoft Active Directory

User accounts and service principal

Make sure that you have set up accounts in Active Directory for your user principals (the users) and service principals (the accounts that represent your database servers). Your user principals and service principals should both be created as 'Users' within Active Directory.

Encryption

If you intend to use the Java reference GSS Manager implementation, you must use DES encryption for both user and service principals.

❖ **Setting DES encryption**

1. Right-click on the specific user principal or service principal name in the Active Directory Users list.
2. Select Properties.
3. Click the Account tab. The Account Options list appears.
4. For both the user principal and service principal, specify that DES encryption types should be used.

Client machines

If you plan to use the Java reference implementation to set up an SSO environment, you may need to modify the Windows Registry according to instructions specified at the Microsoft support site at http://support.microsoft.com/.

Configuration file

On Windows, the Kerberos configuration file is called *krb5.ini*. Java looks for *krb5.ini* by default at *C:\WINNT\krb5.ini*. Java allows you to specify the location of this file. The format of *krb5.ini* is identical to that of *krb5.conf*.

For examples of the *krb5.conf* file, see white paper on setting up Kerberos. The URL for this document can be found in the *jConnect for JDBC Release Bulletin*.

For more information on Kerberos for Microsoft Active Directory, see the Microsoft Developer Network at http://msdn.microsoft.com.

Sample applications

The following two commented code samples are provided in the *jConnect-7_0/sample2* directory to illustrate how to establish a Kerberos connection to Adaptive Server Enterprise:

- *ConnectKerberos.java* – A simple Kerberos login to Adaptive Server Enterprise
- *ConnectKerberosJAAS.java* – A more detailed sample showing how a Kerberos login might be implemented within application-server code

ConnectKerberos.java

To run the *ConnectKerberos.java* sample application, use the following procedure.

❖ **Running *ConnectKerberos.java***

1 Make sure your machine has valid Kerberos credentials. This task varies depending on your machine and environment.

Windows – You can establish Kerberos credentials for a machine in an Active Directory environment by successfully logging in using Kerberos authentication.

UNIX or Linux – You can establish Kerberos credentials for a UNIX or Linux machine using the kinit utility for your Kerberos client. If you do not obtain an initial credential using kinit, you are prompted for a user name and password when you attempt to run the sample application.

Note The Sun JDK can use only the DES_CBC_MD5 and DES_CBC_CRC encryption types. You may be able to use other encryption types by using third-party software and setting the GSSMANAGER_CLASS property.

---

2 Determine the location of the credentials for your machine.

Windows – For a machine running in an Active Directory environment, Kerberos credentials are stored in an in-memory ticket cache.

UNIX or Linux – For a UNIX or Linux machine using the Sun Java, CyberSafe, Solaris, or MIT implementations of Kerberos, kinit places credentials by default in */tmp/krb5cc_{user_id_number}*, where *{user_id_number}* is unique to your user name.

If the credentials are placed elsewhere, you must specify that location in the *sample2/exampleLogin.conf* file by setting the ticketCache property.

3 Specify to the Java reference implementation the default realm and host name of the KDC machine. Java may obtain this information from the *krb5.conf* or *krb5.ini* configuration files or from Java System properties. If you use a vendor GSS Manager implementation, that implementation may obtain host and realm information from DNS SRV records.

Sybase recommends that you use a Kerberos configuration file, which allows for more control of the Kerberos environment, including the ability to specify to Java the type of encryption to request during authentication.

Note On Linux, the Java reference implementation looks for the Kerberos configuration file in */etc/krb5.conf*.

If you do not use a Kerberos configuration file, and your Kerberos configuration is not set up to use DNS SRV records, you can specify the realm and KDC using the java.security.krb5.realm and java.security.krb5.kdc system properties.

4 Edit *ConnectKerberos.java* so that the connection URL points to your database.

5 Compile *ConnectKerberos.java*.

Ensure that you are using JDK version 1.6 or later. Read through the source code comments, and ensure the *jconn4.jar* from your jConnect installation is specified in your CLASSPATH environment variable.

6 Execute ConnectKerberos.class:

```
java ConnectKerberos
```

Ensure that you are using java version 1.6 executable. The sample application output explains that a successful connection has been established and executes the following SQL:

```
select 1
```

- *To execute the sample without using a Kerberos configuration file*, use the following command:

  ```
  java -Djava.security.krb5.realm=your_realm
  -Djava.security.krb5.kdc=your_kdc
  ConnectKerberos
  ```

  where *your_realm* is your default realm, and *your_kdc* is your KDC.

- *If necessary, you can run the sample application in debug mode* to see debug output from the Java Kerberos layer:

  ```
  java -Dsun.security.krb5.debug=true
  ConnectKerberos
  ```

You can also make a Kerberos connection using IsqlApp, the Java version of isql, located in the *jConnect-7_0/classes* directory:

```
java IsqlApp -S jdbc:sybase:Tds:hostName:portNum
-K service_principal_name
-F path_to_JAAS_login_module_config_file
```

For details on using IsqlApp, see

The *krb5.conf* configuration file

The following are examples of *krb5.conf* files.

CyberSafe or MIT KDC

This is an example of a *krb5.conf* file a client might use with a CyberSafe or MIT KDC.

```
# Please note that customers must alter the
```

```
# default_realm, [realms] and [doamin_realm]
# information to reflect their Kerberos environment.
# Customers should *not* attempt to use this file as is.
#

[libdefaults]
        default_realm = ASE
        default_tgs_enctypes = des-cbc-crc
        default_tkt_enctypes = des-cbc-crc
        kdc_req_checksum_type = 2
        ccache_type = 2

[realms]

        ASE = {
            kdc = kdchost
            admin_server = kdchost
        }

[domain_realm]
        .sybase.com = ASE
        sybase.com = ASE

[logging]
        default = FILE:/var/krb5/kdc.log
        kdc = FILE:/var/krb5/kdc.log
 kdc_rotate = {

# How often to rotate kdc.log. Logs will get rotated
# no more often than the period, and less often if the
# KDC is not used frequently.

  period = 1d

# how many versions of kdc.log to keep around
# (kdc.log.0, kdc.log.1, ...)

  versions = 10
 }

[appdefaults]
 kinit = {
  renewable = true
  forwardable= true
 }
```

Active Directory KDC

This is an example of a *krb5.conf* file a client might use with Active Directory as the KDC.

```
# Please note that customers must alter the
# default_realm, [realms] and [domain_realm]
# information to reflect their Kerberos environment.
# Customers should *not* attempt to use this file as is.
#

[libdefaults]
        default_realm = W2K.SYBASE.COM
        default_tgs_enctypes = des-cbc-crc
        default_tkt_enctypes = des-cbc-crc
        kdc_req_checksum_type = 2
        ccache_type = 2

[realms]

        W2K.SYBASE.COM = {
           kdc = 1.2.3.4:88
           admin_server = adserver
         }

[domain_realm]
        .sybase.com = W2K.SYBASE.COM
        sybase.com = W2K.SYBASE.COM

[logging]
        default = FILE:/var/krb5/kdc.log
        kdc = FILE:/var/krb5/kdc.log
 kdc_rotate = {

# How often to rotate kdc.log. Logs will get rotated no
# more often than the period, and less often if the KDC
# is not used frequently.

  period = 1d

# how many versions of kdc.log to keep around
# (kdc.log.0, kdc.log.1, ...)

  versions = 10
 }

[appdefaults]
```

```
kinit = {
 renewable = true
 forwardable= true
}
```

Interoperability

Table 3-1 shows combinations of KDCs, GSS libraries, and platforms on which Sybase has successfully established a connection to Adaptive Server Enterprise. The absence of any particular combination does not indicate that a connection cannot be established with that combination. You can find the most recent status at the jConnect for JDBC Web site at http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect:

Table 3-1: Interoperability combinations

Client platform	KDC	GSSManager	GSS C libraries[a]	ASE platform
Solaris 8[b]	CyberSafe	Java GSS	CyberSafe	Solaris 8
Solaris 8	Active Directory[c]	Java GSS	CyberSafe	Solaris 8
Solaris 8	MIT	Java GSS	CyberSafe	Solaris 8
Solaris 8	MIT	Wedgetail GSS[d]	MIT	Solaris 8
Solaris 8	CyberSafe	Wedgetail GSS[e]	CyberSafe	Solaris 8
Windows 2000	Active Directory	Java GSS	CyberSafe	Solaris 8
Windows XP	Active Directory	Java GSS[f]	CyberSafe	Solaris 8

a. These are the libraries that Adaptive Server Enterprise is using for its GSS functionality.
b. All Solaris 8 platforms in this table are 32-bit.
c. All Active Directory entries in the table refer to an Active Directory server running on Windows 2000. For Kerberos interoperability, Active Directory users must be set to “Use DES encryption types for this account.”
d. Used Wedgetail JCSI Kerberos 2.6. The encryption type was 3DES.
e. Used Wedgetail JCSI Kerberos 2.6. The encryption type was DES.
f. Java 1.4.x has a bug which requires that clients use **`System.setProperty("os.name", "Windows 2000");`** to ensure that Java can find the in-memory credential on Windows XP clients.

Sybase recommends that you use the latest versions of these libraries. Contact the vendor if you intend to use older versions or if you have problems with non-Sybase products.

Encryption types

The standard Java GSS implementation provided by Sun supports only DES encryption. If you intend to use the 3DES, RC4-HMAC, AES-256, or AES-128 encryption standards, you must use the CyberSafe or Wedgetail GSSManagers.

Refer to the respective documentation for more information about Wedgetail and CyberSafe.

Troubleshooting

This section documents issues to consider when troubleshooting Kerberos security.

Kerberos

Consider the following when troubleshooting problems with Kerberos security:

- The Java reference implementation supports only the DES encryption type. You must configure your Active Directory and KDC principals to use DES encryption.
- The value of the SERVICE_PRINCIPAL_NAME property must be set to the same name you specify with the -s option when you start your data server.
- Check the *krb5.conf* and *krb5.ini* files. For CyberSafe clients, check the *krb.conf* and *krb.realms* files or DNS SRV records.
- You can set the debug property to "true" in the JAAS login configuration file.
- You can set the debug property to "true" at the command line:

  ```
  -Dsun.security.krb5.debug=true
  ```
- The JAAS login configuration file provides several options that you can set for your particular needs. For information about JAAS and the Java GSS API, refer to:
  - JAAS login configuration file at http://java.sun.com/j2se/1.4.2/docs/guide/security/jgss/tutorials/LoginConfigFile.html
  - Class Krb5LoginModule at http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/spec/com/sun/security/auth/module/Krb5LoginModule.html
  - Troubleshooting JGSS at http://java.sun.com/j2se/1.4.2/docs/guide/security/jgss/tutorials/Troubleshooting.html

Related documents

These documents provide additional information on Kerberos security.

- Java tutorial on JAAS and the Java GSS API at http://java.sun.com/j2se/1.4.2/docs/guide/security/jgss/tutorials/index.html
- MIT Kerberos documentation and download site at http://web.mit.edu/kerberos/www/index.html
- CyberSafe Limited at http://www.cybersafe.ltd.uk
- CyberSafe Limited document on Windows-Kerberos interoperability at http://www.cybersafe.ltd.uk/docs_cybersafe/Kerberos%20Interoperability%20-%20Microsoft%20W2k%20&%20ActiveTRUST.pdf
- Description of how Windows implements authentication, including information about Active Directory Kerberos at http://www.windowsitlibrary.com/Content/617/06/1.html
- Kerberos RFC 1510 at http://www.linuxdig.com/rfc/individual/1510.php

CHAPTER 4

Troubleshooting

This chapter describes solutions and workarounds for problems you might encounter when using jConnect.

Debugging with jConnect

jConnect includes a Debug class that contains a set of debugging functions. The Debug methods include a variety of assert, trace, and timer functions that let you define the scope of the debugging process and the output destination for the debugging results.

The jConnect installation also includes a complete set of debug-enabled classes. These classes are located in the *devclasses* subdirectory under your jConnect installation directory. For debugging purposes, you must redirect your CLASSPATH environment variable to reference the debug mode runtime classes (*devclasses/jconn4d.jar*), rather than the standard jConnect *classes* directory. You can also do this by explicitly providing a -classpath argument to the java command when you run a Java program.

Obtaining an instance of the Debug class

To use the jConnect debugging feature, your application must import the Debug interface and obtain an instance of the Debug class by calling the getDebug method on the SybDriver class.

```
import com.sybase.jdbcx.Debug;
```

```
//
...
SybDriver sybDriver = (SybDriver)
Class.forName("com.sybase.jdbc4.jdbc.SybDriver").newInstance();
Debug sybdebug = sybDriver.getDebug();
...
```

Turning on debugging in your application

To use the debug method on the Debug object to turn on debugging within your application, add this call:

```
sybdebug.debug(true, [classes], [printstream]);
```

The *classes* parameter is a string that lists the specific classes you want to debug, separated by colons. For example:

```
sybdebug.debug(true,"MyClass")
```

and

```
sybdebug.debug(true,"MyClass:YourClass")
```

Using “STATIC” in the class string turns on debugging for all static methods in jConnect in addition to the designated classes. For example:

```
sybdebug.debug(true,"STATIC:MyClass")
```

You can specify “ALL” to turn on debugging for all classes. For example:

```
sybdebug.debug(true,"ALL");
```

The *printstream* parameter is optional. If you do not specify a printstream, the debug output goes to the output file you specified with DriverManager.setLogStream.

Turning off debugging in your application

To turn off debugging, add this call:

```
sybdebug.debug(false);
```

Setting the CLASSPATH for debugging

Before you run your debug-enabled application, replace the optimized jConnect driver jar file *jconn4.jar* with the debug version *jconn4d.jar*, which you can find in the *devclasses* subdirectory under your jConnect installation directory.

If you use the CLASSPATH environment variable:

- For UNIX, replace *$JDBC_HOME/classes/jconn4.jar* with *$JDBC_HOME/devclasses/jconn4d.jar.*
- For Windows, replace *%JDBC_HOME%\classes\jconn4.jar* with *%JDBC_HOME%\devclasses\jconn4d.jar.*

Using the Debug methods

To customize the debugging process, you can add calls to other Debug methods.

In these methods, the first (object) parameter is usually this to specify the calling object. If any of these methods are static, use null for the object parameter.

- println

  Use this method to define the message to print in the output log if debugging is enabled and the object is included in the list of classes to debug. The debug output goes to the file you specified with *sybdebug*.debug.

  The syntax is:

  ```
  sybdebug.println(object,message string);
  ```

  For example:

  ```
  sybdebug.println(this,"Query: "+ query);
  ```

  produces a message similar to this in the output log:

  ```
  myApp(thread[x,y,z]): Query: select * from authors
  ```

- assert

  Use this method to assert a condition and throw a runtime exception when the condition is not met. You can also define the message to print in the output log if the condition is not met. The syntax is:

```
sybdebug.assert(object,boolean condition,message
   string);
```

For example:

```
sybdebug.assert(this,amount<=buf.length,amount+"
   too big!");
```

produces a message similar to this in the output log if “amount” exceeds the value of buf.length:

```
java.lang.RuntimeException:myApp(thread[x,y,z]):
Assertion failed: 513 too big!
at jdbc.sybase.utils.sybdebug.assert(
sybdebug.java:338)
at myApp.myCall(myApp.java:xxx)
at .... more stack:
```

- startTimer
  stopTimer

  Use these methods to start and stop a timer that measures the milliseconds that elapse during an event. The method keeps one timer per object, and one for all static methods. The syntax to start the timer is:

```
sybdebug.startTimer(object);
```

The syntax to stop the timer is:

```
sybdebug.stopTimer(object,message string);
```

For example:

```
sybdebug.startTimer(this);
stmt.executeQuery(query);
sybdebug.stopTimer(this,"executeQuery");
```

produces a message similar to this in the output log:

```
myApp(thread[x,y,z]):executeQuery elapsed time =
  25ms
```

Capturing TDS communication

Tabular Data Stream (TDS) is the Sybase proprietary protocol for handling communication between a client application and Adaptive Server. jConnect includes a PROTOCOL_CAPTURE connection property that allows you to capture raw TDS packets to a file.

If you are having problems with an application that you cannot resolve within either the application or the server, you can use PROTOCOL_CAPTURE to capture the communication between the client and the server in a file. You can then send the file, which contains binary data and is not directly interpretable, to Sybase Technical Support for analysis.

Note You can use the Ribo utility to capture, translate, and display the protocol stream flowing between the client and the server. You have the option to install Ribo when you install the Sybase Software Developer’s Kit.

PROTOCOL_CAPTURE connection property

Use the PROTOCOL_CAPTURE connection property to specify a file for receiving the TDS packets exchanged between an application and an Adaptive Server. PROTOCOL_CAPTURE takes effect immediately so that TDS packets exchanged during connection establishment are written to the specified file. All packets continue to be written to the file until Capture.pause is executed or the session is closed.

The following example shows the use of PROTOCOL_CAPTURE to send TDS data to the file *tds_data*:

```
...
props.put("PROTOCOL_CAPTURE", "tds_data")
Connection conn = DriverManager.getConnection(url,
props);
```

where *url* is the connection URL, and *props* is a Properties object for specifying connection properties.

pause and *resume* methods in the Capture class

The Capture class is contained in the com.sybase.jdbcx package. It contains two public methods:

- public void pause
- public void resume

Capture.pause stops the capture of raw TDS packets into a file; Capture.resume restarts the capture.

The TDS capture file for an entire session can become very large. If you want to limit the size of the capture file, and you know where in an application you want to capture TDS data, you can perform the following.

❖ **To limit the size of the capture file**

1. Immediately after you have established a connection, get the Capture object for the connection and use the pause method to stop capturing TDS data:

   ```
   Capture cap = ((SybConnection)conn).getCapture();
    cap.pause();
   ```

2. Place cap.resume just before the point where you want to start capturing TDS data.
3. Place cap.pause just after the point where you want to stop capturing data.

Resolving connection errors

This section addresses problems that may arise when you are trying to establish a connection or start a gateway.

Gateway connection refused

```
Gateway connection refused:
HTTP/1.0 502 Bad Gateway|Restart Connection
```

This error message indicates that something is wrong with the *hostname* or *port#* used to connect to your Adaptive Server. Check the [query] entry in *$SYBASE/interfaces* (UNIX) or in *%SYBASE%\ini\sql.ini* (Windows).

If the problem persists after you have verified the *hostname* and *port#*, you can learn more by starting the HTTP server using the "verbose" system property.

For Windows, go to a DOS prompt and enter:

```
httpd -Dverbose=1 > filename
```

For UNIX, enter:

```
sh httpd.sh -Dverbose=1 > filename &
```

where *filename is* the debug messages output file.

Your Web server probably does not support the connect method. Applets can connect only to the host from which they were downloaded.

The HTTP gateway and your Web server must run on the same host. In this scenario, your applet can connect to the same machine/host through the port controlled by the HTTP gateway, which routes the request to the appropriate database.

To see how this is accomplished, review the source of *Isql.java* and *gateway.html* in the *sample2* subdirectory under the jConnect installation directory. Search for “proxy.”

Managing memory in jConnect applications

The following situations and their solutions may be helpful if you notice increased memory use in jConnect applications.

- In jConnect applications, you should explicitly close all Statement objects and subclasses (for example, PreparedStatement, CallableStatement) after their last use to prevent statements from accumulating in memory. Closing only the ResultSet is not sufficient.

  For example, the following statement causes problems:

  ```
  ResultSet rs = _conn.prepareCall(_query).execute();
  ...
  rs.close();
  ```

  Instead, use the following:

  ```
  PreparedStatement ps = _conn.prepareCall(_query);
  ResultSet rs = ps.executeQuery();
  ...
  rs.close();
  ps.close();
  ```

- Native support for Scrollable or Updatable Scrollable cursors may not be available depending on the version of Adaptive Server or SQL Anywhere database you are connecting to. To support scrollable or updatable scrollable cursors when not supported natively by the backend server, jConnect caches the row data on demand, on the client, on each call to ResultSet.next. However, when the end of the result set is reached, the entire result set is stored in client memory. Because this may cause a performance strain, Sybase recommends that you use TYPE_SCROLL_INSENSITIVE result sets only when the result set is reasonably small. With this release, jConnect determines if the Adaptive Server connection supports native scrollable cursor functionality and uses it instead of client-side caching. As a result, most applications can expect significant performance gain in accessing out-of-order rows and reduction in client-side memory requirements.

Resolving stored procedure errors

This section addresses problems that may arise when you are trying to use jConnect and stored procedures.

RPC returns fewer output parameters than registered

```
SQLState: JZ0SG - An RPC did not return as many output
parameters as the application had registered for it.
```

This error occurs if you call CallableStatement.registerOutParam for more parameters than you have declared as “OUTPUT” parameters in the stored procedure. Make sure that you have declared all of the appropriate parameters as “OUTPUT.” Look at the line of code that reads:

```
create procedure yourproc (@p1 int OUTPUT, ...
```

Note If you receive this error while using SQL Anywhere, upgrade to SQL Anywhere version 5.5.04 or later.

Fetch/state errors when output parameters are returned

If a query does not return row data, then it should use the CallableStatement.executeUpdate or execute methods rather than the executeQuery method.

As required by the JDBC standards, jConnect throws a SQL exception if executeQuery has no result sets.

Stored procedure executed in unchained transaction mode

```
Sybase Error 7713 - Stored Procedure can only be
executed in unchained transaction mode.
```

This error occurs when JDBC attempts to put the connection in autocommit(true) mode. The application can change the connection to chained mode using Connection.setAutoCommit(false) or by using a “set chained on” language command. This error occurs if the stored procedure was not created in a compatible mode.

To fix the problem, use:

sp_procxmode *procedure_name,"*anymode"

Resolving a custom socket implementation error

You may receive an exception similar to the following while trying to set up an SSL socket when calling sun.security.ssl.SSLSocketImpl.setEnabledCipherSuites:

```
java.lang.IllegalArgumentException:
 SSL_SH_anon_EXPORT_WITH_RC4_40_MDS
```

Verify that the SSL libraries are in the system library path.

CHAPTER 5 Performance and Tuning

This chapter describes how to fine-tune or improve performance when working with jConnect.

Improving jConnect performance

There are a number of ways to optimize the performance of an application using jConnect:

- Use TextPointer.sendData methods to send text and image data to an Adaptive Server database. See "Updating image data in the database" on page 62.
- Create precompiled PreparedStatement objects for dynamic SQL statements that are used repeatedly during a session. See "Performance tuning for prepared statements in dynamic SQL" on page 144.
- Use batch updates to improve performance by reducing network traffic; specifically, all queries are sent to the server in one group and all responses returned to the client are sent in one group. See "Support for batch updates" on page 59.
- For sessions that are likely to move image data, large row sets, and lengthy text data, use the PACKETSIZE connection property to set the maximum feasible packet size.
- For TDS-tunneled HTTP, set the maximum TDS packet size and configure your Web server to support the HTTP1.1 Keep-Alive feature. Also, set the *SkipDoneProc* servlet argument to "true."

- Use protocol cursors, the default setting of the LANGUAGE_CURSOR connection property. See “LANGUAGE_CURSOR connection property” on page 150 for more information.
- If you use TYPE_SCROLL_INSENSITIVE result sets, use them only when the result set is reasonably small. See “Using TYPE_SCROLL_INSENSITIVE result sets in jConnect” on page 55 for more information.

Additional considerations for improving performance are described in the following sections.

BigDecimal rescaling

The JDBC 1.0 specification requires a scale factor with getBigDecimal. Then, when a BigDecimal object is returned from the server, it must be rescaled using the original scale factor you used with getBigDecimal.

To eliminate the processing time required for rescaling, use the JDBC 2.0 getBigDecimal method, which jConnect implements in the SybResultSet class and does not require a *scale* value:

```
public BigDecimal getBigDecimal(int columnIndex)
  throws SQLException
```

For example:

```
SybResultSet rs =
   (SybResultSet)stmt.executeQuery("SELECT
   numeric_column from T1");
 while (rs.next())
 {
   BigDecimal bd rs.getBigDecimal(
     "numeric_column");
      ...
 }
```

REPEAT_READ connection property

You can improve performance on retrieving a result set from the database if you set the REPEAT_READ connection property to "false." However, when REPEAT_READ is "false:"

- You must read column values in order, according to column index. This is difficult if you want to access columns by name rather than column number.
- You cannot read a column value in a row more than once.

SunIoConverter character-set conversion

If you are using multibyte character sets and need to improve driver performance, you can use the SunIoConverter class provided with the jConnect samples. This converter is based on the sun.io classes provided by the Java Software Division of Sun Microsystems, Inc.

The SunIoConverter class is not a pure Java implementation of the character-set converter feature and, therefore, is not integrated with the standard jConnect product. However, Sybase has provided this converter class for your reference, and you can use it with the jConnect driver to improve character-set conversion performance.

Note Based on Sybase testing, the SunIoConverter class improved performance on all VMs on which it was tested. However, the Java Software Division of Sun Microsystems, Inc. reserves the right to remove or change the sun.io classes with future releases of the JDK. Therefore, this SunIoConverter class may not be compatible with later JDK releases.

To use the SunIoConverter class, you must install the jConnect sample applications. Once the samples are installed, set the CHARSET_CONVERTER_CLASS connection property to reference the SunIoConverter class in the *sample2* subdirectory under your jConnect installation directory. See the *Sybase jConnect for JDBC Installation Guide* for complete instructions on installing jConnect and its components, including the sample applications.

If you are using a database with its default character set as iso_1, or if you are using only the first 7 bits of ASCII, you can gain significant performance benefits by using the TruncationConverter. See “jConnect character-set converters” on page 33.

Performance tuning for prepared statements in dynamic SQL

In Embedded SQL™, dynamic statements are SQL statements that need to be compiled at runtime, rather than statically. Typically, dynamic statements contain input parameters, although this is not a requirement. In SQL, the prepare command is used to precompile a dynamic statement and save it so that it can be executed repeatedly without being recompiled during a session.

If a statement is used multiple times in a session, precompiling it provides better performance than sending it to the database and compiling it for each use. The more complex the statement, the greater the performance benefit.

If a statement is likely to be used only a few times, precompiling it may be inefficient because of the overhead involved in precompiling, saving, and later deallocating it in the database.

Precompiling a dynamic SQL statement for execution and saving it in memory uses time and resources. If a statement is not likely to be used multiple times during a session, the costs of doing a database prepare may outweigh its benefits. Another consideration is that once a dynamic SQL statement is prepared in the database, it is very similar to a stored procedure. In some cases, it may be preferable to create stored procedures and have them reside on the server, rather than defining prepared statements in the application. This is discussed under "Choosing prepared statements and stored procedures" on page 145.

You can use jConnect to optimize the performance of dynamic SQL statements on a Sybase database as follows:

- Create PreparedStatement objects that contain precompiled statements in cases where a statement is likely to be executed several times in a session.
- Create PreparedStatement objects that contain uncompiled SQL statements in cases where a statement is used very few times in a session.

As described in the following sections, the optimal way to set the DYNAMIC_PREPARE connection property and create PreparedStatement objects can depend on whether your application needs to be portable across JDBC drivers or whether you are writing an application that allows jConnect-specific extensions to JDBC.

jConnect provides performance tuning features for dynamic SQL statements.

Choosing prepared statements and stored procedures

If you create a PreparedStatement object containing a precompiled dynamic SQL statement, once the statement is compiled in the database, it effectively becomes a stored procedure that is retained in memory and attached to the data structure associated with your session. In deciding whether to maintain stored procedures in the database or to create PreparedStatement objects containing compiled SQL statements in your application, resource demands and database and application maintenance are important considerations:

- Once a stored procedure is compiled, it is globally available across all connections. In contrast, a dynamic SQL statement in a PreparedStatement object needs to be compiled and deallocated in every session that uses it.
- If your application accesses multiple databases, using stored procedures means that the same stored procedures need to be available on all target databases. This can create a database maintenance problem. If you use PreparedStatement objects for dynamic SQL statements, you avoid this problem.
- If your application creates CallableStatement objects for invoking stored procedures, you can encapsulate SQL code and table references in the stored procedures. You can then modify the underlying database or SQL code without have to change the application.

Prepared statements in portable applications

If your application runs on databases from different vendors and you want some PreparedStatement objects to contain precompiled statements and others to contain uncompiled statements, proceed as follows:

- When you access a Sybase database, make sure that the DYNAMIC_PREPARE connection property is set to "true."
- To return PreparedStatement objects containing precompiled statements, use Connection.prepareStatement in the standard way:

  ```
  PreparedStatement ps_precomp =
     Connection.prepareStatement(sql_string);
  ```

- To return PreparedStatement objects containing uncompiled statements, use Connection.prepareCall.

  Connection.prepareCall returns a CallableStatement object, but because CallableStatement is a subclass of PreparedStatement, you can upcast a CallableStatement object to a PreparedStatement object, as follows:

```
PreparedStatement ps_uncomp =
   Connection.prepareCall(sql_string);
```

The PreparedStatement object *ps_uncomp* is guaranteed to contain an uncompiled statement, because only Connection.prepareStatement is implemented to return PreparedStatement objects containing precompiled statements.

Prepared statements with jConnect extensions

If you are not concerned about portability across drivers, you can write code that uses SybConnection.prepareStatement to specify whether a PreparedStatement object contains precompiled or uncompiled statements. In this case, how you code prepared statements can depend on whether most of the dynamic statements in an application are likely to be executed many times or only a few times during a session.

If most dynamic statements are executed infrequently

For an application in which most dynamic SQL statements are likely to be executed only once or twice in a session:

- Set the connection property DYNAMIC_PREPARE to "false."
- To return PreparedStatement objects containing uncompiled statements, use Connection.prepareStatement in the standard way:

  ```
  PreparedStatement ps_uncomp =
     Connection.prepareStatement(sql_string);
  ```

- To return PreparedStatement objects containing precompiled statements, use SybConnection.prepareStatement with *dynamic* set to "true," as shown:

```
PreparedStatement ps_precomp =
   (SybConnection)conn.prepareStatement(sql_string, true);
```

If most dynamic statements are executed many times in a session

If most of the dynamic statements in an application are likely to be executed many times in the course of a session, proceed as follows:

- Set the connection property DYNAMIC_PREPARE to "true."
- To return PreparedStatement objects containing precompiled statements, use Connection.prepareStatement in the standard way:

```
PreparedStatement ps_precomp = Connection.prepareStatement(sql_string);
```

- To return PreparedStatement objects containing uncompiled statements, you can use either Connection.prepareCall (see Prepared statements in portable applications) or SybConnection.prepareStatement, with *dynamic* set to “false”:

```
PreparedStatement ps_uncomp =
   (SybConnection)conn.prepareStatement(sql_string, false);

PreparedStatement ps_uncomp = Connection.prepareCall(sql_string);
```

Connection.prepareStatement

jConnect implements Connection.prepareStatement so you can set it to return either precompiled SQL statements or uncompiled SQL statements in PreparedStatement objects. If you set Connection.prepareStatement to return precompiled SQL statements in PreparedStatement objects, it sends dynamic SQL statements to the database to be precompiled and saved exactly as they would be under direct execution of the prepare command. If you set Connection.prepareStatement to return uncompiled SQL statements, it returns them in PreparedStatement objects without sending them to the database.

The type of SQL statement that Connection.prepareStatement returns is determined by the connection property DYNAMIC_PREPARE, and applies throughout a session.

For Sybase-specific applications, jConnect 6.05 or later provides a prepareStatement method under the jConnect SybConnection class. SybConnection.prepareStatement allows you to specify whether an individual dynamic SQL statement is to be precompiled, independent of the session-level setting of the DYNAMIC_PREPARE connection property.

DYNAMIC_PREPARE connection property

DYNAMIC_PREPARE is a Boolean-valued connection property for enabling dynamic SQL prepared statements:

- If DYNAMIC_PREPARE is set to "true," every invocation of Connection.prepareStatement during a session attempts to return a precompiled statement in a PreparedStatement object.

In this case, when a PreparedStatement is executed, the statement it contains is already precompiled in the database, with place holders for dynamically assigned values, and the statement needs only to be executed.

- If DYNAMIC_PREPARE is set to "false" for a connection, the PreparedStatement object returned by Connection.prepareStatement does not contain a precompiled statement.

  In this case, each time a PreparedStatement is executed, the dynamic SQL statement it contains must be sent to the database to be both compiled and executed.

The default value for DYNAMIC_PREPARE is "false."

In the following example, DYNAMIC_PREPARE is set to "true" to enable precompilation of dynamic SQL statements. In the example, props is a Properties object for specifying connection properties.

```
...
props.put("DYNAMIC_PREPARE", "true")
Connection conn = DriverManager.getConnection(url, props);
```

When DYNAMIC_PREPARE is set to "true," note that:

- Not all dynamic statements can be precompiled under the prepare command. The SQL-92 standard places some restrictions on the statements that can be used with the prepare command, and individual database vendors may have their own constraints.
- If the database generates an error because it is unable to precompile and save a statement sent to it through Connection.prepareStatement, jConnect traps the error and returns a PreparedStatement object containing an uncompiled dynamic SQL statement. Each time the PreparedStatement object is executed, the statement is re-sent to the database to be compiled and executed.
- A precompiled statement resides in memory in the database and persists either to the end of a session or until its PreparedStatement object is explicitly closed. Garbage collection on a PreparedStatement object does not remove the prepared statement from the database.

As a general rule, you should explicitly close every PreparedStatement object after its last use to prevent prepared statements from accumulating in server memory during a session and slowing performance.

SybConnection.prepareStatement

If your application allows jConnect-specific extensions to JDBC, you can use the SybConnection.prepareStatement extension method to return dynamic SQL statements in PreparedStatement objects:

```
PreparedStatement SybConnection.prepareStatement(String sql_stmt,
   boolean dynamic) throws SQLException
```

SybConnection.prepareStatement can return PreparedStatement objects containing either precompiled or uncompiled SQL statements, depending on the setting of the *dynamic* parameter. If *dynamic* is "true," SybConnection.prepareStatement returns a PreparedStatement object with a precompiled SQL statement. If *dynamic* is "false," it returns a PreparedStatement object with an uncompiled SQL statement.

The following example shows the use of SybConnection.prepareStatement to return a PreparedStatement object containing a precompiled statement:

```
PreparedStatement precomp_stmt = ((SybConnection) conn).prepareStatement
   ("SELECT * FROM authors WHERE au_fname LIKE ?", true);
```

In the example, the connection object *conn* is cast to a SybConnection object to allow the use of SybConnection.prepareStatement. The SQL string passed to SybConnection.prepareStatement is precompiled in the database, even if the connection property DYNAMIC_PREPARE is “false.”

If the database generates an error because it is unable to precompile a statement sent to it through SybConnection.prepareStatement, jConnect throws a SQLException, and the call fails to return a PreparedStatement object. This is unlike Connection.prepareStatement, which traps SQL errors and, in the event of an error, returns a PreparedStatement object containing an uncompiled statement.

ESCAPE_PROCESSING_DEFAULT connection property

By default, jConnect parses all SQL statements submitted to the database for valid JDBC function escapes. If your application is not going to use JDBC function escapes in its SQL calls, you can set this connection property to "false" to circumvent this parsing. This may give a slight performance benefit.

Cursor performance

When you use the Statement.setCursorName method or the setFetchSize( ) method in the SybCursorResultSet class, jConnect creates a cursor in the database. Other methods cause jConnect to open, fetch, and update a cursor.

jConnect creates and manipulates cursors either by sending SQL statements to the database or by encoding cursor commands as tokens within the TDS communication protocol. Cursors of the first type are “language cursors;” cursors of the second type are “protocol cursors.”

Protocol cursors provide better performance than language cursors. In addition, not all databases support language cursors. For example, SQL Anywhere databases do not support language cursors.

In jConnect, the default condition is for all cursors to be protocol cursors. However, the LANGUAGE_CURSOR connection property gives you the option of having cursors created and manipulated through language commands in the database.

LANGUAGE_CURSOR connection property

LANGUAGE_CURSOR is a Boolean-valued connection property in jConnect that allows you to determine whether cursors are created as protocol cursors or language cursors:

- If LANGUAGE_CURSOR is set to "false," all cursors created during a session are protocol cursors, which provide better performance. jConnect creates and manipulates the cursors by sending cursor commands as tokens in the TDS protocol.

  By default, LANGUAGE_CURSOR is set to "false."

- If LANGUAGE_CURSOR is set to "true," all cursors created during a session are language cursors. jConnect creates and manipulates the cursors by sending SQL statements to the database for parsing and compilation.

  There is no known advantage to setting LANGUAGE_CURSOR to "true," but the option is provided in case an application displays unexpected behavior when LANGUAGE_CURSOR is "false."

CHAPTER 6 Migrating jConnect Applications

This chapter explains how to migrate applications from jConnect 5.x and 6.x to jConnect 7.x.

Migrating applications to jConnect 7.x

Use the following procedure to upgrade to jConnect 7.x.

❖ **Migrating to jConnect 7.x**

1 If your code uses Sybase extensions, or if you explicitly import any jConnect class in your code, change package import statements as needed.

For example, change import statements such as

```
import com.sybase.jdbc.*
```

and

```
import com.sybase.jdbc2.jdbc.*
```

to

```
import com.sybase.jdbcx.*
```

For information on using the Sybase extension APIs, see “Changing Sybase extensions” on page 152.

2 Set JDBC_HOME to the top directory of the jConnect driver you installed:

```
JDBC_HOME=jConnect-7_0
```

3 Change your CLASSPATH environment variable to reflect the new installation. Your classpath must include the following:

```
JDBC_HOME/classes/jconn4.jar
```

4 Change the source code where the driver is loaded, and recompile the application to use the new driver:

```
Class.forName("com.sybase.jdbc4.jdbc.SybDriver");
```

5 Verify that the jConnect 7.0 driver is the first jConnect driver specified in your CLASSPATH environment variable.

Changing Sybase extensions

jConnect version 4.1 and later include the package com.sybase.jdbcx that contains all of the Sybase extensions to JDBC. In versions of jConnect previous to 4.1, these extensions were available in the com.sybase.jdbc and com.sybase.utils packages.

The com.sybase.jdbcx package provides a consistent interface across different versions of jConnect. All of the Sybase extensions are defined as Java interfaces, which allow the underlying implementations to change without affecting applications built using these interfaces.

When you develop new applications that use Sybase extensions, use com.sybase.jdbcx. The interfaces in this package allow you to upgrade applications to versions of jConnect that follow version 4.0 with minimal changes.

Some of the Sybase extensions have been changed to accommodate the com.sybase.jdbcx interface.

Extension change example

If an application uses the SybMessageHandler, the code differences would be:

- **jConnect 4.0** code:

```
import com.sybase.jdbc.SybConnection;
import com.sybase.jdbc.SybMessageHandler;
.
.
Connection con = DriverManager.getConnection(url, props);
SybConnection sybCon = (SybConnection) con;
sybCon.setMessageHandler(new ConnectionMsgHandler());
```

- **jConnect 6.0** code:

```
import com.sybase.jdbcx.SybConnection;
import com.sybase.jdbcx.SybMessageHandler;
.
.
Connection con = DriverManager.getConnection(url, props);
SybConnection sybCon = (SybConnection) con;
sybCon.setSybMessageHandler(new ConnectionMsgHandler());
```

See the samples provided with jConnect for more examples of how to use Sybase extensions.

Method names

Table 6-1 lists how methods have been renamed in the new interface.

Table 6-1: Method name changes

Class	Old name	New name
SybConnection	getCapture( )	createCapture( )
SybConnection	setMessageHandler( )	setSybMessageHandler( )
SybConnection	getMessageHandler( )	getSybMessageHandler( )
SybStatement	setMessageHandler( )	setSybMessageHandler( )
SybStatement	getMessageHandler( )	getSybMessageHandler( )

Debug class

Direct static references to the Debug class are no longer supported, but exist in deprecated form in the com.sybase.utils package. To use jConnect debugging facilities, use the getDebug method of the SybDriver class to obtain a reference to the Debug class. For example:

```
import com.sybase.jdbcx.SybDriver;
import com.sybase.jdbcx.Debug;
.
.
.
SybDriver sybDriver =
    SybDriver)Class.forName
    ("com.sybase.jdbc4.jdbc.SybDriver") newInstance();
Debug sybDebug = sybDriver.getDebug();
sybDebug.debug(true, "ALL", System.out);
```

A complete list of Sybase extensions is in the jConnect javadoc documentation located in the *docs/* directory of your jConnect installation directory.

CHAPTER 7 Web Server Gateways

This chapter describes Web server gateways and explains how to use them with jConnect.

About Web server gateways

If your database server runs on a different host than your Web server, or if you are developing Internet applications that must connect to a secure database server through a firewall, you need a gateway to act as a proxy, providing a path to the database server.

To connect to servers using the Secure Sockets Layer (SSL) protocol, jConnect provides a Java servlet that you can install on any Web server that supports the javax.servlet interfaces. This servlet enables jConnect to support encryption using the Web server as the gateway.

Note jConnect includes support for SSL on the client system. For more information, see "Implementing custom socket plug-ins" on page 112.

Using TDS tunnelling

jConnect uses TDS to communicate with database servers. HTTP-tunnelled TDS is useful for forwarding requests. Requests from a client to a back-end server that go through the gateway contain TDS in the body of the request. The request header indicates the length of the TDS included in the request packet.

TDS is a connection-oriented protocol, whereas HTTP is not. To support security features such as encryption for Internet applications, jConnect uses a TDS-tunnelling servlet to maintain a logical connection across HTTP requests. The servlet generates a session ID during the initial login request, and the session ID is included in the header of every subsequent request. Using session IDs lets you identify active sessions and even resume a session, as long as the servlet has an open connection using that specific session ID.

The logical connection provided by the TDS-tunnelling servlet enables jConnect to support encrypted communication between two systems; for example, a jConnect client with the CONNECT_PROTOCOL connection property set to “https” can connect to a Web server running the TDS-tunnelling servlet.

Configuring jConnect and gateways

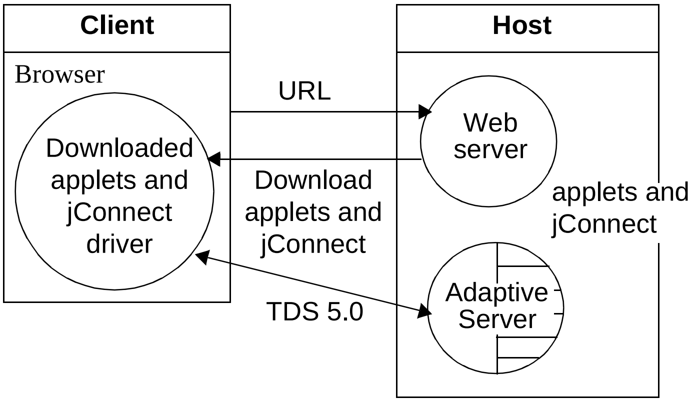
There are several options for setting up your Web servers and Adaptive Servers. The following examples are four common configurations that show where to install the jConnect driver and when to use a gateway with the TDS-tunnelling servlet.

Web server and Adaptive Server on one host

In this two-tier configuration, the Web server and Adaptive Server are both installed on the same host:

- Install jConnect on the Web server host.
- No gateway is required.

Figure 7-1: Web server and Adaptive Server on one host

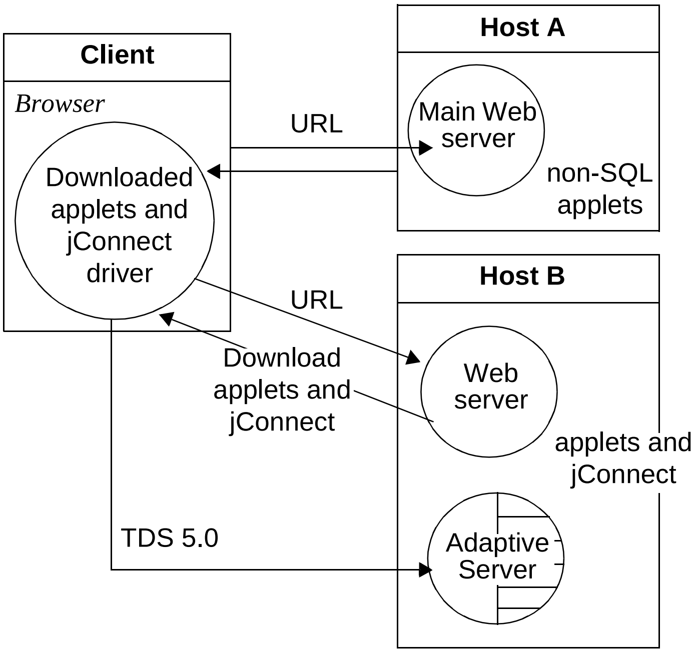


Dedicated JDBC Web server and Adaptive Server on one host

In this configuration, you have a separate host for your main Web server. A second host is shared by a Web server specifically for Adaptive Server access and the Adaptive Server. Links from the main server send requests requiring SQL access to the dedicated Web server. Install on a second host:

- Install jConnect on the second (Adaptive Server) host.
- No gateway is required.

Figure 7-2: Dedicated JDBC Web server and Adaptive Server on one host

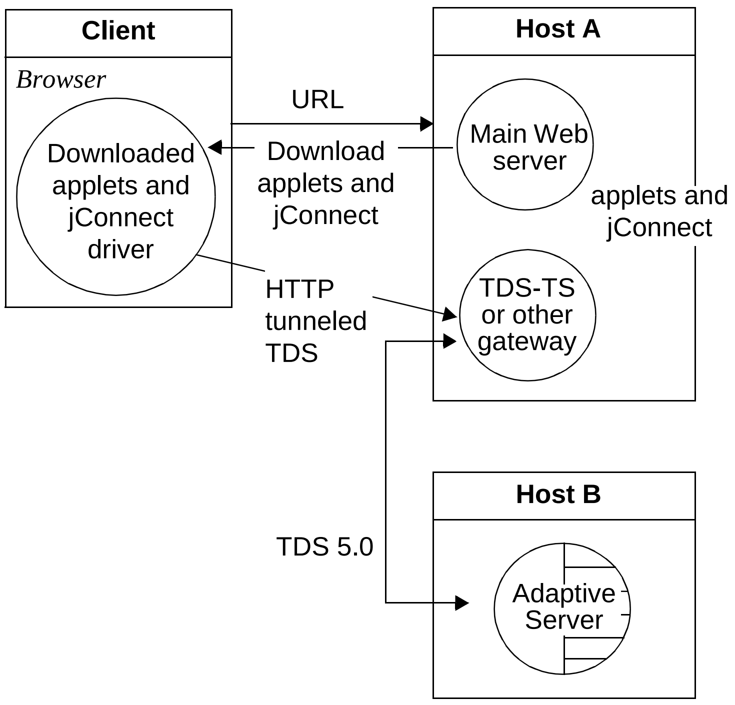


Web server and Adaptive Server on separate hosts

In this three-tier configuration, the Adaptive Server is on a separate host from the Web server. jConnect requires a gateway to act as a proxy to the Adaptive Server.

- Install jConnect on the Web server host.
- Install a TDS-tunnelling servlet or a different gateway.

Figure 7-3: Web server and Adaptive server on separate hosts

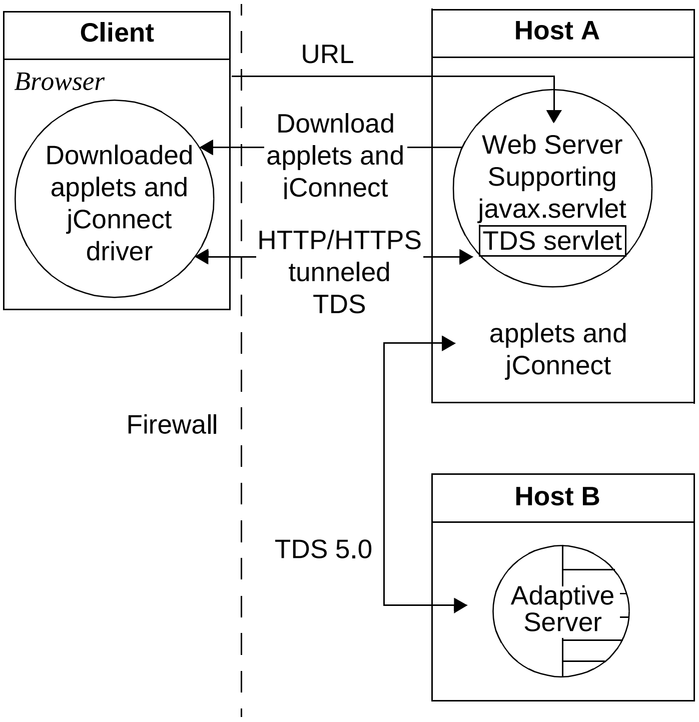


Connecting to a server through a firewall

To connect to a server protected by a firewall, you must use a Web server with the TDS-tunnelling servlet to support transmission of database request responses over the Internet.

- Install jConnect on the Web server host.
- Requires a Web server that supports the javax.servlet interfaces.

Figure 7-4: Connecting to a server through a firewall



Usage requirements

The following sections describe use requirements for Web server gateways.

Reading the *index.html* file

Use your Web browser to view the *index.html* file in your jConnect installation directory. *index.html* provides links to the jConnect documentation and sample code.

Note If you use Netscape on the same machine where you have installed jConnect, be sure that your browser does not have access to your CLASSPATH environment variable. See “Restrictions on Setting CLASSPATH When You Use Netscape” in Chapter 3 of the *Sybase jConnect for JDBC Installation Guide and Release Bulletin*.

❖ **To view the index.html file**

1. Open your Web browser.
2. Enter the URL that matches your setup. For example, if your browser and the Web server are running on the same host, enter:

   ```
   http://localhost:8000/index.html
   ```

   If the browser and the Web server are running on different hosts, enter:

   ```
   http://host:port/index.html
   ```

   where *host* is the name of the host on which the Web server is running, and *port* is the listening port.

Running the sample Isql applet

After loading the *index.html* file in your browser:

❖ **To run the sample applet**

1. Click “Run Sample JDBC Applets.”

   This takes you to the jConnect Sample Programs page.
2. Move down the Sample Programs page to find the table under “Executable Samples.”
3. Locate “Isql.java” in the table and click Run at the end of the row.

The sample Isql.java applet prompts for a simple query on a sample database and displays the results. The applet displays a default Adaptive Server host name, port number, user name (*guest*), password (*sybase*), database, and query. Using the default values, the applet connects to the Sybase demonstration database. It returns results after you click Go.

Troubleshooting

On UNIX, if the applet does not appear as expected, you can modify the applet screen dimensions:

❖ **To modify the applet screen dimensions**

1. Use a text editor to edit the following:

   $JDBC_HOME/sample2/gateway.html

2. Change the height parameter in line 7 to 650. You can experiment with different height settings.
3. Reload the Web page on your browser.

Using the TDS-tunnelling servlet

To use the TDS-tunnelling servlet, you need a Web server that supports the javax.servlet interfaces, such as the Sun Microsystems Java Web server. When you install the Web server, include the jConnect TDS-tunnelling servlet in the list of active servlets. You can also set servlet parameters to define connection timeouts and maximum packet size.

With the TDS-tunnelling servlet, requests from a client to the back-end server that go through the gateway include a GET or POST command, the TDS session ID (after the initial request), back-end address, and status of the request.

TDS is in the body of the request. Two header fields indicate the length of the TDS stream and the session ID assigned by the gateway.

When the client sends a request, the Content-Length header field indicates the size of the TDS content, and the request command is POST. If there is no TDS data in the request because the client is either retrieving the next portion of the response data from the server, or closing the connection, the request command is GET.

The following example illustrates how information is passed between the client and an HTTPS gateway using the TDS-tunneled HTTPS protocol; it shows a connection to a back-end server named "DBSERVER" with a port number of "1234."

Table 7-1: Client to gateway login request. No session ID.

Query	POST/tds?ServerHost=dbserver&ServerPort=1234&Operation=more HTTP/1.0
Header	Content-Length: 605
Content (TDS)	Login request

Table 7-2: Gateway to client. Header contains session ID assigned by the TDS servlet.

Query	200 SUCCESS HTTP/1.0
Header	Content-Length: 210 TDS-Session: TDS00245817298274292
Content (TDS)	Login acknowledgment EED

Table 7-3: Client to gateway. Headers for all subsequent requests contain the session ID.

Query	POST/tds?TDS-Session=TDS00245817298274292&Operation=more HTTP/1.0
Header	Content-Length: 32
Content (TDS)	Query "SELECT * from authors"

Table 7-4: Gateway to client. Headers for all subsequent responses contain the session ID.

Query	200 SUCCESS HTTP/1.0
Header	Content-Length: 2048 TDS-Session: TDS00245817298274292
Content (TDS)	Row format and some rows from query response

Reviewing requirements

To use the jConnect servlet for TDS-tunneled HTTP, you need:

- A Web server that supports javax.servlet interfaces. To install the server, follow the instructions that are provided with it.

Installing the servlet

Your jConnect installation includes a *gateway2* subdirectory under the *classes* directory. The subdirectory contains files required for the TDS-tunnelling servlet.

Copy the jConnect gateway package to a *gateway2* subdirectory under the *servlets* directory of your Web server. After you have copied the servlets, activate the servlets by following the instructions for your Web server.

Setting servlet arguments

When you add the servlet to your Web server, you can enter optional arguments to customize performance:

- *SkipDoneProc [true|false]* – Sybase databases often return row count information while intermediate processing steps are performed during the execution of a query. Usually, client applications ignore this data. If you set *SkipDoneProc* to "true," the servlet removes this extra information from responses “on the fly,” which reduces network usage and processing requirements on the client. This is particularly useful when using HTTPS/SSL, because the unwanted data does not get encrypted/decrypted before it is ignored.
- *TdsResponseSize* – set the maximum TDS packet size for the tunneled HTTPS. A larger *TdsResponseSize* is more efficient if you have only a few users with a large volume of data. Use a smaller *TdsResponseSize* if you have many users making smaller transactions.
- *TdsSessionIdleTimeout* – define the amount of time (in milliseconds) that the server connection can remain idle before the connection is automatically closed. The default *TdsSessionIdleTimeout* is 600,000 (10 minutes).

  If you have interactive client programs that may be idle for long periods of time and you do not want the connections broken, increase the *TdsSessionIdleTimeout*.

  You can also set the connection timeout value from the jConnect client using the SESSION_TIMEOUT connection property. This is useful if you have specific applications that may be idle for longer periods. In this case, set a longer timeout for those connections with the SESSION_TIMEOUT connection property, rather than setting it for the servlet.
- *Debug* – turn on debugging. See “Debugging with jConnect” on page 131.

Enter the servlet arguments in a comma-delimited string. For example:

```
TdsResponseSize=[size],TdsSessionIdleTimeout=
  [timeout],Debug=true
```

Refer to your Web server documentation for complete instructions on entering servlet arguments.

Invoking the servlet

jConnect determines when to use the gateway where the TDS-tunnelling servlet is installed based on the path extension of the *proxy* connection property. jConnect recognizes the servlet path extension to the *proxy* and invokes the servlet on the designated gateway.

Define the connection URL using this format:

```
http://host:port/TDS-servlet-path
```

jConnect invokes the TDS-tunnelling servlet on the Web server to tunnel TDS through HTTP. The servlet path must be the path you defined in the servlet alias list for your Web server.

Tracking active TDS sessions

You can view information about active TDS sessions, including the server connections for each session. Use your Web browser to open the administrative URL:

```
http://host:port/TDS-servlet-path?Operation=list
```

For example, if your server is “myserver” and the TDS servlet path is */tds*, enter:

```
http://myserver:8080/tds?Operation=list
```

This shows you a list of active TDS sessions. You can click on a session to see more information, including the server connection.

Terminating TDS sessions

You can use the URL described above to terminate any active TDS session. Click on an active session from the list of sessions on the first page, then click Terminate This Session.

Resuming a TDS session

You can set the SESSION_ID connection property so that, if necessary, you can resume an existing open connection. When you specify a SESSION_ID, jConnect skips the login phase of the protocol and resumes the connection with the gateway using the designated session ID. If the session ID you specified does not exist on the servlet, jConnect throws a SQL exception the first time you attempt to use the connection.

APPENDIX A SQL Exception and Warning Messages

The following table lists the SQL exception and warning messages that you may encounter when using jConnect.

SQL state	Message/description/action
010AF	`SEVERE WARNING: An assertion has failed, please use devclasses to determine the source of this serious bug. Message = _____.` **Description:** An internal assertion in the jConnect driver has failed. **Action:** Use the devclasses debug classes to determine the reason for this message and report the problem to Sybase Technical Support.
010DF	`Attempt to set database at login failed. Error message:______.` **Description:** jConnect cannot connect to the database specified in the connection URL. **Action:** Be sure the database name in the URL is correct. Also, if connecting to SQL Anywhere, use the SERVICENAME connection property to specify the database.
010DP	`Duplicate connection property _____ ignored.` **Description:** A connection property is defined twice. It may be defined twice in the driver connection properties list with different capitalization, for example “password” and “PASSWORD.” Connection property names are not case-sensitive; therefore, jConnect does not distinguish between property names with the same name but different capitalization. The connection property may also be defined both in the connection properties list, and in the URL. In this case, the property value in the connection property list takes precedence. **Action:** Be sure your application defines the connection property only once. However, you may want your application to take advantage of the precedence of connection properties defined in the property list over those defined in the URL. In this case, you can safely ignore this warning.
010HA	`The server denied your request to use the high-availability feature. Please reconfigure your database, or do not request a high-availability session.` **Description:** The server to which jConnect attempted an HA-enabled connection did not allow the connection. **Action:** Reconfigure the server to support high availability failover or do not set REQUEST_HA_SESSION to "true."

SQL state	Message/description/action
010HD	`Sybase high-availability failover is not supported by this type of database server.` **Description:** The database to which jConnect attempted a connection does not support high availability failover. **Action:** Connect only to database servers that support high availability failover.
010HN	`The client did not specify a SERVICE_PRINCIPAL_NAME Connection property. Therefore, jConnect is using the hostname of _____ as the service principal name` **Action:** Make sure to explicitly specify a service principal name using the connection property.
010HT	`Hostname property truncated, maximum length is 30.` **Description:** You provided a string greater than 30 characters for the HOSTNAME connection property, or the host machine on which the jConnect application is running has a name longer than 30 bytes in length. **Action:** No action is necessary, since this is just a warning to indicate that jConnect is truncating the name to 30 bytes. However, if you wish to avoid this warning, you should set the HOSTNAME to a string less than or equal to 30 bytes in length.
010KF	`The server rejected your Kerberos login attempt. Most likely, this was because of a Generic Security Services (GSS) exception. Please check your Kerberos environment and configuration.` **Action:** Check your Kerberos environment, and make sure that you authenticated properly to the KDC. See Chapter 3, "Security" for more information.
010MX	`Metadata accessor information was not found on this database. Please install the required tables as mentioned in the jConnect documentation. Error encountered while attempting to retrieve metadata information: _____` **Description:** The server may not have the necessary stored procedures for returning metadata information. **Action:** Be sure that stored procedures for providing metadata are installed on the server. See "Installing Stored Procedures" in Chapter 3 of the *jConnect for JDBC Installation Guide*.
010P4	`An output parameter was received and ignored.` **Description:** The query you executed returned an output parameter but the application result-processing code did not fetch it so it was ignored. **Action:** If your application needs the output parameter data, you must rewrite the application so it can get it. This may require using a CallableStatement to execute the query, and adding calls to registerOutputParameter and getXXX. You can also prevent jConnect from returning this warning, and possibly get a performance improvement, by setting the DISABLE_UNPROCESSED_PARAM_WARNINGS connection property to "true."

SQL state	Message/description/action
010PF	`One or more jars specified in the PRELOAD_JARS connection property could not be loaded.` **Description:** This happens when using the DynamicClassLoader with the PRELOAD_JARS connection property set to a comma-delimited list of *.jar* file names. When the DynamicClassLoader opens its connection to the server from which the classes are to be loaded, it attempts to "preload" all the *.jar* files mentioned in this connection property. If one or more of the *.jar* file names specified does not exist on the server, the above error message results. **Action:** Verify that every *.jar* file mentioned in the PRELOAD_JARS connection property for your application exists and is accessible on the server.
010PO	`Property LITERAL_PARAM has been reset to "false" because DYNAMIC_PREPARE was set to "true".` **Description:** If you wish to use precompiled dynamic statements, then you must allow for parameters to be sent to those statements (if the statements take parameters). Setting LITERAL_PARAMS to "true" forces all parameters to be sent as literal values in the SQL that you send to the server. Therefore, you cannot set both properties to "true." **Action:** To avoid this warning, do not set LITERAL_PARAMS to "true" when you wish to use dynamic SQL. See "Performance tuning for prepared statements in dynamic SQL" on page 144 for more information.
010RC	`The requested ResultSet type and concurrency is not supported. They have been converted.` **Description:** You requested a type and concurrency combination for the ResultSet that is not supported. The requested values had to be converted. See "Using cursors with result sets" on page 47 for more information about what ResultSet types and concurrencies are available in jConnect **Action:** Request a type and concurrency combination for the ResultSet that is supported.
010SJ	`Metadata accessor information was not found on this database. Please install the required tables as mentioned in the jConnect documentation.` **Description:** The metadata information is not configured on the server. **Action:** If your application requires metadata, install the stored procedures for returning metadata that come with jConnect (see "Installing Stored Procedures" in Chapter 3 of the *jConnect for JDBC Installation Guide*). If you do not need metadata, set the USE_METADATA property to "false."
010SK	`Database cannot set connection option _____.` **Description:** Your application attempted an operation that the database you are connected to does not support. **Action:** You may need to upgrade your database or make sure that the latest version of metadata information is installed on it.

SQL state	Message/description/action
010SL	`Out-of-date metadata accessor information was found on this database. Ask your database administrator to load the latest scripts.` **Description:** The metadata information on the server is old and needs to be updated. **Action:** Install the stored procedures for returning metadata that come with jConnect (see “Installing Stored Procedures” in Chapter 3 of the *jConnect for JDBC Installation Guide*).
010SM	`This database does not support the initial proposed set of capabilities, retrying.` **Description:** Adaptive Server Enterprise versions 11.9.2 and lower had a bug that caused them to reject logins from clients that requested capabilities that the servers did not have. This warning indicates that jConnect has detected this condition and is retrying the connection using the greatest number of capabilities that the server can accept. When jConnect encounters this bug, it attempts to connect to the server twice. **Action:** Clients can safely ignore this warning, but if they wish to eliminate the warning and ensure that jConnect makes only one connection attempt, they can set the ELIMINATE_010SM connection property to "true." Please note that this property should not be set to "true" when connecting to Adaptive Server versions 12.0 and later.
010SN	`Permission to write to file was denied. File: _____. Error message: _____` **Description:** Permission to write to a file specified in the PROTOCOL_CAPTURE connection property is denied because of a security violation in the VM. This can occur if an applet attempts to write to the specified file. **Action:** If you are attempting to write to the file from an applet, make sure that the applet has access to the target file system.
010SP	`File could not be opened for writing. File: _____. Error message: _____` **Action:** Make sure that the file name is correct and that the file is writable.
010SQ	`The connection or login was refused, retrying connection with the host/port address.` **Description:** The CONNECTION_FAILOVER connection property is set to "true," and jConnect was unable to connect to one of the database servers in the list of servers to which to connect. Therefore, jConnect now tries to connect to the next server in the list. **Action:** No action is required, as long as jConnect is able to connect to another database server. However, you should determine why jConnect was unable to connect to the particular server that caused the connection warning to be issued.

SQL state	Message/description/action
010TP	`The connection’s initial character set,_____, could not be converted by the server. The server’s proposed character set,_____, will be used, with conversions performed by jConnect.` **Description:** The server cannot use the character set initially requested by jConnect, and has responded with a different character set. jConnect accepts the change, and performs the necessary character-set conversions. The message is strictly informational and has no further consequences. **Action:** To avoid this message, set the CHARSET connection property to a character set that the server supports.
010TQ	`jConnect could not determine the server's default character set. This is likely because of a metadata problem. Please install the required tables as mentioned in the jConnect documentation. The connection is defaulting to the ascii_7 character set, which can handle only characters in the range from 0x00 through 0x7F.` **Description:** jConnect could not determine the server's default character set. When this occurs, the only characters that are guaranteed to translate properly are the first 127 ASCII characters. Therefore, jConnect reverts to 7-bit ASCII in this case. The message is strictly informational and has no further consequences. **Action:** Install the stored procedures for returning metadata that comes with jConnect (see “Installing Stored Procedures” in Chapter 3 of the *jConnect for JDBC Installation Guide*).
010UF	`Attempt to execute use database command failed. Error message:_____` **Description:** jConnect cannot connect to the database specified in the connection URL. Two possibilities are: • The name was entered incorrectly in the URL. • USE_METADATA is "true" (the default condition), but the stored procedures for returning metadata have not been installed. As a result, jConnect tried to execute the use *database* command with the database in the URL, but the command failed. This may be because you attempted to access an Adaptive Anywhere database. SQL Anywhere databases do not support the use *database* command. **Action:** Make sure the database name in the URL is correct. Make sure that the stored procedures for returning metadata are installed on the server (see “Installing Stored Procedures” in Chapter 3 of the *jConnect for JDBC Installation Guide* and *jConnect for JDBC Release Bulletin*). If you are attempting to access a SQL Anywhere database, either do not specify a database name in the URL, or set USE_METADATA to “false.”
010UP	`Unrecognized connection property _____ ignored.` **Description:** You attempted to set a connection property in the URL that jConnect does not currently recognize. jConnect ignores the unrecognized property. **Action:** Check the URL definition in your application to make sure it references only valid jConnect driver connection properties.

SQL state	Message/description/action
0100V	`The version of TDS protocol being used is too old. Version: _____` **Description:** The server does not support the required version of the TDS protocol. jConnect requires version 5.0 or later. **Action:** Use a server that supports the required version of TDS. See the system requirements section in the jConnect installation guide for details.
01S08	`This connection has been enlisted in a Global transaction. All pending statements on the current local transaction(if any) have been rolled back.` **Description**: jConnect issues rollback to clear out any current local transactions. This occurs when Global transaction has been enlisted, after issuing XAResource.start(). **Action:** If you have local transactions active prior to issuing the XAResource.start() method, you need to either commit or rollback the local transactions.
01S09	`The local transaction method ___________ cannot be used while a global transaction is active on this connection` **Description:** Warns that a local operation is being performed in the global transaction. An example of a local operation is calling the commit() method on the connection. Other operations that can't be used: rollback(), rollback(Savepoint), setSavepoint(), setSavepoint(String), releaseSavepoint(Savepoint), and setAutoCommit(). **Action:** Local transactions need to be kept separated from global transactions. Make sure to complete all local transactions and their operations prior to starting the Global transaction.
JZ001	`User name property '_____' too long. Maximum length is 30.` **Action:** Do not exceed the 30-byte maximum.
JZ002	`Password property '_____' too long. Maximum length is 30.` **Action:** Do not exceed the 30-byte maximum.
JZ003	`Incorrect URL format. URL: _____` **Action:** Verify the URL format. See “URL connection property parameters” on page 23. If you are using the PROXY connection property, you may get a JZ003 exception while trying to connect if the format for the PROXY property is incorrect. The PROXY format for the Cascade proxy is: *ip_address*:*port_number* The PROXY format for the TDS tunnelling servlet is: http[s]://*host*:*port*/*tunneling_servlet_alias*
JZ004	`User name property missing in DriverManager.getConnection(..., Properties)` **Action:** Provide the required user property.

SQL state	Message/description/action
JZ006	`Caught IOException: _____` **Description:** An unexpected I/O error was detected from a lower layer. When such I/O exceptions are caught, they are rethrown as SQL exceptions using the ERR_IO_EXCEPTION JZ006 sqlstate. These errors are often the result of network communication problems. If the I/O exception causes the database connection to be closed, jConnect chains a JZ0C1 exception to the JZ006. Client applications can look for the JZ0C1 exception in the chain to see if the connection is still usable. **Action:** Examine the text of the original I/O exception message, and proceed from there.
JZ008	`Invalid column index value _____.` **Description:** You have requested a column index value of less than 1 or greater than the highest available value. **Action:** Check call to the getXXX method and the text of the original query, or be sure to call rs.next.
JZ009	`Error encountered in conversion. Error message: _____` **Description:** Some of the possibilities are: • A conversion between two incompatible datatypes was attempted, such as date to int. • There was an attempt to convert a string containing a nonnumeric character to a numeric type. • There was a formatting error, such as an incorrectly formatted time/date string. **Action:** Make sure that the JDBC specification supports the attempted type conversion. Make sure that strings are correctly formatted. If a string contains non-numeric characters, do not attempt to convert it to a numeric type.
JZ00B	`Numeric overflow.` **Description:** You tried to send a BigInteger as a TDS numeric, and the value was too large, or you tried to send a Java long as an int and the value was too large. **Action:** These values cannot be stored in Sybase. For long, consider using a Sybase numeric. There is no solution for Bignum.
JZ00C	`The precision and scale specified cannot accommodate numeric value _____.` **Description:** When using the setBigDecimal method, the BigDecimal value has a precision or scale that exceeds the specified precision or scale. **Action:** Make sure that the specified precision and scale can accommodate the BigDecimal value.
JZ00E	`Attempt to call execute() or executeUpdate() for a statement where setCursorName() has been called.` **Action:** Do not try to call execute or executeUpdate on a statement that has a cursor name set. Use a separate statement to delete or update a cursor. See "Using cursors with result sets" on page 47 for more information

SQL state	Message/description/action
JZ00F	`Cursor name has already been set by setCursorName().` **Action:** Do not set the cursor name twice for a statement. Close the result set of the current cursor statement.
JZ00G	`No column values were set for this row update.` **Description:** You attempted to update a row in which no column values were changed. **Action:** To change column values in a row, call updateXX methods before calling updateRow.
JZ00H	`The result set is not updatable. Use Statement.setResultSetConcurrencyType().` **Action:** To change a result set from read-only to updatable, use the Statement.setResultSetConcurrencyType method or add a for update clause to your SQL select statement.
JZ00I	`Invalid scale. The specified scale must be >=0.` **Description:** The scale value must be greater than zero. **Action:** Be sure the scale value is not negative.
JZ00L	`Login failed. Examine the SQLWarnings chained to this exception for the reason(s).` **Action:** See message text; proceed according to the reason(s) given for the login failure.
JZ00M	`Login timed out. Check that your database server is running on the host and port number you specified. Also check the database server for other conditions (such as a full tempdb) that might be causing it to hang.` **Action:** Follow the recommended actions in the error message.
JZ010	`Unable to deserialize an Object value. Error text: _____` **Action:** Make sure that the Java object from the database implements the Serializable interface and is in your local CLASSPATH variable.
JZ011	`Number format exception encountered while parsing numeric connection property _____.` **Description:** A noninteger value was specified for a numeric connection property. **Action:** Specify an integer value for the connection property.
JZ012	`Internal Error. Please report it to Sybase technical support. Wrong access type for connection property _____.` **Action:** Contact Sybase Technical Support.
JZ013	`Error obtaining JNDI entry: _____` **Action:** Correct the JNDI URL, or make a new entry in the directory service.
JZ014	`You may not setTransactionIsolation(Connection.TRANSACTION_NONE). This level cannot be set; it can only be returned by a server.` **Action:** Check your application code, where it calls Connection.setTransactionIsolation, and verify the value you are passing to the method.

SQL state	Message/description/action
JZ015	`Illegal value set for the GSSMANAGER_CLASS connection property. The property value must be a String or an Object that extends org.ietf.jgss.GSSManager.` **Action:** Check the value to which you set the GSSMANAGER_CLASS property.
JZ0BD	`Out of range or invalid value used for method parameter.` **Action:** Verify that the parameter value in the method is correct.
JZ0BI	**Message:** `setFetchSize: The fetch size should be set with the following restrictions - 0 <= rows <= (maximum number of rows in the ResultSet).` **Description:** The client application has tried to call setFetchSize with an invalid number of rows. **Action:** Verify that you are calling setFetchSize with the parameter falling within the above range of values.
JZ0BP	`Output parameters are not allowed in Batch Update Statements.` **Action:** Examine your application code and check that you did not try to declare an output parameter in your batch.
JZ0BR	`The cursor is not positioned on a row that supports the _____ method.` **Description:** You attempted to call a ResultSet method that is invalid for the current row position (for example, calling insertRow when the cursor is not on the insert row). **Action:** Do not attempt to call a ResultSet method that is invalid for the current row position.
JZ0BS	`Batch Statements not supported.` **Action:** Install or update the jConnect metadata stored procedures on your database with the latest versions.
JZ0BT	`The _____ method is not supported for ResultSets of type _____.` **Description:** You attempted to call a ResultSet method that is invalid for the type of ResultSet. **Action:** Do not attempt to call a ResultSet method that is invalid for the type of ResultSet.
JZ0C0	`Connection is already closed.` **Description:** The application has already called Connection.close on this connection object; it cannot be used any more. **Action:** Fix the code so that connection object references are nulled out when a connection is closed.
JZ0C1	`An IOException occurred which closed the connection.` **Description:** An unrecoverable IOException occurred that caused the connection to be closed. The connection cannot be used for any further database activity. If this exception occurs, it can always be found in an exception chain with the JZ006 Exception (explained earlier). **Action:** Determine the cause of the IOException that disrupted the connection.

SQL state	Message/description/action
JZ0CL	`You must define the CLASS_LOADER property when using the PRELOAD_JARS property.` **Action:** Be sure to specify a CLASS_LOADER when setting PRELOAD_JARS to a non-null value.
JZ0CU	`getUpdateCount can only be called once after a successful call to getMoreResults, or execute methods.` **Description:** As per the JDBC API, getUpdateCount should be called only once per result. **Action:** Be sure your code does not call getUpdateCount more than once per result.
JZ0D4	`Unrecognized protocol in Sybase JDBC URL:_____.` **Description:** You specified a connection URL using a protocol other than TDS, which is the only protocol currently supported by jConnect. **Action:** Check the URL definition. If the URL specifies TDS as a subprotocol, make sure that the entry uses the following format and capitalization: jdbc:sybase:Tds:*host*:*port* If the URL specifies JNDI as a subprotocol, make sure that it starts with: jdbc:sybase:jndi:
JZ0D5	`Error loading protocol _____.` **Action:** Check the settings for the CLASSPATH system variable.
JZ0D6	`Unrecognized version number _____ specified in setVersion. Choose one of the SybDriver.VERSION_* values, and make sure that the version of jConnect that you are using is at or beyond the version you specify.` **Action:** See message text.
JZ0D7	`Error loading url provider _____. Error message: _____` **Action:** Check the JNDI URL to make sure it is correct.
JZ0D8	`Error initializing url provider: _____` **Action:** Check the JNDI URL to make sure it is correct.
JZ0DP	`This statement has no metadata because it was not dynamically prepared. Set the DYNAMIC_PREPARE connection property to true to ensure use of dynamic statements.` **Action:** Refer to the error message.
JZ0EM	`End of data.` **Action:** Please report this error to Sybase Technical Support.

SQL state	Message/description/action
JZ0F1	`Sybase high-availability failover connection was requested but the companion server address is missing.` **Description:** When you set the REQUEST_HA_SESSION connection property to "true," you must also specify a failover server. **Action:** You can specify the secondary server using the SECONDARY_SERVER_HOSTPORT connection property, or you can set the secondary server using JNDI (see “Implementing high availability failover support” on page 39).
JZ0F2	`Sybase high-availability failover has occurred. The current transaction is aborted, but the connection is still usable. Retry your transaction.` **Description:** The back-end database server to which you were connected has gone down, but you have failed over to a secondary server. The database connection is still usable. **Action:** Client code should catch this exception, then restart the transaction from the last committed point. Assuming you properly handle the exception, you can continue executing JDBC calls on the same connection object.
JZ0GC	`Error casting a ____ as a GSSManager. Please check the value you are setting for the GSSMANAGER_CLASS connection property. The value must be a String that specifies the fully qualified class name of a GSSManager implementation. Or, it must be an Object that extends org.ietf.jgss.GSSManager.` **Action:** See message text.
JZ0GN	`Error instantiating the class ___ as a GSSManager. The exception was ____. Please check your CLASSPATH and make sure the GSSMANAGER_CLASS property value refers to a fully qualified class name of a GSSManager implementation.` **Action:** Make sure your CLASSPATH environment variable includes any *.jar* files required by your third-party GSSManager implementation.
JZ0GS	`A Generic Security Services API exception occurred. The major error code is ___. The major error message is ___. The minor error code is ___. The minor error message is ____.` **Action:** Examine the major and minor error codes and messages. Check your Kerberos configuration. See Chapter 3, “Security” for more information.
JZ0H0	`Unable to start thread for event handler; event name = _____.` **Action:** Please report this error to Sybase Technical Support.
JZ0H1	`An event notification was received but the event handler was not found; event name = _____.` **Action:** Please report this error to Sybase Technical Support.

SQL state	Message/description/action
JZ0HC	`Illegal character '_____' encountered while parsing hexadecimal number.` **Description:** A string that is supposed to represent a binary value contains a character that is not in the range (0–9, a–f) that is required for a hexadecimal number. **Action:** Check the character values in the string to make sure they are in the required range.
JZ0I1	`I/O Layer: Error reading stream.` **Description:** The connection was unable to read the amount requested. Most likely, the statement timeout period was exceeded and the connection timed out. **Action:** Increase the statement timeout value.
JZ0I2	`I/O layer: Error writing stream.` **Description:** The connection was unable to write the output requested. Most likely, the statement time-out period was exceeded and the connection timed out. **Action:** Increase the statement time out value.
JZ0I3	`Unknown property. This message indicates an internal product problem. Report this error to Sybase Technical support.` **Action:** Indicates an internal product problem. Please report this error to Sybase Technical Support.
JZ0I5	`An unrecognized CHARSET property was specified: _____.` **Description:** You specified an unsupported character set code for the CHARSET connection property. **Action:** Enter a valid character-set code for the connection property. See "jConnect character-set converters" on page 33.
JZ0I6	`An error occurred converting UNICODE to the charset used by the server. Error message: _____` **Action:** Choose a different character set code for the CHARSET connection property on the jConnect client that can support all the characters you need to send to the server. You may need to install a different character set on the server, too. Also, if you are using jConnect version 6.05 or later, and Adaptive Server Enterprise 12.5 or later, you can send your data to the server as unichar/univarchar datatypes. Please see "Using jConnect to pass Unicode data" on page 32.
JZ0I7	`No response from proxy gateway.` **Description:** The Cascade or security gateway is not responding. **Action:** Be sure the gateway is properly installed and running.
JZ0I8	`Proxy gateway connection refused. Gateway response: _____` **Description:** The Web server/gateway indicated by the PROXY connection property has refused your connection request. **Action:** Check the access and error logs on the proxy to determine why the connection was refused. Be sure the proxy is a JDBC gateway.

SQL state	Message/description/action
JZ0I9	`This InputStream was closed.` **Description:** You tried to read an InputStream obtained from getAsciiStream, getUnicodeStream, or getBinaryStream, but the InputStream was already closed. The stream may have been closed because you moved to another column or cancelled the result set and there were not enough resources to cache the data. **Action:** Increase the cache size or read columns in order.
JZ0IA	`Truncation error trying to send_____.` **Description:** There was a truncation error on character set conversion prior to sending a string. The converted string is longer than the size allocated for it. **Action:** Choose a different character-set code for the CHARSET connection property on the jConnect client that can support all the characters you need to send to the server. You may need to install a different character set on the server, too.
JZ0IR	`getXXX may not be called on a column after it has been updated in the result set with a java.io.Reader.` **Action:** Remove the getXXX call on the ResultSet column which you updated using a Reader.
JZ0IS	`getXXXStream may not be called on a column after it has been updated in the result set.` **Description:** After updating a column in a result set, you attempted to read the updated column value using one of the following SybResultSet methods: getAsciiStream, getUnicodeStream, getBinaryStream. jConnect does not support this usage. **Action:** Do not attempt to fetch input streams from columns you are updating.
JZ0J0	`Offset and/or length values exceed the actual text/image length.` **Action**: Verify that the offset and/or length values you used are correct.
JZ0LC	`You cannot call the ____ method on a ResultSet which is using a language cursor to fetch rows. Try setting the LANGUAGE_CURSOR connection property to false.` **Description:** The application tried to call one of the ResultSet cursor scrolling methods on a ResultSet which was created with a language cursor. **Action:** See the error message.
JZ0NC	`wasNull called without a preceding call to get a column.` **Description:** You can only call wasNull after a call to get a column, such as getInt or getBinaryStream. **Action:** Change the code to move the call to wasNull.
JZ0NE	`Incorrect URL format. URL: _____. Error message: _____` **Action:** Check the format of the URL. Make sure that the port number consists only of numeric characters.

SQL state	Message/description/action
JZ0NF	`Unable to load SybSocketFactory. Make sure that you have spelled the class name correctly, that the package is fully specified, that the class is available in your class path, and that it has a public zero-argument constructor.` **Action:** See message text.
JZ0P1	`Unexpected result type.` **Description:** The database has returned a result that the statement cannot return to the application, or that the application is not expecting at this point. This generally indicates that the application is using JDBC incorrectly to execute the query or stored procedure. If the JDBC application is connected to an Open Server application, it may indicate an error in the Open Server application that causes the Open Server to send unexpected sequences of results. **Action:** Use the com.sybase.utils.Debug(true, "ALL") debugging tools to try to determine what unexpected result is seen, and to understand its causes.
JZ0P4	`Protocol error. This message indicates an internal product problem. Report this error to Sybase technical support.` **Action:** See message text.
JZ0P7	`Column is not cached; use RE-READABLE_COLUMNS property.` **Description:** With the REPEAT_READ connection property set to "false," an attempt was made to reread a column or read a column in the wrong order. When REPEAT_READ is "false," you can only read the column value for a row once, and you can only read columns in ascending column-index order. For example, after reading column 3 for a row, you cannot read its value a second time and you cannot read column 2 for the row. **Action:** Either set REPEAT_READ to "true," or do not attempt to reread a column value and be sure that you read columns in ascending column-index order.
JZ0P8	`The RSMDA Column Type Name you requested is unknown.` **Description:** jConnect cannot determine the name of a column type in the ResultSetMetaData.getColumnTypeName method. **Action:** Be sure that your database has the most recent stored procedures for metadata.
JZ0P9	`A COMPUTE BY query has been detected. That type of result is unsupported and has been cancelled.` **Description:** The query you executed returned COMPUTE results, which are not supported by jConnect. **Action:** Change your query or stored procedure so it does not use COMPUTE BY.
JZ0PA	`The query has been cancelled and the same response discarded.` **Description:** A cancel was probably issued by another statement on the connection. **Action:** Check the chain of SQL exceptions and warnings on this and other statements to determine the cause.

SQL state	Message/description/action
JZ0PB	`The server does not support a requested operation.` **Description:** When jConnect creates a connection with a server, it informs the server of capabilities it wants supported, and the server informs jConnect of the capabilities that it supports. This error message is sent when an application requests an operation that was denied in the original capabilities negotiation. For example, if the database does not support precompilation of dynamic SQL statements, and your code invokes SybConnection.prepareStatement(*sql_stmt*, *dynamic*), and *dynamic* is set to "true," jConnect generates this message. **Action:** Modify your code so that it does not request an unsupported capability.
JZ0PC	`The number and size of parameters in your query require wide table support. But either the server does not offer such support, or it was not requested during the login sequence. Try setting the JCONNECT_VERSION property to >=6 if you wish to request widetable support.` **Description:** You are trying to execute a statement with a large number of parameters, and the server is not configured to handle that many parameters. The number of parameters that can produce this exception varies, depending on the datatypes of the data you are sending. You never get this exception if you are sending 481 or fewer parameters. **Action:** You must run this query against an Adaptive Server 12.5 or later server. When you connect to the database, set the JCONNECT_VERSION property to “6”.
JZ0PD	`The size of the query in your dynamic prepare is large enough that you require widetable support. But either the server does not offer such support, or it was not requested during the login sequence. Try setting the JCONNECT_VERSION property to >=6 if you wish to request widetable support.` **Description:** You are trying to execute a dynamic prepared statement with a large number of parameters, and the server is not configured to handle that many parameters. **Action:** You must run this query against an Adaptive Server 12.5 or later server. When you connect to the database, set the JCONNECT_VERSION property to “6”.
JZ0PE	`The number of columns in your cursor declaration OR the size of your cursor declaration itself are large enough that you require widetable support. But either the server does not offer such support, or it was not requested during the login sequence. Try setting the JCONNECT_VERSION property to >= 6 if you wish to request wide table support.` **Description:** This error can occur when your SELECT statement tries to return data from more than 255 columns, or when the actual length of the SELECT statement is very large (greater than approximately 65500 characters). **Action:** You must run this query against a version 12.5 or later Adaptive Server. When you connect to the database, set the JCONNECT_VERSION property to “6”.

SQL state	Message/description/action
JZ0PN	`Specified port number of ____ was out of range. Port numbers must meet the following conditions: 0<= portNumber <=65535.` **Action:** Check the port number that is specified in the database URL.
JZ0R0	`Result set has already been closed.` **Description:** The ResultSet.close method has already been called on the result set object; you cannot use the result set for anything else. **Action:** Fix the code so that ResultSet object references are set to null whenever a result set is closed.
JZ0R1	`Result set is IDLE as you are not currently accessing a row.` **Description:** The application has called one of the ResultSet.getXXX column-data retrieval methods, but there is no current row; the application has not called ResultSet.next, or ResultSet.next returned "false" to indicate that there is no data. **Action:** Verify that rs.next is set to "true" before calling rs.getXXX.
JZ0R2	`No result set for this query.` **Description:** You used Statement.executeQuery, but the statement returned no rows. **Action:** Use executeUpdate for statements returning no rows.
JZ0R3	`Column is DEAD. This is an internal error. Please report it to Sybase technical support.` **Action:** See message text.
JZ0R4	`Column does not have a text pointer. It is not a text/image column or the column is NULL.` **Description:** You cannot update a text/image column if it is null. A null text/image column does not contain a text pointer. **Action:** Be sure that you are not trying to update or get a text pointer to a column that does not support text/image data. Be sure that you are not trying to update a text/image column that is null. Insert data first, then make the update.
JZ0R5	`The ResultSet is currently positioned beyond the last row. You cannot perform a get* operation to read data in this state.` **Description:** The application has moved the ResultSet row pointer beyond the last row. In this position, there is no data to read, so any get* operations are illegal. **Action:** Alter the code so that it does not attempt to read column data when the ResultSet is positioned beyond the last row.
JZ0RD	`You cannot call any of the ResultSet.get* methods on a row that has been deleted with the deleteRow() method.` **Description:** An application is trying to retrieve data from a row that has been deleted. There is no valid data to be retrieved. **Action:** Alter the code so that the application does not attempt to retrieve data from a deleted row.

SQL state	Message/description/action
JZ0RM	`refreshRow may not be called after updateRow or deleteRow.` **Description:** After updating a row in the database with SybCursorResult.updateRow, or deleting it with SybCursorResult.deleteRow, you used SybCursorResult.refreshRow to refresh the row from the database. **Action:** Do not attempt to refresh a row after updating it or deleting it from the database.
JZ0S0	`Statement state machine: Statement is BUSY.` **Description:** The only time this error is raised is from the Statement.setCursorname method, if the application is trying to set the cursor name when the statement is already in use and has noncursor results that need to be read. **Action:** Set the cursor name on a statement before you execute any queries on it, or call Statement.cancel before setting the cursor name, to make sure that the statement is not busy.
JZ0S1	`Statement state machine: Trying to FETCH on IDLE statement.` **Description:** An internal error occurred on the statement. **Action:** Close the statement and open another one.
JZ0S2	`Statement object has already been closed.` **Description:** The Statement.close method has already been called on the statement object; you cannot use the statement for anything else. **Action:** Fix the application so that statement object references are set to null whenever a statement is closed.
JZ0S3	`The inherited method _____ cannot be used in this subclass.` **Description:** PreparedStatement does not support executeQuery(String), executeUpdate(String), or execute(String). **Action:** To pass a query string, use Statement, not PreparedStatement.
JZ0S4	`Cannot execute an empty (zero-length) query.` **Action:** Do not execute an empty query (““).
JZ0S5	`The local transaction method ____ cannot be used while a global transaction is active on this connection.` **Description:** This exception can occur when using distributed transactions. **Action:** See Chapter 7, “Distributed Transactions,” in the *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*) for more information on diagnosing the problem.
JZ0S6	`The local transaction method _____ cannot be used on a pre-System 12 XAConnection.` **Description:** This exception can occur when using distributed transactions. **Action:** See Chapter 7, “Distributed Transactions,” in the *JDBC 2.0 Optional Package*(formerly the JDBC 2.0 Standard Extension API) for more information on diagnosing the problem.
JZ0S8	`An escape sequence in a SQL Query was malformed: ‘_____’.` **Description:** This error results from bad escape syntax. **Action:** Check JDBC documentation for correct syntax.

SQL state	Message/description/action
JZ0S9	`Cannot execute an empty (zero-length) query.` **Action:** Do not execute an empty query (““).
JZ0SA	`Prepared Statement: Input parameter not set, index: _____.` **Action:** Be sure that each input parameter has a value.
JZ0SB	`Parameter index out of range: _____.` **Description:** You have attempted to get, set, or register a parameter that goes beyond the maximum number of parameters. **Action:** Check the number of parameters in your query.
JZ0SC	`Callable Statement: attempt to set the return status as an InParameter.` **Description:** You have prepared a call to a stored procedure that returns a status, but you are trying to set parameter 1, which is the return status. **Action:** Parameters that you can set start at 2 with this type of call.
JZ0SD	`No registered parameter found for output parameter.` **Description:** This indicates an application logic error. You attempted to call getXXX or wasNull on a parameter, but you have not read any parameters yet, or there are no output parameters. **Action:** Check to make sure that the application has registered output parameters on the CallableStatement, that the statement has been executed, and that the output parameters were read.
JZ0SE	`Invalid object type specified for setObject().` **Description:** Illegal type argument passed to PreparedStatement.setObject. **Action:** Check the JDBC documentation. The argument must be a constant from java.sql.Types.
JZ0SF	`No Parameters expected. Has query been sent?` **Description:** You tried to set a parameter on a statement with no parameters. **Action:** Make sure the query has been sent before you set the parameters.

JZ0SG	`An RPC did not return as many output parameters as the application had registered for it.`
	Description: This error occurs if you call CallableStatement.registerOutParam for more parameters than you declared as “OUTPUT” parameters in the stored procedure. See “RPC returns fewer output parameters than registered” on page 138 for more information.
	Action: Check your stored procedures and registerOutParameter calls. Make sure that you have declared all of the appropriate parameters as “OUTPUT.” Look at the line of code that reads:
	`create procedure yourproc (@p1 int OUTPUT, ...`
	Note If you receive this error while using SQL Anywhere, upgrade to SQL Anywhere version 5.5.04.

SQL state	Message/description/action
JZ0SH	`A static function escape was used, but the metadata accessor information was not found on this server.` **Action:** Install metadata accessor information before using static function escapes.
JZ0SI	`A static function escape _____ was used which is not supported by this server.` **Action:** Do not use this escape.
JZ0SJ	`Metadata accessor information was not found on this database.` **Action:** Install metadata information before making metadata calls.
JZ0SK	`The oj escape is not supported for this type of database server. Workaround: use server-specific outer join syntax, if supported. Consult server documentation.` **Action:** Read the error message. Also, install the latest version of the jConnect metadata.
JZ0SL	`Unsupported SQL type _____.` **Description:** The application has declared a parameter to be of a type that jConnect does not support. **Action:** If possible, try declaring the parameter to be of a different type. Do not use Types.NULL or PreparedStatement.setObject (null).
JZ0SM	`jConnect could not execute a stored procedure because there was a problem sending the parameter(s). This problem was likely caused because the server does not support a specific datatype, or because jConnect did not request support for that datatype at connect time. Try setting the JCONNECT_VERSION connection property to a higher value. Or, if possible, try sending your procedure execution command as a language statement.`
JZ0SN	`setMaxFieldSize: field size cannot be negative.` **Action:** Use a positive value or zero (unlimited) when calling setMaxFieldSize.
JZ0SO	`Invalid ResultSet concurrency type:_____.` **Action:** Check that your declared concurrency is either ResultSet.CONCUR_READ_ONLY or ResultSet.CONCUR_UPDATABLE.
JZ0SP	`Invalid ResultSet type:_____.` **Action:** Check that your declared ResultSet type is ResultSet.TYPE_FORWARD_ONLY or ResultSet.TYPE_SCROLL_INSENSITIVE. jConnect does not support the ResultSet.TYPE_SCROLL_SENSITIVE ResultSet type.
JZ0SQ	`In valid UDT type _____.` **Description:** When calling the DatabaseMetaData.getUDTs method, jConnect throws this exception if the user-defined type is not either Types.JAVA_OBJECT, Types.STRUCT or Types.DISTINCT. **Action:** Use one of the three UDTs mentioned above.

SQL state	Message/description/action
JZ0SR	`setMaxRows: max rows cannot be negative.` **Action:** Use a positive value or zero (unlimited) when calling setMaxRows.
JZ0SS	`setQueryTimeout: query timeout cannot be negative.`
JZ0ST	`jConnect cannot send a Java object as a literal parameter in a query. Make sure that your database server supports Java objects and that the LITERAL_PARAMS connection property is set to false when you execute this query.`
JZ0SU	`A Date or Timestamp parameter was set with a year of ______, but the server can only support year values between _____ and _______. If you're trying to send data to date or timestamp columns or parameters on Adaptive Server Anywhere, you may wish to send your data as Strings, and let the server convert them.` **Description:** Adaptive Server Enterprise and SQL Anywhere have different allowable ranges for datetime and date values. datetime values must have years greater or equal to 1753. The date datatype, however, can hold years greater or equal to 1. **Action:** Make sure that the date/timestamp value you are sending falls in the acceptable range.
JZ0T2	`Listener thread read error.` **Action:** Check your network communications.
JZ0T3	`Read operation timed out.` **Description:** The time allotted to read the response to a query was exceeded. **Action:** Increase the timeout period by calling Statement.setQueryTimeout.
JZ0T4	`Write operation timed out. Timeout in milliseconds: _____.` **Description:** The time allotted to send a request was exceeded. **Action:** Increase the timeout period by calling Statement.setQueryTimeout.
JZ0T5	`Cache used to store responses is full.` **Action:** Use default or larger value for the STREAM_CACHE_SIZE connection property.
JZ0T6	`Error reading tunneled TDS URL.` **Description:** The tunneled protocol failed while reading the URL header. **Action:** Check the URL you defined for the connection.
JZ0T7	`Listener thread read error -- caught ThreadDeath. Check network connection.` **Action:** Check the network connections and try to run the application again. If the threads continue to be aborted, please contact Sybase Technical Support.
JZ0T8	`Data received for an unknown request. Please report this error to Sybase Technical Support.`
JZ0T9	`Request to send not synchronized. Please report this error to Sybase Technical Support.` **Action:** See message text.

SQL state	Message/description/action
JZ0TC	`Attempted conversion between an illegal pair of types.` **Description:** Conversion between a Java type and a SQL type failed. **Action:** Check the requested type conversion to make sure it is supported in the JDBC specification.
JZ0TE	`Attempted conversion between an illegal pair of types. Valid database types are: '__________.'` **Description:** The database column datatype and the datatype requested in theResultSet.getXXX call are not implicitly convertible. **Action:** Use one of the valid datatypes listed in the error message.
JZ0TI	`jConnect cannot make a meaningful conversion between the database type of _________ and the requested type of _________.` **Description:** This kind of exception can occur, for example, if an application tries to call ResultSet.getObject(int, Types.DATE) on a time value that is returned from the database. **Action:** Make sure that the database datatype is implicitly convertible to the Object type you wish to retrieve.
JZ0TO	`Read operation timed out.` **Description:** This exception occurs when there is a socket read timeout. **Action:** Increase the timeout period by calling Statement.setQueryTimeout. Also, check the query or stored procedure you are executing to determine why it is taking longer than expected.
JZ0TS	`Truncation error trying to send __________.` **Description:** The application specified a string that was longer than the length that the application wanted to send. Therefore, the string is truncated to the declared length. **Action:** Set the length properly to avoid truncation.
JZ0US	`The SybSocketFactory connection property was set, and the PROXY connection property was set to the URL of a servlet. The jConnect driver does not support this combination. If you want to send secure HTTP from an applet running within a browser, use a proxy URL beginning with "https://".` **Action:** See message text.
JZ0XC	`____________ is an unrecognized transaction coordinator type.` **Description:** The metadata information indicates that the server supports distributed transactions, but jConnect does not support the protocol being used. **Action:** Verify that you have installed the latest metadata scripts. If the error persists, please contact Sybase Technical Support.

SQL state	Message/description/action
JZ0XS	`The server does not support XA-style transactions. Please verify that the transaction feature is enabled and licensed on this server.` **Description:** The server to which jConnect attempted a connection does not support distributed transactions. **Action:** Do not use XADataSource with this server, or upgrade or configure the server for distributed transactions.
JZ0XU	`Current user does not have permission to do XA-style transactions. Be sure user has _______ role.` **Description:** The user connected to the database is not authorized to conduct distributed transactions, most likely because the user does not have the proper role (shown in the blank). **Action:** Grant the user the role shown in the error message, or have another user with that role conduct the transaction.
S0022	`Invalid column name '_____'.` **Description:** You attempted to reference a column by name and there is no column with that name. **Action:** Check the spelling of the column name.
ZZ00A	`The method _____ has not been completed and should not be called.` **Description:** You attempted to use a method that is not implemented. **Action:** Check the release bulletin that came with your version of jConnect for further information. You can also check the jConnect Web page at http://www.sybase.com to see whether a more recent version of jConnect implements the method. If not, do not use the method.

APPENDIX B jConnect Sample Programs

This appendix is a guide to jConnect sample programs:

Running IsqlApp

IsqlApp allows you to issue isql commands from the command line, and run jConnect sample programs.

The syntax for IsqlApp is:

```
IsqlApp [-U username]
   [-P password]
   [-S servername]
   [-G gateway]
   [-p {http|https}]
   [-D debug_class_list]
   [-v]
   [-I input_command_file]
   [-c command_terminator]
   [-C charset]
   [-L language]
   [-K service_principal_name]
   [-F JAAS_login_config_file_path]
   [-T sessionID]
   [-V <version {2,3,4,5}>]
```

Parameter	Description
-U	The login ID with which you want to connect to a server.
-P	The password for the specified login ID.
-S	The name of the server to which you want to connect.

Parameter	Description
-G	The gateway address. For the HTTP protocol, the URL is: http://*host:port*. To use the HTTPS protocol that supports encryption, the URL is https://*host*:*port*/*servlet_alias*.
-p	Specifies whether you want to use the HTTP protocol or the HTTPS protocol that supports encryption.
-D	Turns on debugging for all classes or for just the ones you specify, separated by a comma. For example, *-D ALL* displays debugging output for all classes. *-D SybConnection, Tds* displays debugging output only for the SybConnection and Tds classes.
-v	Turns on verbose output for display or printing.
-I	Causes IsqlApp to take commands from a file instead of the keyboard. After the parameter, you specify the name of the file to use for the IsqlApp input. The file must contain command terminators ("go" by default).
-c	Lets you specify a keyword (for example, "go") that, when entered on a line by itself, terminates the command. This lets you enter multiline commands before using the terminator keyword. If you do not specify a command terminator, each new line terminates a command.
-C	Specifies the character set for strings passed through TDS. If you do not specify a character set, IsqlApp uses the default charset of the server.
-L	Specifies the language in which to display error messages returned from the server and for jConnect messages.
-K	Indicates the user wants to make a Kerberos login to Adaptive Server. This parameter sets the service principal name. For example: `-K myASE` This example indicates that you wish to perform a Kerberos login and that the service principal name for your server is myASE. See Chapter 3, "Security" for more information.
-F	Specifies the path to the JAAS login configuration file. You must set this property if you use the -K option. For example: `-F /foo/bar/exampleLogin.conf` See the *ConnectKerberos.java* sample in the *sample2* directory of your jConnect installation. For more information, see Chapter 3, "Security".
-T	When this parameter is set, jConnect assumes that an application is trying to resume communication on an existing TDS session held open by the TDS-tunnelling gateway. jConnect skips the login negotiations and forwards all requests from the application to the specified session ID.

Parameter	Description
-V	Enables the use version-specific characteristics. See “Using JCONNECT_ VERSION” on page 6.

Note You must enter a space after each option flag.

To obtain a full description of the command-line options, enter:

```
java IsqlApp -help
```

The following example shows how to connect to a database on a host named “myserver” through port “3756”, and run an isql script named “myscript”:

```
java IsqlApp -U sa -P sapassword
   -S jdbc:sybase:Tds:myserver:3756
   -I $JDBC_HOME/sp/myscript -c run
```

Note An applet that provides GUI access to isql commands is available as:
$JDBC_HOME/sample2/gateway.html (UNIX)
%JDBC_HOME%\sample2\gateway.html (Windows)

Running jConnect sample programs and code

jConnect includes several sample programs that illustrate many of the topics covered in this chapter and that are intended to help you understand how jConnect works with various JDBC classes and methods. In addition, this section includes a sample code fragment for your reference.

Sample applications

When you install jConnect, you can also the install sample programs. These samples include the source code so that you can review how jConnect implements various JDBC classes and methods. See the *jConnect for JDBC Installation Guide* for complete instructions for installing the sample programs.

Note The jConnect sample programs are intended for demonstration purposes only.

The sample programs are installed in the *sample2* subdirectory under your jConnect installation directory. The file *index.html* in the *sample2* subdirectory contains a complete list of the samples that are available along with a description of each sample. *index.html* also lets you view and run the sample programs as applets.

Running the sample applets

Using your Web browser, you can run some of the sample programs as applets. This enables you to view the source code while viewing the output results.

Enter http://localhost:8000/sample2/index.html on a Web browser to start the Web server gateway if you want to run the sample programs as applets.

Running the sample programs with SQL Anywhere

All of the sample programs are compatible with Adaptive Server, but only a limited number are compatible with SQL Anywhere. Refer to *index.html* in the *sample2* subdirectory for a current list of the sample programs that are compatible with SQL Anywhere.

To run the sample programs that are available for SQL Anywhere, you must install the *pubs2_any.sql* script on your SQL Anywhere server. This script is located in the *sample2* subdirectory.

For Windows, go to DOS command window and enter:

```
java IsqlApp -U dba -P password
 -S jdbc:sybase:Tds:[hostname]:[port]
 -I %JDBC_HOME%\sample2\pubs2_any.sql -c go
```

For UNIX, enter:

```
java IsqlApp -U dba -P password
```

```
-S jdbc:sybase:Tds:[hostname]:[port]
-I $JDBC_HOME/sample2/pubs2_any.sql -c go
```

Sample code

The following sample code illustrates how to invoke the jConnect driver, make a connection, issue a SQL statement, and process results.

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

   public class SampleCode
   {
       public static void main(String args[])
       {
           try
           {
           /*
            * Open the connection. May throw a SQLException.
            */
             DriverManager.registerDriver(
             (Driver) Class.forName(
              "com.sybase.jdbc4.jdbc.SybDriver").newInstance());

              Connection con = DriverManager.getConnection(
                 "jdbc:sybase:Tds:myserver:3767", "sa", "");
           /*
            * Create a statement object, the container for the SQL
            * statement. May throw a SQLException.
            */
             Statement stmt = con.createStatement();
           /*
            * Create a result set object by executing the query.
            * May throw a SQLException.
            */
             ResultSet rs = stmt.executeQuery("Select 1");
           /*
            * Process the result set.
            */

             if (rs.next())
             {
                int value = rs.getInt(1);
                System.out.println("Fetched value " + value);
             }
            rs.close()
            stmt.close()
            con.close()
         }//end try
      /*
       * Exception handling.
       */
         catch (SQLException sqe)
         {
             System.out.println("Unexpected exception : " +
                         sqe.toString() + ", sqlstate = " +
                             sqe.getSQLState());
             System.exit(1);
         }//end catch


         catch (Exception e)
         {
             e.printStackTrace();
             System.exit(1);
         }//end catch

         System.exit(0);
     }
  }
```

Index